IV Multidimensional and Spatial Structures

16

Multidimensional Spatial Data Structures

Hanan Samet
University of Maryland

16.1 Introduction

The representation of multidimensional data is an important issue in applications in diverse fields that include database management systems (Chapter 60), computer graphics (Chapter 54), computer vision, computational geometry (Chapters 62, 63 and 64), image processing (Chapter 57), geographic information systems (GIS) (Chapter 55), pattern recognition, VLSI design (Chapter 52), and others. The most common definition of multidimensional data is a collection of points in a higher dimensional space. These points can represent locations and objects in space as well as more general records where only some, or even none, of the attributes are locational. As an example of nonlocational point data, consider an employee record which has attributes corresponding to the employee's name, address, sex, age, height, weight, and social security number. Such records arise in database management systems and can be treated as points in, for this example, a seven-dimensional space (i.e., there is one dimension for each attribute) albeit the different dimensions have different type units (i.e., name and address are strings of characters, sex is binary; while age, height, weight, and social security number are numbers).

When multidimensional data corresponds to locational data, we have the additional property that all of the attributes have the same unit which is distance in space. In this case, we can combine the distance-denominated attributes and pose queries that involve proximity. For example, we may wish to find the closest city to Chicago within the two-dimensional space from which the locations of the cities are drawn. Another query seeks to find all cities within 50 miles of Chicago. In contrast, such queries are not very meaningful when the attributes do not have the same type.

[*] All figures ©2003 by Hanan Samet.

When multidimensional data spans a continuous physical space (i.e., an infinite collection of locations), the issues become more interesting. In particular, we are no longer just interested in the locations of objects, but, in addition, we are also interested in the space that they occupy (i.e., their extent). Some example objects include lines (e.g., roads, rivers), regions (e.g., lakes, counties, buildings, crop maps, polygons, polyhedra), rectangles, and surfaces. The objects may be disjoint or could even overlap. One way to deal with such data is to store it explicitly by parameterizing it and thereby reduce it to a point in a higher dimensional space. For example, a line in two-dimensional space can be represented by the coordinate values of its endpoints (i.e., a pair of x and a pair of y coordinate values) and then stored as a point in a four-dimensional space (e.g., [33]). Thus, in effect, we have constructed a transformation (i.e., mapping) from a two-dimensional space (i.e., the space from which the lines are drawn) to a four-dimensional space (i.e., the space containing the representative point corresponding to the line).

The transformation approach is fine if we are just interested in retrieving the data. It is appropriate for queries about the objects (e.g., determining all lines that pass through a given point or that share an endpoint, etc.) and the immediate space that they occupy. However, the drawback of the transformation approach is that it ignores the geometry inherent in the data (e.g., the fact that a line passes through a particular region) and its relationship to the space in which it is embedded.

For example, suppose that we want to detect if two lines are near each other, or, alternatively, to find the nearest line to a given line. This is difficult to do in the four-dimensional space, regardless of how the data in it is organized, since proximity in the two-dimensional space from which the lines are drawn is not necessarily preserved in the four-dimensional space. In other words, although the two lines may be very close to each other, the Euclidean distance between their representative points may be quite large, unless the lines are approximately the same size, in which case proximity is preserved (e.g., [69]).

Of course, we could overcome these problems by projecting the lines back to the original space from which they were drawn, but in such a case, we may ask what was the point of using the transformation in the first place? In other words, at the least, the representation that we choose for the data should allow us to perform operations on the data. Thus when the multidimensional spatial data is nondiscrete, we need representations besides those that are designed for point data. The most common solution, and the one that we focus on in the rest of this chapter, is to use data structures that are based on spatial occupancy. Such methods decompose the space from which the spatial data is drawn (e.g., the two-dimensional space containing the lines) into regions that are often called *buckets* because they often contain more than just one element. They are also commonly known as *bucketing methods*.

In this chapter, we explore a number of different representations of multidimensional data bearing the above issues in mind. While we cannot give exhaustive details of all of the data structures, we try to explain the intuition behind their development as well as to give literature pointers to where more information can be found. Many of these representations are described in greater detail in [60, 62, 63] including an extensive bibliography. Our approach is primarily a descriptive one. Most of our examples are of two-dimensional spatial data although the representations are applicable to higher dimensional spaces as well.

At times, we discuss bounds on execution time and space requirements. Nevertheless, this information is presented in an inconsistent manner. The problem is that such analyses are very difficult to perform for many of the data structures that we present. This is especially true for the data structures that are based on spatial occupancy (e.g., quadtree (see Chapter 19 for more details) and R-tree (see Chapter 21 for more details) variants). In particular, such methods have good observable average-case behavior but may have very bad worst cases which may only arise rarely in practice. Their analysis is beyond the scope of this chapter and usually we do not say anything about it. Nevertheless, these representations find frequent use in applications where their behavior is deemed acceptable, and is often found to be better than that of solutions whose theoretical behavior would appear to be superior. The problem is primarily attributed to the presence of large constant factors which are usually ignored in the *big* O and Ω analyses [46].

The rest of this chapter is organized as follows. Section 16.2 reviews a number of representations of point data of arbitrary dimensionality. Section 16.3 describes bucketing methods that organize collections of spatial objects (as well as multidimensional point data) by aggregating the space that they occupy. The remaining sections focus on representations of non-point objects of different types. Section 16.4 covers representations of region data, while Section 16.5 discusses a subcase of region data which consists of collections of rectangles. Section 16.6 deals with curvilinear data which also includes polygonal subdivisions and collections of line segments. Section 16.7 contains a summary and a brief indication of some research issues.

16.2 Point Data

The simplest way to store point data of arbitrary dimension is in a sequential list. Accesses to the list can be sped up by forming sorted lists for the various attributes which are known as *inverted lists* (e.g., [45]). There is one list for each attribute. This enables pruning the search with respect to the value of one of the attributes. It should be clear that the inverted list is not particularly useful for multidimensional range searches. The problem is that it can only speed up the search for one of the attributes (termed the *primary* attribute). A widely used solution is exemplified by the *fixed-grid* method [10, 45]. It partitions the space from which the data is drawn into rectangular cells by overlaying it with a grid. Each grid cell c contains a pointer to another structure (e.g., a list) which contains the set of points that lie in c. Associated with the grid is an access structure to enable the determination of the grid cell associated with a particular point p. This access structure acts like a directory and is usually in the form of a d-dimensional array with one entry per grid cell or a tree with one leaf node per grid cell.

There are two ways to build a fixed grid. We can either subdivide the space into equal-sized intervals along each of the attributes (resulting in congruent grid cells) or place the subdivision lines at arbitrary positions that are dependent on the underlying data. In essence, the distinction is between organizing the data to be stored and organizing the embedding space from which the data is drawn [55]. In particular, when the grid cells are congruent (i.e., equal-sized when all of the attributes are locational with the same range and termed a *uniform grid*), use of an array access structure is quite simple and has the desirable property that the grid cell associated with point p can be determined in constant time. Moreover, in this case, if the width of each grid cell is twice the search radius for a rectangular range query, then the average search time is $O(F \cdot 2^d)$ where F is the number of points that have been found [12]. Figure 16.1 is an example of a uniform-grid representation for a search radius equal to 10 (i.e., a square of size 20×20).

Use of an array access structure when the grid cells are not congruent requires us to have a way of keeping track of their size so that we can determine the entry of the array access structure corresponding to the grid cell associated with point p. One way to do this is to make use of what are termed *linear scales* which indicate the positions of the grid lines (or partitioning hyperplanes in $d > 2$ dimensions). Given a point p, we determine the grid cell in which p lies by finding the "coordinate values" of the appropriate grid cell. The linear

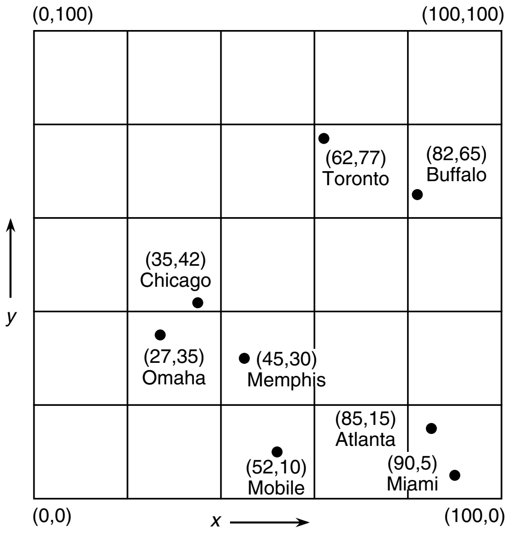


FIGURE 16.1: Uniform-grid representation corresponding to a set of points with a search radius of 20.

scales are usually implemented as one-dimensional trees containing ranges of values.

The array access structure is fine as long as the data is static. When the data is dynamic, it is likely that some of the grid cells become too full while other grid cells are empty. This means that we need to rebuild the grid (i.e., further partition the grid or reposition the grid partition lines or hyperplanes) so that the various grid cells are not too full. However, this creates many more empty grid cells as a result of repartitioning the grid (i.e., empty grid cells are split into more empty grid cells). The number of empty grid cells can be reduced by merging spatially-adjacent empty grid cells into larger empty grid cells, while splitting grid cells that are too full, thereby making the grid adaptive. The result is that we can no longer make use of an array access structure to retrieve the grid cell that contains query point p. Instead, we make use of a tree access structure in the form of a k-ary tree where k is usually 2^d. Thus what we have done is marry a k-ary tree with the fixed-grid method. This is the basis of the point quadtree [22] and the PR quadtree [56, 63] which are multidimensional generalizations of binary trees.

The difference between the point quadtree and the PR quadtree is the same as the difference between *trees* and *tries* [25], respectively. The binary search tree [45] is an example of the former since the boundaries of different regions in the search space are determined by the data being stored. Address computation methods such as radix searching [45] (also known as digital searching) are examples of the latter, since region boundaries are chosen from among locations that are fixed regardless of the content of the data set. The process is usually a recursive halving process in one dimension, recursive quartering in two dimensions, etc., and is known as *regular decomposition*.

In two dimensions, a point quadtree is just a two-dimensional binary search tree. The first point that is inserted serves as the root, while the second point is inserted into the relevant quadrant of the tree rooted at the first point. Clearly, the shape of the tree depends on the order in which the points were inserted. For example, Figure 16.2 is the point quadtree corresponding to the data of Figure 16.1 inserted in the order `Chicago`, `Mobile`, `Toronto`, `Buffalo`, `Memphis`, `Omaha`, `Atlanta`, and `Miami`.

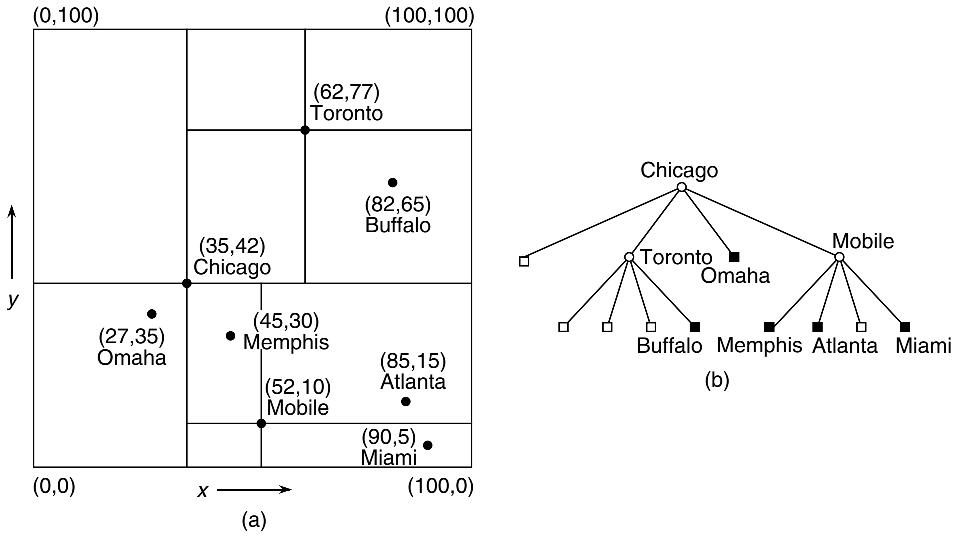


FIGURE 16.2: A point quadtree and the records it represents corresponding to Figure 16.1: (a) the resulting partition of space, and (b) the tree representation.

In two dimensions, the PR quadtree is based on a recursive decomposition of the underlying space into four congruent (usually square in the case of locational attributes) cells until each cell contains no more than one point. For example, Figure 16.3a is the partition of the underlying space induced by the PR quadtree corresponding to the data of Figure 16.1, while Figure 16.3b is its tree representation. The shape of the PR quadtree is independent of the order in which data points are inserted into it. The disadvantage of the PR quadtree is that the maximum level of decomposition depends on the minimum separation between two points. In particular, if two points are very close, then the decomposition can be very deep. This can be overcome by viewing the cells or nodes as buckets with capacity c and only decomposing a cell when it contains more than c points.

As the dimensionality of the space increases, each level of decomposition of the quadtree results in many new cells as the fanout value of the tree is high (i.e., 2^d). This is alleviated by making use of a *k-d tree* [8]. The k-d tree is a binary tree where at each level of the tree, we subdivide along a different attribute so that, assuming d locational attributes, if the first split is along the x axis, then after d levels, we cycle back and again split along the x axis. It is applicable to both the point quadtree and the PR quadtree (in which case we have a *PR k-d tree*, or a bintree in the case of region data).

At times, in the dynamic situation, the data volume becomes so large that a tree access structure such as the one used in the point and PR quadtrees is inefficient. In particular, the grid cells can become so numerous that they cannot all fit into memory thereby causing them to be grouped into sets (termed *buckets*) corresponding to physical storage units (i.e., pages) in secondary storage. The problem is that, depending on the implementation of the tree access structure, each time we must follow a pointer, we may need to make a disk access. Below, we discuss two possible solutions: one making use of an array access structure and one making use of an alternative tree access structure with a much larger fanout. We assume that the original decomposition process is such that the data is only associated with the leaf nodes of the original tree structure.

The difference from the array access structure used with the static fixed-grid method

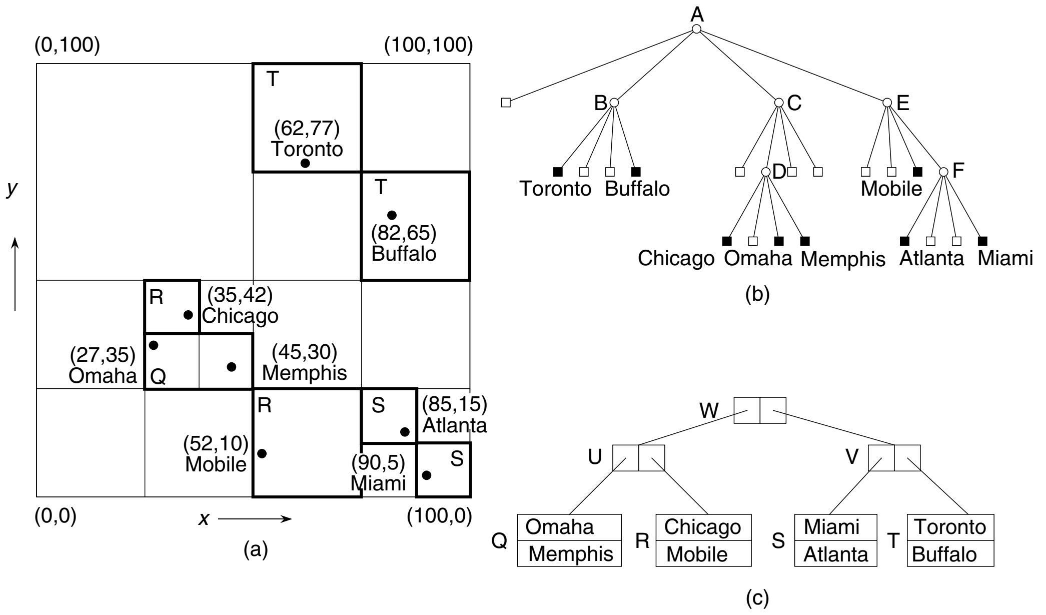


FIGURE 16.3: A PR quadtree and the points it represents corresponding to Figure 16.1: (a) the resulting partition of space, (b) the tree representation, and (c) one possible B$^+$-tree for the nonempty leaf grid cells where each node has a minimum of 2 and a maximum of 3 entries. The nonempty grid cells in (a) have been labeled with the name of the B$^+$-tree leaf node in which they are a member.

described earlier is that the array access structure (termed *grid directory*) may be so large (e.g., when d gets large) that it resides on disk as well, and the fact that the structure of the grid directory can change as the data volume grows or contracts. Each grid cell (i.e., an element of the grid directory) stores the address of a bucket (i.e., page) that contains the points associated with the grid cell. Notice that a bucket can correspond to more than one grid cell. Thus any page can be accessed by two disk operations: one to access the grid cell and one more to access the actual bucket.

This results in *EXCELL* [71] when the grid cells are congruent (i.e., equal-sized for locational data), and *grid file* [55] when the grid cells need not be congruent. The difference between these methods is most evident when a grid partition is necessary (i.e., when a bucket becomes too full and the bucket is not shared among several grid cells). In particular, a grid partition in the grid file only splits one interval in two thereby resulting in the insertion of a $(d-1)$-dimensional cross-section. On the other hand, a grid partition in EXCELL means that all intervals must be split in two thereby doubling the size of the grid directory.

An alternative to the array access structure is to assign an ordering to the grid cells resulting from the adaptive grid, and then to impose a tree access structure on the elements of the ordering that correspond to the nonempty grid cells. The ordering is analogous to using a mapping from d dimensions to one dimension. There are many possible orderings (e.g., Chapter 2 in [60]) with the most popular shown in Figure 16.4.

The domain of these mappings is the set of locations of the smallest possible grid cells (termed *pixels*) in the underlying space and thus we need to use some easily identifiable pixel in each grid cell such as the one in the grid cell's lower-left corner. Of course, we

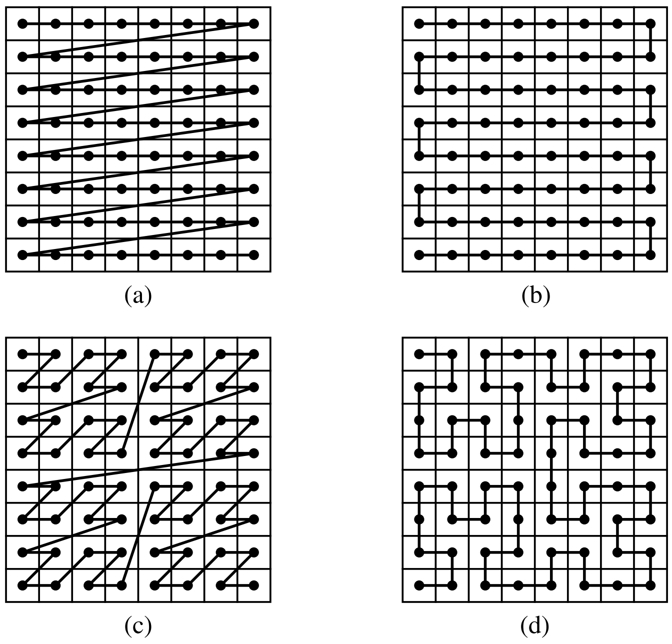


FIGURE 16.4: The result of applying four common different space-ordering methods to an 8×8 collection of pixels whose first element is in the upper-left corner: (a) row order, (b) row-prime order, (c) Morton order, (d) Peano-Hilbert.

also need to know the size of each grid cell. One mapping simply concatenates the result of interleaving the binary representations of the coordinate values of the lower-left corner (e.g., (a, b) in two dimensions) and i of each grid cell of size 2^i so that i is at the right. The resulting number is termed a *locational code* and is a variant of the Morton ordering (Figure 16.4c). Assuming such a mapping and sorting the locational codes in increasing order yields an ordering equivalent to that which would be obtained by traversing the leaf nodes (i.e., grid cells) of the tree representation (e.g., Figure 16.8b) in the order `SW`, `SE`, `NW`, `NE`. The Morton ordering (as well as the Peano-Hilbert ordering shown in Figure 16.4d) is particularly attractive for quadtree-like decompositions because all pixels within a grid cell appear in consecutive positions in the ordering. Alternatively, these two orders exhaust a grid cell before exiting it.

For example, Figure 16.3c shows the result of imposing a B^+-tree [18] access structure on the leaf grid cells of the PR quadtree given in Figure 16.3b. Each node of the B^+-tree in our example has a minimum of 2 and a maximum of 3 entries. Figure 16.3c does not contain the values resulting from applying the mapping to the individual grid cells nor does it show the discriminator values that are stored in the nonleaf nodes of the B^+-tree. The leaf grid cells of the PR quadtree in Figure 16.3a are marked with the label of the leaf node of the B^+-tree of which they are a member (e.g., the grid cell containing `Chicago` is in leaf node `Q` of the B^+-tree).

It is important to observe that the above combination of the PR quadtree and the B^+-tree has the property that the tree structure of the partition process of the underlying space has been decoupled [61] from that of the node hierarchy (i.e., the grouping process of the nodes resulting from the partition process) that makes up the original tree directory. More precisely, the grouping process is based on proximity in the ordering of the locational codes and on the minimum and maximum capacity of the nodes of the B$^+$-tree. Unfortunately, the resulting structure has the property that the space that is spanned by a leaf node of the B$^+$-tree (i.e., the grid cells spanned by it) has an arbitrary shape and, in fact, does not usually correspond to a k-dimensional hyper-rectangle. In particular, the space spanned by the leaf node may have the shape of a staircase (e.g., the leaf grid cells in Figure 16.3a that comprise leaf nodes `S` and `T` of the B$^+$-tree in Figure 16.3c) or may not even be connected in the sense that it corresponds to regions that are not contiguous (e.g., the leaf grid cells in Figure 16.3a that comprise leaf node `R` of the B$^+$-tree in Figure 16.3c). The PK-tree [73] is an alternative decoupling method which overcomes these drawbacks by basing the grouping process on k-instantiation which stipulates that each node of the grouping process contains a minimum of k objects or grid cells. The result is that all of the grid cells of the grouping process are congruent at the cost that the result is not balanced although use of relatively large values of k ensures that the resulting trees are relatively shallow. It can be shown that when the partition process has a fanout of f, then k-instantiation means that the number of objects in each node of the grouping process is bounded by $f \cdot (k-1)$. Note that k-instantiation is different from bucketing where we only have an upper bound on the number of objects in the node.

Fixed-grids, quadtrees, k-d trees, indexkd tree grid file, EXCELL, as well as other hierarchical representations are good for range searching queries such as finding all cities within 80 miles of St. Louis. In particular, they act as pruning devices on the amount of search that will be performed as many points will not be examined since their containing cells lie outside the query range. These representations are generally very easy to implement and have good expected execution times, although they are quite difficult to analyze from a mathematical standpoint. However, their worst cases, despite being rare, can be quite bad. These worst cases can be avoided by making use of variants of range trees [11] and priority search trees [51]. For more details about these data structures, see Chapter 18.

16.3 Bucketing Methods

There are four principal approaches to decomposing the space from which the objects are drawn. The first approach makes use of an object hierarchy and the space decomposition is obtained in an indirect manner as the method propagates the space occupied by the objects up the hierarchy with the identity of the propagated objects being implicit to the hierarchy. In particular, associated with each object is a an object description (e.g., for region data, it is the set of locations in space corresponding to the cells that make up the object). Actually, since this information may be rather voluminous, it is often the case that an approximation of the space occupied by the object is propagated up the hierarchy rather than the collection of individual cells that are spanned by the object. For spatial data, the approximation is usually the minimum bounding rectangle for the object, while for non-spatial data it is simply the hyperrectangle whose sides have lengths equal to the ranges of the values of the attributes. Therefore, associated with each element in the hierarchy is a bounding rectangle corresponding to the union of the bounding rectangles associated with the elements immediately below it.

The R-tree (e.g., [7, 31]) is an example of an object hierarchy which finds use especially in database applications. The number of objects or bounding rectangles that are aggregated in each node is permitted to range between $m \leq \lceil M/2 \rceil$ and M. The root node in an R-tree has at least two entries unless it is a leaf node in which case it has just one entry corresponding to the bounding rectangle of an object. The R-tree is usually built as the objects are encountered rather than waiting until all objects have been input. The hierarchy

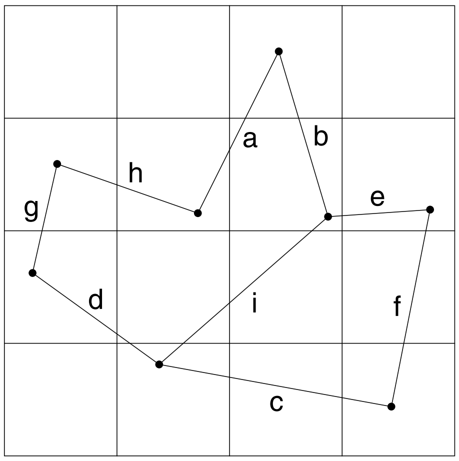


FIGURE 16.5: Example collection of line segments embedded in a 4×4 grid.

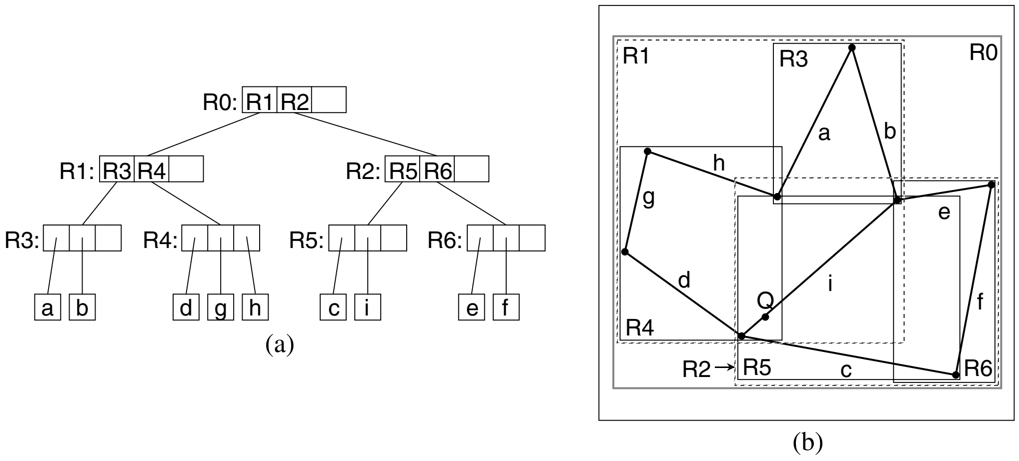


FIGURE 16.6: (a) R-tree for the collection of line segments with m=2 and M=3, in Figure 16.5, and (b) the spatial extents of the bounding rectangles. Notice that the leaf nodes in the index also store bounding rectangles although this is only shown for the nonleaf nodes.

is implemented as a tree structure with grouping being based, in part, on proximity of the objects or bounding rectangles.

For example, consider the collection of line segment objects given in Figure 16.5 shown embedded in a 4×4 grid. Figure 16.6a is an example R-tree for this collection with $m = 2$ and $M = 3$. Figure 16.6b shows the spatial extent of the bounding rectangles of the nodes in Figure 16.6a, with heavy lines denoting the bounding rectangles corresponding to the leaf nodes, and broken lines denoting the bounding rectangles corresponding to the subtrees rooted at the nonleaf nodes. Note that the R-tree is not unique. Its structure depends heavily on the order in which the individual objects were inserted into (and possibly deleted from) the tree.

Given that each R-tree node can contain a varying number of objects or bounding rectangles, it is not surprising that the R-tree was inspired by the B-tree [6]. Therefore, nodes are viewed as analogous to disk pages. Thus the parameters defining the tree (i.e., m and M) are chosen so that a small number of nodes is visited during a spatial query (i.e., point and range queries), which means that m and M are usually quite large. The actual implementation of the R-tree is really a B^+-tree [18] as the objects are restricted to the leaf nodes.

The efficiency of the R-tree for search operations depends on its ability to distinguish between occupied space and unoccupied space (i.e., coverage), and to prevent a node from being examined needlessly due to a false overlap with other nodes. In other words, we want to minimize coverage and overlap. These goals guide the initial R-tree creation process as well, subject to the previously mentioned constraint that the R-tree is usually built as the objects are encountered rather than waiting until all objects have been input.

The drawback of the R-tree (and any representation based on an object hierarchy) is that it does not result in a disjoint decomposition of space. The problem is that an object is only associated with one bounding rectangle (e.g., line segment `i` in Figure 16.6 is associated with bounding rectangle `R5`, yet it passes through `R1`, `R2`, `R4`, and `R5`, as well as through `R0` as do all the line segments). In the worst case, this means that when we wish to determine which object (e.g., an intersecting line in a collection of line segment objects, or a containing rectangle in a collection of rectangle objects) is associated with a particular point in the two-dimensional space from which the objects are drawn, we may have to search the entire collection. For example, in Figure 16.6, when searching for the line segment that passes through point `Q`, we need to examine bounding rectangles `R0`, `R1`, `R4`, `R2`, and `R5`, rather than just `R0`, `R2`, and `R5`.

This drawback can be overcome by using one of three other approaches which are based on a decomposition of space into disjoint cells. Their common property is that the objects are decomposed into disjoint subobjects such that each of the subobjects is associated with a different cell. They differ in the degree of regularity imposed by their underlying decomposition rules, and by the way in which the cells are aggregated into buckets.

The price paid for the disjointness is that in order to determine the area covered by a particular object, we have to retrieve all the cells that it occupies. This price is also paid when we want to delete an object. Fortunately, deletion is not so common in such applications. A related costly consequence of disjointness is that when we wish to determine all the objects that occur in a particular region, we often need to retrieve some of the objects more than once [1, 2, 19]. This is particularly troublesome when the result of the operation serves as input to another operation via composition of functions. For example, suppose we wish to compute the perimeter of all the objects in a given region. Clearly, each object's perimeter should only be computed once. Eliminating the duplicates is a serious issue (see [1] for a discussion of how to deal with this problem for a collection of line segment objects, and [2] for a collection of rectangle objects).

The first method based on disjointness partitions the embedding space into disjoint subspaces, and hence the individual objects into subobjects, so that each subspace consists of disjoint subobjects. The subspaces are then aggregated and grouped in another structure, such as a B-tree, so that all subsequent groupings are disjoint at each level of the structure. The result is termed a k-d-B-tree [59]. The R^+-tree [67, 70] is a modification of the k-d-B-tree where at each level we replace the subspace by the minimum bounding rectangle of the subobjects or subtrees that it contains. The cell tree [30] is based on the same principle as the R^+-tree except that the collections of objects are bounded by minimum convex polyhedra instead of minimum bounding rectangles.

The R^+-tree (as well as the other related representations) is motivated by a desire to avoid overlap among the bounding rectangles. Each object is associated with all the bounding

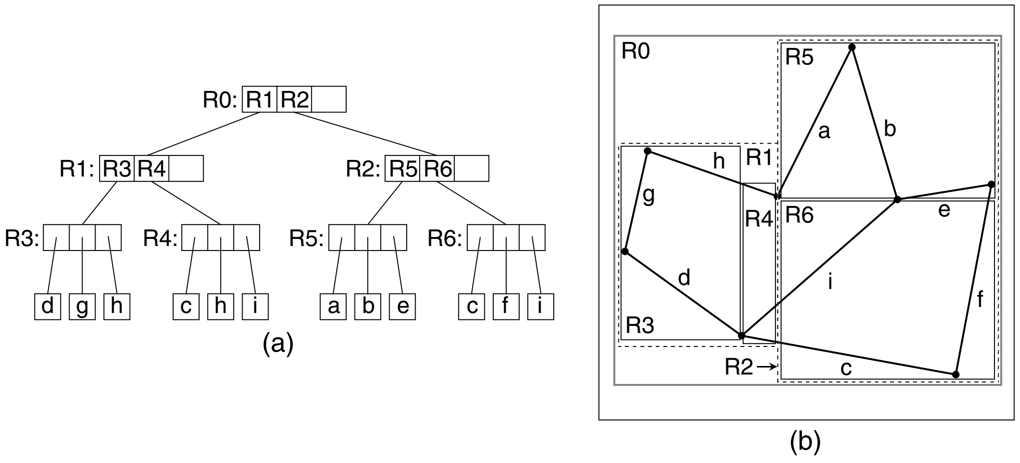


FIGURE 16.7: (a) R$^+$-tree for the collection of line segments in Figure 16.5 with m=2 and M=3, and (b) the spatial extents of the bounding rectangles. Notice that the leaf nodes in the index also store bounding rectangles although this is only shown for the nonleaf nodes.

rectangles that it intersects. All bounding rectangles in the tree (with the exception of the bounding rectangles for the objects at the leaf nodes) are non-overlapping*. The result is that there may be several paths starting at the root to the same object. This may lead to an increase in the height of the tree. However, retrieval time is sped up.

Figure 16.7 is an example of one possible R$^+$-tree for the collection of line segments in Figure 16.5. This particular tree is of order (2,3) although in general it is not possible to guarantee that all nodes will always have a minimum of 2 entries. In particular, the expected B-tree performance guarantees are not valid (i.e., pages are not guaranteed to be m/M full) unless we are willing to perform very complicated record insertion and deletion procedures. Notice that line segment objects `c`, `h`, and `i` appear in two different nodes. Of course, other variants are possible since the R$^+$-tree is not unique.

The problem with representations such as the k-d-B-tree and the R$^+$-tree is that overflow in a leaf node may cause overflow of nodes at shallower depths in the tree whose subsequent partitioning may cause repartitioning at deeper levels in the tree. There are several ways of overcoming the repartitioning problem. One approach is to use the LSD-tree [32] at the cost of poorer storage utilization. An alternative approach is to use representations such as the hB-tree [49] and the BANG file [27] which remove the requirement that each block be a hyper-rectangle at the cost of multiple postings. This has a similar effect as that obtained when decomposing an object into several subobjects in order to overcome the nondisjoint decomposition problem when using an object hierarchy. The multiple posting problem is overcome by the BV-tree [28] which decouples the partitioning and grouping processes at the cost that the resulting tree is no longer balanced although as in the PK-tree [73] (which we point out in Section 16.2 is also based on decoupling), use of relatively large fanout

*From a theoretical viewpoint, the bounding rectangles for the objects at the leaf nodes should also be disjoint However, this may be impossible (e.g., when the objects are line segments and if many of the line segments intersect at a point).

values ensure that the resulting trees are relatively shallow.

Methods such as the R$^+$-tree (as well as the R-tree) also have the drawback that the decomposition is data-dependent. This means that it is difficult to perform tasks that require composition of different operations and data sets (e.g., set-theoretic operations such as overlay). The problem is that although these methods are good are distinguishing between occupied and unoccupied space in the underlying space (termed *image* in much of the subsequent discussion) under consideration, they re unable to correlate occupied space in two distinct images, and likewise for unoccupied space in the two images.

In contrast, the remaining two approaches to the decomposition of space into disjoint cells have a greater degree of data-independence. They are based on a regular decomposition. The space can be decomposed either into blocks of uniform size (e.g., the uniform grid [24]) or adapt the decomposition to the distribution of the data (e.g., a quadtree-based approach such as [66]). In the former case, all the blocks are congruent (e.g., the 4×4 grid in Figure 16.5). In the latter case, the widths of the blocks are restricted to be powers of two and their positions are also restricted. Since the positions of the subdivision lines are restricted, and essentially the same for all images of the same size, it is easy to correlate occupied and unoccupied space in different images.

The uniform grid is ideal for uniformly-distributed data, while quadtree-based approaches are suited for arbitrarily-distributed data. In the case of uniformly-distributed data, quadtree-based approaches degenerate to a uniform grid, albeit they have a higher overhead. Both the uniform grid and the quadtree-based approaches lend themselves to set-theoretic operations and thus they are ideal for tasks which require the composition of different operations and data sets. In general, since spatial data is not usually uniformly distributed, the quadtree-based regular decomposition approach is more flexible. The drawback of quadtree-like methods is their sensitivity to positioning in the sense that the placement of the objects relative to the decomposition lines of the space in which they are embedded effects their storage costs and the amount of decomposition that takes place. This is overcome to a large extent by using a bucketing adaptation that decomposes a block only if it contains more than b objects.

16.4 Region Data

There are many ways of representing region data. We can represent a region either by its boundary (termed a *boundary-based* representation) or by its interior (termed an *interior-based* representation). In this section, we focus on representations of collections of regions by their interior. In some applications, regions are really objects that are composed of smaller primitive objects by use of geometric transformations and Boolean set operations. *Constructive Solid Geometry (CSG)* [58] is a term usually used to describe such representations. They are beyond the scope of this chapter. Instead, unless noted otherwise, our discussion is restricted to regions consisting of congruent cells of unit area (volume) with sides (faces) of unit size that are orthogonal to the coordinate axes.

Regions with arbitrary boundaries are usually represented by either using approximating bounding rectangles or more general boundary-based representations that are applicable to collections of line segments that do not necessarily form regions. In that case, we do not restrict the line segments to be perpendicular to the coordinate axes. Such representations are discussed in Section 16.6. It should be clear that although our presentation and examples in this section deal primarily with two-dimensional data, they are valid for regions of any dimensionality.

The region data is assumed to be uniform in the sense that all the cells that comprise each region are of the same type. In other words, each region is homogeneous. Of course, an image may consist of several distinct regions. Perhaps the best definition of a region is as a set of four-connected cells (i.e., in two dimensions, the cells are adjacent along an edge rather than a vertex) each of which is of the same type. For example, we may have a crop map where the regions correspond to the four-connected cells on which the same crop is grown. Each region is represented by the collection of cells that comprise it. The set of collections of cells that make up all of the regions is often termed an *image array* because of the nature in which they are accessed when performing operations on them. In particular, the array serves as an access structure in determining the region associated with a location of a cell as well as all remaining cells that comprise the region.

When the region is represented by its interior, then often we can reduce the storage requirements by aggregating identically-valued cells into blocks. In the rest of this section we discuss different methods of aggregating the cells that comprise each region into blocks as well as the methods used to represent the collections of blocks that comprise each region in the image.

The collection of blocks is usually a result of a space decomposition process with a set of rules that guide it. There are many possible decompositions. When the decomposition is recursive, we have the situation that the decomposition occurs in stages and often, although not always, the results of the stages form a containment hierarchy. This means that a block b obtained in stage i is decomposed into a set of blocks b_j that span the same space. Blocks b_j are, in turn, decomposed in stage $i+1$ using the same decomposition rule. Some decomposition rules restrict the possible sizes and shapes of the blocks as well as their placement in space. Some examples include:

- congruent blocks at each stage
- similar blocks at all stages
- all sides of a block are of equal size
- all sides of each block are powers of two
- etc.

Other decomposition rules dispense with the requirement that the blocks be rectangular (i.e., there exist decompositions using other shapes such as triangles, etc.), while still others do not require that they be orthogonal, although, as stated before, we do make these assumptions here. In addition, the blocks may be disjoint or be allowed to overlap. Clearly, the choice is large. In the following, we briefly explore some of these decomposition processes. We restrict ourselves to disjoint decompositions, although this need not be the case (e.g., the field tree [23]).

The most general decomposition permits aggregation along all dimensions. In other words, the decomposition is arbitrary. The blocks need not be uniform or similar. The only requirement is that the blocks span the space of the environment. The drawback of arbitrary decompositions is that there is little structure associated with them. This means that it is difficult to answer queries such as determining the region associated with a given point, besides exhaustive search through the blocks. Thus we need an additional data structure known as an index or an access structure. A very simple decomposition rule that lends itself to such an index in the form of an array is one that partitions a d-dimensional space having coordinate axes x_i into d-dimensional blocks by use of h_i hyperplanes that are parallel to the hyperplane formed by $x_i = 0$ $(1 \leq i \leq d)$. The result is a collection of $\prod_{i=1}^{d}(h_i + 1)$ blocks. These blocks form a grid of irregular-sized blocks rather than congruent blocks. There is no recursion involved in the decomposition process. We term the resulting decomposition an *irregular grid* as the partition lines are at arbitrary positions in contrast to a *uniform* *grid* [24] where the partition lines are positioned so that all of the resulting grid cells are congruent.

Although the blocks in the irregular grid are not congruent, we can still impose an array access structure by adding d access structures termed *linear scales*. The linear scales indicate the position of the partitioning hyperplanes that are parallel to the hyperplane formed by $x_i = 0$ $(1 \leq i \leq d)$. Thus given a location l in space, say (a,b) in two-dimensional space, the linear scales for the x and y coordinate values indicate the column and row, respectively, of the array access structure entry which corresponds to the block that contains l. The linear scales are usually represented as one-dimensional arrays although they can be implemented using tree access structures such as binary search trees, range trees, segment trees, etc.

Perhaps the most widely known decompositions into blocks are those referred to by the general terms *quadtree* and *octree* [60, 63]. They are usually used to describe a class of representations for two and three-dimensional data (and higher as well), respectively, that are the result of a recursive decomposition of the environment (i.e., space) containing the regions into blocks (not necessarily rectangular) until the data in each block satisfies some condition (e.g., with respect to its size, the nature of the regions that comprise it, the number of regions in it, etc.). The positions and/or sizes of the blocks may be restricted or arbitrary. It is interesting to note that quadtrees and octrees may be used with both interior-based and boundary-based representations although only the former are discussed in this section.

There are many variants of quadtrees and octrees (see also Sections 16.2, 16.5, and 16.6), and they are used in numerous application areas including high energy physics, VLSI, finite element analysis, and many others. Below, we focus on *region quadtrees* [43] and to a lesser extent on *region octrees* [39, 53] They are specific examples of interior-based representations for two and three-dimensional region data (variants for data of higher dimension also exist), respectively, that permit further aggregation of identically-valued cells.

Region quadtrees and region octrees are instances of a restricted-decomposition rule where the environment containing the regions is recursively decomposed into four or eight, respectively, rectangular congruent blocks until each block is either completely occupied by a region or is empty (such a decomposition process is termed *regular*). For example, Figure 16.8a is the block decomposition for the region quadtree corresponding to three regions `A`, `B`, and `C`. Notice that in this case, all the blocks are square, have sides whose size is a power of 2, and are located at specific positions. In particular, assuming an origin at the lower-left corner of the image containing the regions, then the coordinate values of the lower-left corner of each block (e.g., (a, b) in two dimensions) of size $2^i \times 2^i$ satisfy the property that $a \bmod 2^i = 0$ and $b \bmod 2^i = 0$.

The traditional, and most natural, access structure for a region quadtree corresponding to a d-dimensional image is a tree with a fanout of 2^d (e.g., Figure 16.8b). Each leaf node in the tree corresponds to a different block b and contains the identity of the region associated with b. Each nonleaf node f corresponds to a block whose volume is the union of the blocks corresponding to the 2^d sons of f. In this case, the tree is a containment hierarchy and closely parallels the decomposition in the sense that they are both recursive processes and the blocks corresponding to nodes at different depths of the tree are similar in shape. Of course, the region quadtree could also be represented by using a mapping from the domain of the blocks to a subset of the integers and then imposing a tree access structure such as a B$^+$-tree on the result of the mapping as was described in Section 16.2 for point data stored in a PR quadtree.

As the dimensionality of the space (i.e., d) increases, each level of decomposition in the region quadtree results in many new blocks as the fanout value 2^d is high. In particular, it is too large for a practical implementation of the tree access structure. In this case, an

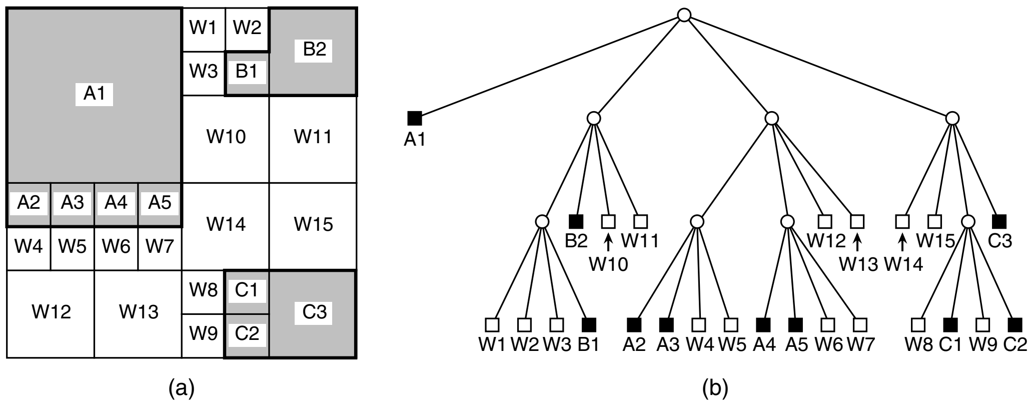


FIGURE 16.8: (a) Block decomposition and (b) its tree representation for the region quadtree corresponding to a collection of three regions A, B, and C.

access structure termed a *bintree* [44, 65, 72] with a fanout value of 2 is used. The bintree is defined in a manner analogous to the region quadtree except that at each subdivision stage, the space is decomposed into two equal-sized parts. In two dimensions, at odd stages we partition along the y axis and at even stages we partition along the x axis. In general, in the case of d dimensions, we cycle through the different axes every d levels in the bintree.

The region quadtree, as well as the bintree, is a regular decomposition. This means that the blocks are congruent — that is, at each level of decomposition, all of the resulting blocks are of the same shape and size. We can also use decompositions where the sizes of the blocks are not restricted in the sense that the only restriction is that they be rectangular and be a result of a recursive decomposition process. In this case, the representations that we described must be modified so that the sizes of the individual blocks can be obtained. An example of such a structure is an adaptation of the point quadtree [22] to regions. Although the point quadtree was designed to represent points in a higher dimensional space, the blocks resulting from its use to decompose space do correspond to regions. The difference from the region quadtree is that in the point quadtree, the positions of the partitions are arbitrary, whereas they are a result of a partitioning process into 2^d congruent blocks (e.g., quartering in two dimensions) in the case of the region quadtree.

As in the case of the region quadtree, as the dimensionality d of the space increases, each level of decomposition in the point quadtree results in many new blocks since the fanout value 2^d is high. In particular, it is too large for a practical implementation of the tree access structure. In this case, we can adapt the k-d tree [8], which has a fanout value of 2, to regions. As in the point quadtree, although the k-d tree was designed to represent points in a higher dimensional space, the blocks resulting from its use to decompose space do correspond to regions. Thus the relationship of the k-d tree to the point quadtree is the same as the relationship of the bintree to the region quadtree. In fact, the k-d tree is the precursor of the bintree and its adaptation to regions is defined in a similar manner in the sense that for d-dimensional data we cycle through the d axes every d levels in the k-d tree. The difference is that in the k-d tree, the positions of the partitions are arbitrary, whereas they are a result of a halving process in the case of the bintree.

The k-d tree can be further generalized so that the partitions take place on the various axes at an arbitrary order, and, in fact, the partitions need not be made on every coordinate axis. The k-d tree is a special case of the *BSP tree* (denoting *Binary Space Partitioning*) [29] where the partitioning hyperplanes are restricted to be parallel to the axes, whereas in the

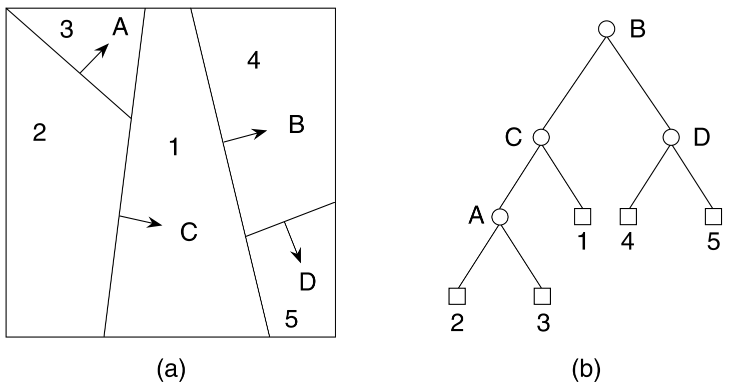


FIGURE 16.9: (a) An arbitrary space decomposition and (b) its BSP tree. The arrows indicate the direction of the positive halfspaces.

BSP tree they have an arbitrary orientation. The BSP tree is a binary tree. In order to be able to assign regions to the left and right subtrees, we need to associate a direction with each subdivision line. In particular, the subdivision lines are treated as separators between two halfspaces[†]. Let the subdivision line have the equation $a \cdot x + b \cdot y + c = 0$. We say that the right subtree is the 'positive' side and contains all subdivision lines formed by separators that satisfy $a \cdot x + b \cdot y + c \geq 0$. Similarly, we say that the left subtree is 'negative' and contains all subdivision lines formed by separators that satisfy $a \cdot x + b \cdot y + c < 0$. As an example, consider Figure 16.9a which is an arbitrary space decomposition whose BSP tree is given in Figure 16.9b. Notice the use of arrows to indicate the direction of the positive halfspaces. The BSP tree is used in computer graphics to facilitate viewing. It is discussed in greater detail in Chapter 20.

As mentioned before, the various hierarchical data structures that we described can also be used to represent regions in three dimensions and higher. As an example, we briefly describe the region octree which is the three-dimensional analog of the region quadtree. It is constructed in the following manner. We start with an image in the form of a cubical volume and recursively subdivide it into eight congruent disjoint cubes (called octants) until blocks are obtained of a uniform color or a predetermined level of decomposition is reached. Figure 16.10a is an example of a simple three-dimensional object whose region octree block decomposition is given in Figure 16.10b and whose tree representation is given in Figure 16.10c.

The aggregation of cells into blocks in region quadtrees and region octrees is motivated, in part, by a desire to save space. Some of the decompositions have quite a bit of structure thereby leading to inflexibility in choosing partition lines, etc. In fact, at times, maintaining the original image with an array access structure may be more effective from the standpoint of storage requirements. In the following, we point out some important implications of the use of these aggregations. In particular, we focus on the region quadtree and region octree. Similar results could also be obtained for the remaining block decompositions.

[†]A (linear) *halfspace* in d-dimensional space is defined by the inequality $\sum_{i=0}^{d} a_i \cdot x_i \geq 0$ on the $d+1$ homogeneous coordinates ($x_0 = 1$). The halfspace is represented by a column vector a. In vector notation, the inequality is written as $a \cdot x \geq 0$. In the case of equality, it defines a hyperplane with a as its normal. It is important to note that halfspaces are volume elements; they are not boundary elements.

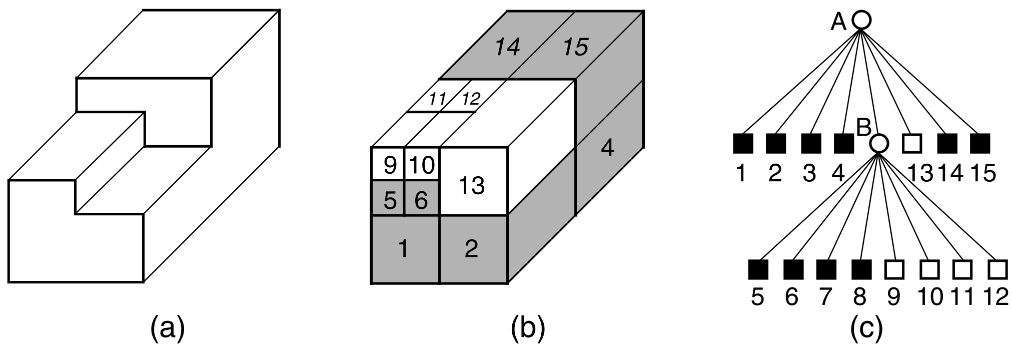


FIGURE 16.10: (a) Example three-dimensional object; (b) its region octree block decomposition; and (c) its tree representation.

The aggregation of similarly-valued cells into blocks has an important effect on the execution time of the algorithms that make use of the region quadtree. In particular, most algorithms that operate on images represented by a region quadtree are implemented by a preorder traversal of the quadtree and, thus, their execution time is generally a linear function of the number of nodes in the quadtree. A key to the analysis of the execution time of quadtree algorithms is the *Quadtree Complexity Theorem* [39] which states that the number of nodes in a region quadtree representation for a simple polygon (i.e., with non-intersecting edges and without holes) is $O(p+q)$ for a $2^q \times 2^q$ image with perimeter p measured in terms of the width of unit-sized cells (i.e., pixels). In all but the most pathological cases (e.g., a small square of unit width centered in a large image), the q factor is negligible and thus the number of nodes is $O(p)$.

The Quadtree Complexity Theorem also holds for three-dimensional data [52] (i.e., represented by a region octree) where perimeter is replaced by surface area, as well as for objects of higher dimensions d for which it is proportional to the size of the $(d-1)$-dimensional interfaces between these objects. The most important consequence of the Quadtree Complexity Theorem is that it means that most algorithms that execute on a region quadtree representation of an image, instead of one that simply imposes an array access structure on the original collection of cells, usually have an execution time that is proportional to the number of blocks in the image rather than the number of unit-sized cells. In its most general case, this means that the use of the region quadtree, with an appropriate access structure, in solving a problem in d-dimensional space will lead to a solution whose execution time is proportional to the $(d-1)$-dimensional space of the surface of the original d-dimensional image. On the other hand, use of the array access structure on the original collection of cells results in a solution whose execution time is proportional to the number of cells that comprise the image. Therefore, region quadtrees and region octrees act like dimension-reducing devices.

16.5 Rectangle Data

The rectangle data type lies somewhere between the point and region data types. It can also be viewed as a special case of the region data type in the sense that it is a region with only four sides. Rectangles are often used to approximate other objects in an image for which they serve as the minimum rectilinear enclosing object. For example, bounding rectangles are used in cartographic applications to approximate objects such as lakes, forests, hills, etc. In such a case, the approximation gives an indication of the existence of an object. Of course, the exact boundaries of the object are also stored; but they are only accessed if greater precision is needed. For such applications, the number of elements in the collection is usually small, and most often the sizes of the rectangles are of the same order of magnitude as the space from which they are drawn.

Rectangles are also used in VLSI design rule checking as a model of chip components for the analysis of their proper placement. Again, the rectangles serve as minimum enclosing objects. In this application, the size of the collection is quite large (e.g., millions of components) and the sizes of the rectangles are several orders of magnitude smaller than the space from which they are drawn.

It should be clear that the actual representation that is used depends heavily on the problem environment. At times, the rectangle is treated as the Cartesian product of two one-dimensional intervals with the horizontal intervals being treated in a different manner than the vertical intervals. In fact, the representation issue is often reduced to one of representing intervals. For example, this is the case in the use of the plane-sweep paradigm [57] in the solution of rectangle problems such as determining all pairs of intersecting rectangles. In this case, each interval is represented by its left and right endpoints. The solution makes use of two passes.

The first pass sorts the rectangles in ascending order on the basis of their left and right sides (i.e., x coordinate values) and forms a list. The second pass sweeps a vertical scan line through the sorted list from left to right halting at each one of these points, say p. At any instant, all rectangles that intersect the scan line are considered *active* and are the only ones whose intersection needs to be checked with the rectangle associated with p. This means that each time the sweep line halts, a rectangle either becomes active (causing it to be inserted in the set of active rectangles) or ceases to be active (causing it to be deleted from the set of active rectangles). Thus the key to the algorithm is its ability to keep track of the active rectangles (actually just their vertical sides) as well as to perform the actual one-dimensional intersection test.

Data structures such as the segment tree [9], interval tree [20], and the priority search tree [51] can be used to organize the vertical sides of the active rectangles so that, for N rectangles and F intersecting pairs of rectangles, the problem can be solved in $O(N \cdot \log_2 N + F)$ time. All three data structures enable intersection detection, insertion, and deletion to be executed in $O(\log_2 N)$ time. The difference between them is that the segment tree requires $O(N \cdot \log_2 N)$ space while the interval tree and the priority search tree only need $O(N)$ space. These algorithms require that the set of rectangles be known in advance. However, they work even when the size of the set of active rectangles exceeds the amount of available memory, in which case multiple passes are made over the data [41]. For more details about these data structures, see Chapter 18.

In this chapter, we are primarily interested in dynamic problems (i.e., the set of rectangles is constantly changing). The data structures that are chosen for the collection of the rectangles are differentiated by the way in which each rectangle is represented. One representation discussed in Section 16.1 reduces each rectangle to a point in a higher dimensional space, and then treats the problem as if we have a collection of points [33]. Again, each rectangle is a Cartesian product of two one-dimensional intervals where the difference from its use with the plane-sweep paradigm is that each interval is represented by its centroid and extent. Each set of intervals in a particular dimension is, in turn, represented by a grid file [55] which is described in Section 16.2.

The second representation is region-based in the sense that the subdivision of the space from which the rectangles are drawn depends on the physical extent of the rectangle — not just one point. Representing the collection of rectangles, in turn, with a tree-like data

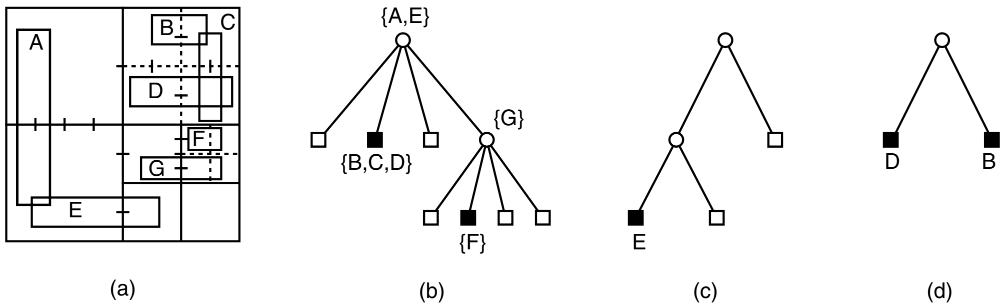


FIGURE 16.11: (a) Collection of rectangles and the block decomposition induced by the MX-CIF quadtree; (b) the tree representation of (a); the binary trees for the y axes passing through the root of the tree in (b), and (d) the NE son of the root of the tree in (b).

structure has the advantage that there is a relation between the depth of node in the tree and the size of the rectangle(s) that is (are) associated with it. Interestingly, some of the region-based solutions make use of the same data structures that are used in the solutions based on the plane-sweep paradigm.

There are three types of region-based solutions currently in use. The first two solutions use the R-tree and the R$^+$-tree (discussed in Section 16.3) to store rectangle data (in this case the objects are rectangles instead of arbitrary objects). The third is a quadtree-based approach and uses the MX-CIF quadtree [42] (see also [47] for a related variant).

In the *MX-CIF quadtree*, each rectangle is associated with the quadtree node corresponding to the smallest block which contains it in its entirety. Subdivision ceases whenever a node's block contains no rectangles. Alternatively, subdivision can also cease once a quadtree block is smaller than a predetermined threshold size. This threshold is often chosen to be equal to the expected size of the rectangle [42]. For example, Figure 16.11b is the MX-CIF quadtree for a collection of rectangles given in Figure 16.11a. Rectangles can be associated with both leaf and nonleaf nodes.

It should be clear that more than one rectangle can be associated with a given enclosing block and, thus, often we find it useful to be able to differentiate between them. This is done in the following manner [42]. Let P be a quadtree node with centroid (CX,CY), and let S be the set of rectangles that are associated with P. Members of S are organized into two sets according to their intersection (or collinearity of their sides) with the lines passing through the centroid of P's block — that is, all members of S that intersect the line $x = CX$ form one set and all members of S that intersect the line $y = CY$ form the other set.

If a rectangle intersects both lines (i.e., it contains the centroid of P's block), then we adopt the convention that it is stored with the set associated with the line through $x = CX$. These subsets are implemented as binary trees (really tries), which in actuality are one-dimensional analogs of the MX-CIF quadtree. For example, Figure 16.11c and Figure 16.11d illustrate the binary trees associated with the y axes passing through the root and the `NE` son of the root, respectively, of the MX-CIF quadtree of Figure 16.11b. Interestingly, the MX-CIF quadtree is a two-dimensional analog of the interval tree. described above. More precisely, the MX-CIF quadtree is a a two-dimensional analog of the tile tree [50] which is a regular decomposition version of the interval tree. In fact, the tile tree and the one-dimensional MX-CIF quadtree are identical when rectangles are not allowed to overlap.

16.6 Line Data and Boundaries of Regions

Section 16.4 was devoted to variations on hierarchical decompositions of regions into blocks, an approach to region representation that is based on a description of the region's interior. In this section, we focus on representations that enable the specification of the boundaries of regions, as well as curvilinear data and collections of line segments. The representations are usually based on a series of approximations which provide successively closer fits to the data, often with the aid of bounding rectangles. When the boundaries or line segments have a constant slope (i.e., linear and termed *line segments* in the rest of this discussion), then an exact representation is possible.

There are several ways of approximating a curvilinear line segment. The first is by digitizing it and then marking the unit-sized cells (i.e., pixels) through which it passes. The second is to approximate it by a set of straight line segments termed a *polyline*. Assuming a boundary consisting of straight lines (or polylines after the first stage of approximation), the simplest representation of the boundary of a region is the polygon. It consists of vectors which are usually specified in the form of lists of pairs of x and y coordinate values corresponding to their start and end points. The vectors are usually ordered according to their connectivity. One of the most common representations is the chain code [26] which is an approximation of a polygon's boundary by use of a sequence of unit vectors in the four (and sometimes eight) principal directions.

Chain codes, and other polygon representations, break down for data in three dimensions and higher. This is primarily due to the difficulty in ordering their boundaries by connectivity. The problem is that in two dimensions connectivity is determined by ordering the boundary elements $e_{i,j}$ of boundary b_i of object o so that the end vertex of the vector v_j corresponding to $e_{i,j}$ is the start vertex of the vector v_{j+1} corresponding to $e_{i,j+1}$. Unfortunately, such an implicit ordering does not exist in higher dimensions as the relationship between the boundary elements associated with a particular object are more complex.

Instead, we must make use of data structures which capture the topology of the object in terms of its faces, edges, and vertices. The winged-edge data structure is one such representation which serves as the basis of the boundary model (also known as *BRep* [5]). For more details about these data structures, see Chapter 17.

Polygon representations are very local. In particular, if we are at one position on the boundary, we don't know anything about the rest of the boundary without traversing it element-by-element. Thus, using such representations, given a random point in space, it is very difficult to find the nearest line to it as the lines are not sorted. This is in contrast to hierarchical representations which are global in nature. They are primarily based on rectangular approximations to the data as well as on a regular decomposition in two dimensions. In the rest of this section, we discuss a number of such representations.

In Section 16.3 we already examined two hierarchical representations (i.e., the R-tree and the R$^+$-tree) that propagate object approximations in the form of bounding rectangles. In this case, the sides of the bounding rectangles had to be parallel to the coordinate axes of the space from which the objects are drawn. In contrast, the *strip tree* [4] is a hierarchical representation of a single curve that successively approximates segments of it with bounding rectangles that does not require that the sides be parallel to the coordinate axes. The only requirement is that the curve be continuous; it need not be differentiable.

The strip tree data structure consists of a binary tree whose root represents the bounding rectangle of the entire curve. The rectangle associated with the root corresponds to a rectangular strip, that encloses the curve, whose sides are parallel to the line joining the endpoints of the curve. The curve is then partitioned in two at one of the locations where it touches the bounding rectangle (these are not tangent points as the curve only needs to be

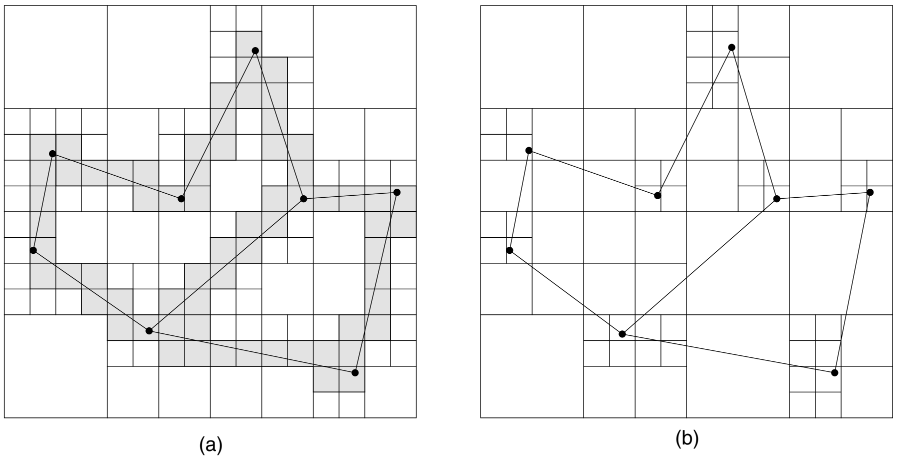


FIGURE 16.12: (a) MX quadtree and (b) edge quadtree for the collection of line segments of Figure 16.5.

continuous; it need not be differentiable). Each subcurve is then surrounded by a bounding rectangle and the partitioning process is applied recursively. This process stops when the width of each strip is less than a predetermined value.

In order to be able to cope with more complex curves such as those that arise in the case of object boundaries, the notion of a strip tree must be extended. In particular, closed curves and curves that extend past their endpoints require some special treatment. The general idea is that these curves are enclosed by rectangles which are split into two rectangular strips, and from now on the strip tree is used as before.

The strip tree is similar to the point quadtree in the sense that the points at which the curve is decomposed depend on the data. In contrast, a representation based on the region quadtree has fixed decomposition points. Similarly, strip tree methods approximate curvilinear data with rectangles of arbitrary orientation, while methods based on the region quadtree achieve analogous results by use of a collection of disjoint squares having sides of length power of two. In the following we discuss a number of adaptations of the region quadtree for representing curvilinear data.

The simplest adaptation of the region quadtree is the MX quadtree [39, 40]. It is built by digitizing the line segments and labeling each unit-sized cell (i.e., pixel) through which it passes as of type `boundary`. The remaining pixels are marked `WHITE` and are merged, if possible, into larger and larger quadtree blocks. Figure 16.12a is the MX quadtree for the collection of line segment objects in Figure 16.5. A drawback of the MX quadtree is that it associates a thickness with a line. Also, it is difficult to detect the presence of a vertex whenever five or more line segments meet.

The edge quadtree [68, 74] is a refinement of the MX quadtree based on the observation that the number of squares in the decomposition can be reduced by terminating the subdivision whenever the square contains a single curve that can be approximated by a single straight line. For example, Figure 16.12b is the edge quadtree for the collection of line segment objects in Figure 16.5. Applying this process leads to quadtrees in which long edges are represented by large blocks or a sequence of large blocks. However, small blocks are required in the vicinity of corners or intersecting edges. Of course, many blocks will contain no edge information at all.

The PM quadtree family [54, 66] (see also edge-EXCELL [71]) represents an attempt

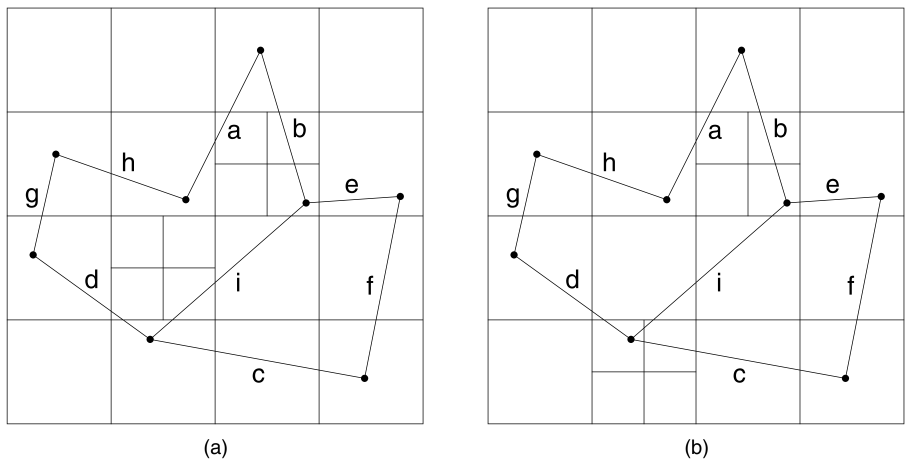


FIGURE 16.13: (a) PM$_1$ quadtree and (b) PMR quadtree for the collection of line segments of Figure 16.5.

to overcome some of the problems associated with the edge quadtree in the representation of collections of polygons (termed *polygonal maps*). In particular, the edge quadtree is an approximation because vertices are represented by pixels. There are a number of variants of the PM quadtree. These variants are either vertex-based or edge-based. They are all built by applying the principle of repeatedly breaking up the collection of vertices and edges (forming the polygonal map) until obtaining a subset that is sufficiently simple so that it can be organized by some other data structure.

The PM$_1$ quadtree [66] is an example of a vertex-based PM quadtree. Its decomposition rule stipulates that partitioning occurs as long as a block contains more than one line segment unless the line segments are all incident at the same vertex which is also in the same block (e.g., Figure 16.13a). Given a polygonal map whose vertices are drawn from a grid (say $2^m \times 2^m$), and where edges are not permitted to intersect at points other than the grid points (i.e., vertices), it can be shown that the maximum depth of any leaf node in the PM$_1$ quadtree is bounded from above by $4m + 1$ [64]. This enables a determination of the maximum amount of storage that will be necessary for each node.

A similar representation has been devised for three-dimensional images (e.g., [3] and the references cited in [63]). The decomposition criteria are such that no node contains more than one face, edge, or vertex unless the faces all meet at the same vertex or are adjacent to the same edge. This representation is quite useful since its space requirements for polyhedral objects are significantly smaller than those of a region octree.

The PMR quadtree [54] is an edge-based variant of the PM quadtree. It makes use of a probabilistic splitting rule. A node is permitted to contain a variable number of line segments. A line segment is stored in a PMR quadtree by inserting it into the nodes corresponding to all the blocks that it intersects. During this process, the occupancy of each node that is intersected by the line segment is checked to see if the insertion causes it to exceed a predetermined *splitting threshold*. If the splitting threshold is exceeded, then the node's block is split *once*, and only once, into four equal quadrants.

For example, Figure 16.13b is the PMR quadtree for the collection of line segment objects in Figure 16.5 with a splitting threshold value of 2. The line segments are inserted in alphabetic order (i.e., `a`–`i`). It should be clear that the shape of the PMR quadtree depends on the order in which the line segments are inserted. Note the difference from the PM$_1$

quadtree in Figure 16.13a – that is, the `NE` block of the `SW` quadrant is decomposed in the PM_1 quadtree while the `SE` block of the `SW` quadrant is not decomposed in the PM_1 quadtree.

On the other hand, a line segment is deleted from a PMR quadtree by removing it from the nodes corresponding to all the blocks that it intersects. During this process, the occupancy of the node and its siblings is checked to see if the deletion causes the total number of line segments in them to be less than the predetermined splitting threshold. If the splitting threshold exceeds the occupancy of the node and its siblings, then they are merged and the merging process is reapplied to the resulting node and its siblings. Notice the asymmetry between the splitting and merging rules.

The PMR quadtree is very good for answering queries such as finding the nearest line to a given point [34–37] (see [38] for an empirical comparison with hierarchical object representations such as the R-tree and R^+-tree). It is preferred over the PM_1 quadtree (as well as the MX and edge quadtrees) as it results in far fewer subdivisions. In particular, in the PMR quadtree there is no need to subdivide in order to separate line segments that are very "close" or whose vertices are very "close," which is the case for the PM_1 quadtree. This is important since four blocks are created at each subdivision step. Thus when many subdivision steps that occur in the PM_1 quadtree result in creating many empty blocks, the storage requirements of the PM_1 quadtree will be considerably higher than those of the PMR quadtree. Generally, as the splitting threshold is increased, the storage requirements of the PMR quadtree decrease while the time necessary to perform operations on it will increase.

Using a random image model and geometric probability, it has been shown [48], theoretically and empirically using both random and real map data, that for sufficiently high values of the splitting threshold (i.e., ≥ 4), the number of nodes in a PMR quadtree is asymptotically proportional to the number of line segments and is independent of the maximum depth of the tree. In contrast, using the same model, the number of nodes in the PM_1 quadtree is a product of the number of lines and the maximal depth of the tree (i.e., n for a $2^n \times 2^n$ image). The same experiments and analysis for the MX quadtree confirmed the results predicted by the Quadtree Complexity Theorem (see Section 16.4) which is that the number of nodes is proportional to the total length of the line segments.

Observe that although a bucket in the PMR quadtree can contain more line segments than the splitting threshold, this is not a problem. In fact, it can be shown [63] that the maximum number of line segments in a bucket is bounded by the sum of the splitting threshold and the depth of the block (i.e., the number of times the original space has been decomposed to yield this block).

16.7 Research Issues and Summary

A review has been presented of a number of representations of multidimensional data. Our focus has been on multidimensional spatial data with extent rather than just multidimensional point data. There has been a particular emphasis on hierarchical representations. Such representations are based on the "divide-and-conquer" problem-solving paradigm. They are of interest because they enable focusing computational resources on the interesting subsets of data. Thus, there is no need to expend work where the payoff is small. Although many of the operations for which they are used can often be performed equally as efficiently, or more so, with other data structures, hierarchical data structures are attractive because of their conceptual clarity and ease of implementation.

When the hierarchical data structures are based on the principle of regular decomposition, we have the added benefit that different data sets (often of differing types) are in registration. This means that they are partitioned in known positions which are often the same or subsets of one another for the different data sets. This is true for all the features including regions, points, rectangles, lines, volumes, etc. The result is that a query such as "finding all cities with more than 20,000 inhabitants in wheat growing regions within 30 miles of the Mississippi River" can be executed by simply overlaying the region (crops), point (i.e., cities), and river maps even though they represent data of different types. Alternatively, we may extract regions such as those within 30 miles of the Mississippi River. Such operations find use in applications involving spatial data such as geographic information systems.

Current research in multidimensional representations is highly application-dependent in the sense that the work is driven by the application. Many of the recent developments have been motivated by the interaction with databases. The choice of a proper representation plays a key role in the speed with which responses are provided to queries. Knowledge of the underlying data distribution is also a factor and research is ongoing to make use of this information in the process of making a choice. Most of the initial applications in which the representation of multidimensional data has been important have involved spatial data of the kind described in this chapter. Such data is intrinsically of low dimensionality (i.e., two and three).

Future applications involve higher dimensional data for applications such as image databases where the data are often points in feature space. Unfortunately, for such applications, the performance of most indexing methods that rely on a decomposition of the underlying space is often unsatisfactory when compared with not using an index at all (e.g., [16]). The problem is that for uniformly-distributed data, most of the data is found to be at or near the boundary of the space in which it lies [13]. The result means that the query region usually overlaps all of the leaf node regions that are created by the decomposition process and thus a sequential scan is preferable. This has led to a number of alternative representations that try to speed up the scan (e.g., VA-file [75], VA$^+$-file [21], IQ-tree [15], etc.). Nevertheless, representations such as the pyramid technique [14] are based on the principle that most of the data lies near the surface and therefore subdivide the data space as if it is an onion by peeling off hypervolumes that are close to its boundary. This is achieved by first dividing the hypercube corresponding to the d-dimensional data space into $2d$ pyramids having the center of the data space as their top point and one of the faces of the hypercube as its base. These pyramids are subsequently cut into slices that are parallel to their base. Of course, the high-dimensional data is not necessarily uniformly-distributed which has led to other data structures with good performance (e.g., the hybrid tree [17]). Clearly, more work needs to be done in this area.

Acknowledgment

This work was supported, in part, by the National Science Foundation under Grants EIA-99-00268, IIS-00-86162, and EIA-00-91474 is gratefully acknowledged.

References

[1] W. G. Aref and H. Samet. Uniquely reporting spatial objects: yet another operation for comparing spatial data structures. In *Proceedings of the 5th International Symposium on Spatial Data Handling*, pages 178–189, Charleston, SC, August 1992.

[2] W. G. Aref and H. Samet. Hashing by proximity to process duplicates in spatial databases. In *Proceedings of the 3rd International Conference on Information*

and Knowledge Management (CIKM), pages 347–354, Gaithersburg, MD, December 1994.

[3] D. Ayala, P. Brunet, R. Juan, and I. Navazo. Object representation by means of nonminimal division quadtrees and octrees. *ACM Transactions on Graphics*, 4(1):41–59, January 1985.

[4] D. H. Ballard. Strip trees: a hierarchical representation for curves. *Communications of the ACM*, 24(5):310–321, May 1981. Also corrigendum, *Communications of the ACM*, 25(3):213, March 1982.

[5] B. G. Baumgart. A polyhedron representation for computer vision. In *Proceedings of the 1975 National Computer Conference*, vol. 44, pages 589–596, Anaheim, CA, May 1975.

[6] R. Bayer and E. M. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1(3):173–189, 1972.

[7] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: an efficient and robust access method for points and rectangles. In *Proceedings of the ACM SIGMOD Conference*, pages 322–331, Atlantic City, NJ, June 1990.

[8] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, September 1975.

[9] J. L. Bentley. Algorithms for Klee's rectangle problems. (unpublished), 1977.

[10] J. L. Bentley and J. H. Friedman. Data structures for range searching. *ACM Computing Surveys*, 11(4):397–409, December 1979.

[11] J. L. Bentley and H. A. Mauer. Efficient worst-case data structures for range searching. *Acta Informatica*, 13:155–168, 1980.

[12] J. L. Bentley, D. F. Stanat, and E. H. Williams Jr. The complexity of finding fixed-radius near neighbors. *Information Processing Letters*, 6(6):209–212, December 1977.

[13] S. Berchtold, C. Böhm, and H.-P. Kriegel. Improving the query performance of high-dimensional index structures by bulk-load operations. In *Advances in Database Technology — EDBT'98, Proceedings of the 6th International Conference on Extending Database Technology*, pages 216–230, Valencia, Spain, March 1998.

[14] S. Berchtold, C. Böhm, and H.-P. Kriegel. The pyramid-technique: Towards breaking the curse of dimensionality. In *Proceedings of the ACM SIGMOD Conference*, L. Hass and A. Tiwary, eds., pages 142–153, Seattle, WA, June 1998.

[15] S. Berchtold, C. Böhm, H.-P. Kriegel, J. Sander, and H. V. Jagadish. Independent quantization: An index compression technique for high-dimensional data spaces. In *Proceedings of the 16th IEEE International Conference on Data Engineering*, pages 577–588, San Diego, CA, February 2000.

[16] K. S. Beyer, J. Goldstein, R. Ramakrishnan, and U. Shaft. When is "nearest neighbor" meaningful? In *Proceedings of the 7th International Conference on Database Theory (ICDT'99)*, C. Beeri and P. Buneman, eds., pages 217–235, Berlin, Germany, January 1999. Also Springer-Verlag Lecture Notes in Computer Science 1540.

[17] K. Chakrabarti and S. Mehrotra. The hybrid tree: an index structure for high dimensional feature spaces. In *Proceedings of the 15th IEEE International Conference on Data Engineering*, pages 440–447, Sydney, Australia, March 1999.

[18] D. Comer. The ubiquitous B-tree. *ACM Computing Surveys*, 11(2):121–137, June 1979.

[19] J.-P. Dittrich and B. Seeger. Data redundancy and duplicate detection in spatial join processing. In *Proceedings of the 16th IEEE International Conference on Data Engineering*, pages 535–546, San Diego, CA, February 2000.

[20] H. Edelsbrunner. Dynamic rectangle intersection searching. Institute for Information Processing 47, Technical University of Graz, Graz, Austria, February 1980.

[21] H. Ferhatosmanoglu, E. Tuncel, D. Agrawal, and A. El Abbadi. Vector approximation based indexing for non-uniform high dimensional data sets. In *Proceedings of the 9th International Conference on Information and Knowledge Management (CIKM)*, pages 202–209, McLean, VA, November 2000.

[22] R. A. Finkel and J. L. Bentley. Quad trees: a data structure for retrieval on composite keys. *Acta Informatica*, 4(1):1–9, 1974.

[23] A. U. Frank and R. Barrera. The Fieldtree: a data structure for geographic information systems. In *Design and Implementation of Large Spatial Databases — 1st Symposium, SSD'89*, A. Buchmann, O. Günther, T. R. Smith, and Y.-F. Wang, eds., pages 29–44, Santa Barbara, CA, July 1989. Also Springer-Verlag Lecture Notes in Computer Science 409.

[24] W. R. Franklin. Adaptive grids for geometric operations. *Cartographica*, 21(2&3):160–167, Summer & Autumn 1984.

[25] E. Fredkin. Trie memory. *Communications of the ACM*, 3(9):490–499, September 1960.

[26] H. Freeman. Computer processing of line-drawing images. *ACM Computing Surveys*, 6(1):57–97, March 1974.

[27] M. Freeston. The BANG file: a new kind of grid file. In *Proceedings of the ACM SIGMOD Conference*, pages 260–269, San Francisco, May 1987.

[28] M. Freeston. A general solution of the n-dimensional B-tree problem. In *Proceedings of the ACM SIGMOD Conference*, pages 80–91, San Jose, CA, May 1995.

[29] H. Fuchs, Z. M. Kedem, and B. F. Naylor. On visible surface generation by a priori tree structures. *Computer Graphics*, 14(3):124–133, July 1980. Also *Proceedings of the SIGGRAPH'80 Conference*, Seattle, WA, July 1980.

[30] O. Günther. *Efficient structures for geometric data management.* PhD thesis, University of California at Berkeley, Berkeley, CA, 1987. Also Lecture Notes in Computer Science 337, Springer-Verlag, Berlin, West Germany, 1988; UCB/ERL M87/77.

[31] A. Guttman. R-trees: a dynamic index structure for spatial searching. In *Proceedings of the ACM SIGMOD Conference*, pages 47–57, Boston, June 1984.

[32] A. Henrich, H. W. Six, and P. Widmayer. The LSD tree: spatial access to multidimensional point and non-point data. In *Proceedings of the 15th International Conference on Very Large Databases (VLDB)*, P. M. G. Apers and G. Wiederhold, eds., pages 45–53, Amsterdam, The Netherlands, August 1989.

[33] K. Hinrichs and J. Nievergelt. The grid file: a data structure designed to support proximity queries on spatial objects. In *Proceedings of WG'83, International Workshop on Graphtheoretic Concepts in Computer Science*, M. Nagl and J. Perl, eds., pages 100–113, Trauner Verlag, Linz, Austria, 1983.

[34] G. R. Hjaltason and H. Samet. Ranking in spatial databases. In *Advances in Spatial Databases — 4th International Symposium, SSD'95*, M. J. Egenhofer and J. R. Herring, eds., pages 83–95, Portland, ME, August 1995. Also Springer-Verlag Lecture Notes in Computer Science 951.

[35] G. R. Hjaltason and H. Samet. Distance browsing in spatial databases. *ACM Transactions on Database Systems*, 24(2):265–318, June 1999. Also Computer Science TR-3919, University of Maryland, College Park, MD.

[36] G. R. Hjaltason and H. Samet. Index-driven similarity search in metric spaces. *ACM Transactions on Database Systems*, 2003. To appear.

[37] E. G. Hoel and H. Samet. Efficient processing of spatial queries in line segment databases. In *Advances in Spatial Databases — 2nd Symposium, SSD'91*, O. Günther and H.-J. Schek, eds., pages 237–256, Zurich, Switzerland, August 1991. Also Springer-Verlag Lecture Notes in Computer Science 525.

[38] E. G. Hoel and H. Samet. A qualitative comparison study of data structures for large line segment databases. In *Proceedings of the ACM SIGMOD Conference*, M. Stonebraker, ed., pages 205–214, San Diego, CA, June 1992.

[39] G. M. Hunter. *Efficient computation and data structures for graphics.* PhD thesis, Department of Electrical Engineering and Computer Science, Princeton University, Princeton, NJ, 1978.

[40] G. M. Hunter and K. Steiglitz. Operations on images using quad trees. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 1(2):145–153, April 1979.

[41] E. Jacox and H. Samet. Iterative spatial join. *ACM Transactions on Database Systems*, 28(3):268–294, September 2003.

[42] G. Kedem. The quad-CIF tree: a data structure for hierarchical on-line algorithms. In *Proceedings of the 19th Design Automation Conference*, pages 352–357, Las Vegas, NV, June 1982.

[43] A. Klinger. Patterns and search statistics. In *Optimizing Methods in Statistics*, J. S. Rustagi, ed., pages 303–337. Academic Press, New York, 1971.

[44] K. Knowlton. Progressive transmission of grey-scale and binary pictures by simple efficient, and lossless encoding schemes. *Proceedings of the IEEE*, 68(7):885–896, July 1980.

[45] D. E. Knuth. *The Art of Computer Programming: Sorting and Searching*, vol. 3. Addison-Wesley, Reading, MA, 1973.

[46] D. E. Knuth. Big omicron and big omega and big theta. *SIGACT News*, 8(2):18–24, April-June 1976.

[47] G. L. Lai, D. Fussell, and D. F. Wong. HV/VH trees: a new spatial data structure for fast region queries. In *Proceedings of the 30th ACM/IEEE Design Automation Conference*, pages 43–47, Dallas, June 1993.

[48] M. Lindenbaum and H. Samet. A probabilistic analysis of trie-based sorting of large collections of line segments. Computer Science Department TR-3455, University of Maryland, College Park, MD, April 1995.

[49] D. Lomet and B. Salzberg. The hB–tree: a multi-attribute indexing method with good guaranteed performance. *ACM Transactions on Database Systems*, 15(4):625–658, December 1990. Also Northeastern University Technical Report NU-CCS-87-24.

[50] E. M. McCreight. Efficient algorithms for enumerating intersecting intervals and rectangles. Technical Report CSL-80-09, Xerox Palo Alto Research Center, Palo Alto, CA, June 1980.

[51] E. M. McCreight. Priority search trees. *SIAM Journal on Computing*, 14(2):257–276, May 1985.

[52] D. Meagher. Octree encoding: a new technique for the representation, manipulation, and display of arbitrary 3-D objects by computer. Electrical and Systems Engineering IPL-TR-80-111, Rensselaer Polytechnic Institute, Troy, NY, October 1980.

[53] D. Meagher. Geometric modeling using octree encoding. *Computer Graphics and Image Processing*, 19(2):129–147, June 1982.

[54] R. C. Nelson and H. Samet. A consistent hierarchical representation for vector data. *Computer Graphics*, 20(4):197–206, August 1986. Also *Proceedings of the SIGGRAPH'86 Conference*, Dallas, TX, August 1986.

[55] J. Nievergelt, H. Hinterberger, and K. C. Sevcik. The grid file: an adaptable, symmetric multikey file structure. *ACM Transactions on Database Systems*, 9(1):38–71, March 1984.

[56] J. A. Orenstein. Multidimensional tries used for associative searching. *Information Processing Letters*, 14(4):150–157, June 1982.

[57] F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction.*

Springer–Verlag, New York, 1985.

[58] A. A. G. Requicha. Representations of rigid solids: theory, methods, and systems. *ACM Computing Surveys*, 12(4):437–464, December 1980.

[59] J. T. Robinson. The K-D-B-tree: a search structure for large multidimensional dynamic indexes. In *Proceedings of the ACM SIGMOD Conference*, pages 10–18, Ann Arbor, MI, April 1981.

[60] H. Samet. *Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS*. Addison-Wesley, Reading, MA, 1990.

[61] H. Samet. Decoupling: A spatial indexing solution. Computer Science TR-4523, University of Maryland, College Park, MD, August 2003.

[62] H. Samet. *Foundations of Multidimensional Data Structures*. To appear, 2004.

[63] H. Samet. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, Reading, MA, 1990.

[64] H. Samet, C. A. Shaffer, and R. E. Webber. Digitizing the plane with cells of non–uniform size. *Information Processing Letters*, 24(6):369–375, April 1987.

[65] H. Samet and M. Tamminen. Efficient component labeling of images of arbitrary dimension represented by linear bintrees. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 10(4):579–586, July 1988.

[66] H. Samet and R. E. Webber. Storing a collection of polygons using quadtrees. *ACM Transactions on Graphics*, 4(3):182–222, July 1985. Also *Proceedings of Computer Vision and Pattern Recognition'83*, pages 127–132, Washington, DC, June 1983 and University of Maryland Computer Science TR-1372.

[67] T. Sellis, N. Roussopoulos, and C. Faloutsos. The R^+–tree: A dynamic index for multi-dimensional objects. In *Proceedings of the 13th International Conference on Very Large Databases (VLDB)*, P. M. Stocker and W. Kent, eds., pages 71–79, Brighton, United Kingdom, September 1987. Also Computer Science TR-1795, University of Maryland, College Park, MD.

[68] M. Shneier. Two hierarchical linear feature representations: edge pyramids and edge quadtrees. *Computer Graphics and Image Processing*, 17(3):211–224, November 1981.

[69] J.-W. Song, K.-Y. Whang, Y.-K. Lee, M.-J. Lee, and S.-W. Kim. Spatial join processing using corner transformation. *IEEE Transactions on Knowledge and Data Engineering*, 11(4):688–695, July/August 1999.

[70] M. Stonebraker, T. Sellis, and E. Hanson. An analysis of rule indexing implementations in data base systems. In *Proceedings of the 1st International Conference on Expert Database Systems*, pages 353–364, Charleston, SC, April 1986.

[71] M. Tamminen. The EXCELL method for efficient geometric access to data. *Acta Polytechnica Scandinavica*, 1981. Also Mathematics and Computer Science Series No. 34.

[72] M. Tamminen. Comment on quad- and octtrees. *Communications of the ACM*, 27(3):248–249, March 1984.

[73] W. Wang, J. Yang, and R. Muntz. PK-tree: a spatial index structure for high dimensional point data. In *Proceedings of the 5th International Conference on Foundations of Data Organization and Algorithms (FODO)*, K. Tanaka and S. Ghandeharizadeh, eds., pages 27–36, Kobe, Japan, November 1998.

[74] J. E. Warnock. A hidden surface algorithm for computer generated half tone pictures. Computer Science Department TR 4–15, University of Utah, Salt Lake City, June 1969.

[75] R. Weber, H.-J. Schek, and S. Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *Proceedings of the 24th In-*

ternational Conference on Very Large Data Bases (VLDB), A. Gupta, O. Shmueli, and J. Widom, eds., pages 194–205, New York, August 1998.

17

Planar Straight Line Graphs

Siu-Wing Cheng
Hong Kong University of Science and Technology

17.1 Introduction

Graphs (Chapter 4) have found extensive applications in computer science as a modeling tool. In mathematical terms, a graph is simply a collection of vertices and edges. Indeed, a popular graph data structure is the adjacency lists representation [14] in which each vertex keeps a list of vertices connected to it by edges. In a typical application, the vertices model entities and an edge models a relation between the entities corresponding to the edge endpoints. For example, the transportation problem calls for a minimum cost shipping pattern from a set of origins to a set of destinations [2]. This can be modeled as a complete directed bipartite graph. The origins and destinations are represented by two columns of vertices. Each origin vertex is labeled with the amount of supply stored there. Each destination vertex is labeled with the amount of demand required there. The edges are directed from the origin vertices to the destination vertices and each edge is labeled with the unit cost of transportation. Only the adjacency information between vertices and edges are useful and captured, apart from the application dependent information.

In geometric computing, graphs are also useful for representing various diagrams. We restrict our attention to diagrams that are planar graphs embedded in the plane using straight edges without edge crossings. Such diagrams are called *planar straight line graphs* and denoted by PSLGs for short. Examples include Voronoi diagrams, arrangements, and triangulations. Their definitions can be found in standard computational geometry texts such as the book by de Berg et al. [3]. See also Chapters 62, 63 and 64. For completeness, we also provide their definitions in section 17.8. The straight edges in a PSLG partition the plane into regions with disjoint interior. We call these regions *faces*. The adjacency lists representation is usually inadequate for applications that manipulate PSLGs. Consider the problem of locating the face containing a query point in a Delaunay triangulation. One practical algorithm is to walk towards the query point from a randomly chosen starting vertex [11], see Figure 17.1. To support this algorithm, one needs to know the first face

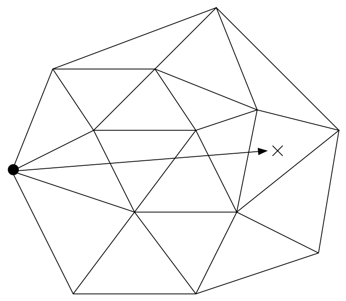


FIGURE 17.1: Locate the face containing the cross by walking from a randomly chosen vertex.

that we enter as well as the next face that we step into whenever we cross an edge. Such information is not readily provided by an adjacency lists representation.

There are three well-known data structures for representing PSLGs: the winged-edge, halfedge, and quadedge data structures. In Sections 17.2 and 17.3, we discuss the PSLGs that we deal with in more details and the operations on PSLGs. Afterwards, we introduce the three data structures in Section 17.4–17.6. We conclude in Section 17.7 with some further remarks.

17.2 Features of PSLGs

We assume that each face has exactly one boundary and we allow dangling edges on a face boundary. These assumptions are valid for many important classes of PSLGs such as triangulations, Voronoi diagrams, planar subdivisions with no holes, arrangements of lines, and some special arrangements of line segments (see Figure 17.2).

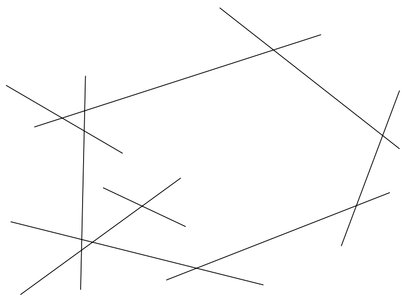


FIGURE 17.2: Dangling edges.

There is at least one unbounded face in a PSLG but there could be more than one, for example, in the arrangement of lines shown in Figure 17.3. The example also shows that there may be some infinite edges. To handle infinite edges like halflines and lines, we need a special vertex v_{inf} at infinity. One can imagine that the PSLG is placed in a small almost flat disk D at the north pole of a giant sphere S and v_{inf} is placed at the south pole. If an edge e is a halfline originating from a vertex u, then the endpoints of e are u and v_{inf}.

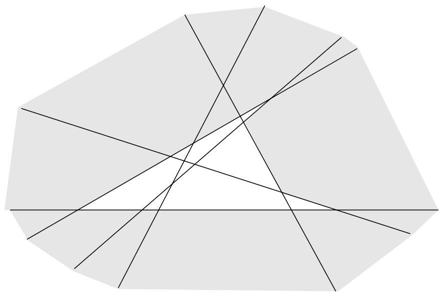


FIGURE 17.3: The shaded faces are the unbounded faces of the arrangement.

One can view e as a curve on S from u near the north pole to v_{inf} at the south pole, but e behaves as a halfline inside the disk D. If an edge e is a line, then v_{inf} is the only endpoint of e. One can view e as a loop from v_{inf} to the north pole and back, but e behaves as a line inside the disk D.

We do not allow isolated vertices, except for v_{inf}. Planarity implies that the incident edges of each vertex are circularly ordered around that vertex. This applies to v_{inf} as well.

A PSLG data structure keeps three kinds of attributes: *vertex attributes*, *edge attributes*, and *face attributes*. The attributes of a vertex include its coordinates except for v_{inf} (we assume that v_{inf} is tagged to distinguish it from other vertices). The attributes of an edge include the equation of the support line of the edge (in the form of $Ax + By + C = 0$). The face attributes are useful for auxiliary information, e.g., color.

17.3 Operations on PSLGs

The operations on a PSLG can be classified into *access functions* and *structural operations*. The access functions retrieve information without modifying the PSLG. Since the access functions partly depend on the data structure, we discuss them later when we introduce the data structures. In this section, we discuss four structural operations on PSLGs: *edge insertion*, *edge deletion*, *vertex split*, and *edge contraction*. We concentrate on the semantics of these four operations and discuss the implementation details later when we introduce the data structures. For vertex split and edge contraction, we assume further that each face in the PSLG is a simple polygon as these two operations are usually used under such circumstances.

Edge insertion and deletion

When a new edge e with endpoints u and v is inserted, we assume that e does not cross any existing edge. If u or v is not an existing vertex, the vertex will be created. If both u and v are new vertices, e is an isolated edge inside a face f. Since each face is assumed to have exactly one boundary, this case happens only when the PSLG is empty and f is the entire plane. Note that e becomes a new boundary of f. If either u or v is a new vertex, then the boundary of exactly one face gains the edge e. If both u and v already exist, then u and v lie on the boundary of a face which is split into two new faces by the insertion of e. These cases are illustrated in Figure 17.4.

The deletion of an edge e has the opposite effects. After the deletion of e, if any of its endpoint becomes an isolated vertex, it will be removed. The vertex v_{inf} is an exception and it is the only possible isolated vertex. The edge insertion is clearly needed to create

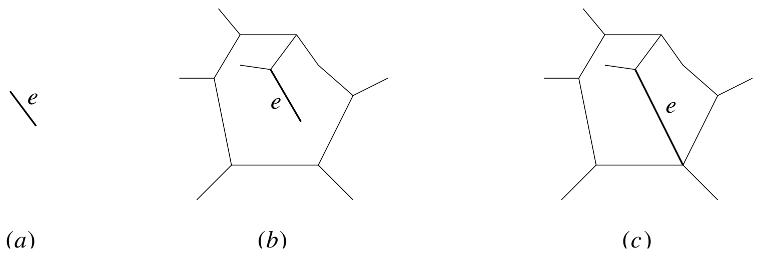


FIGURE 17.4: Cases in edge insertion.

a PSLG from scratch. Other effects can be achieved by combining edge insertions and deletions appropriately. For example, one can use the two operations to overlay two PSLGs in a plane sweep algorithm, see Figure 17.5.

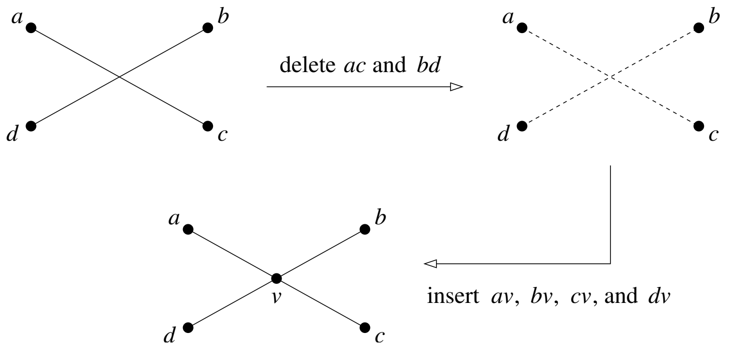


FIGURE 17.5: Intersecting two edges.

Vertex split and edge contraction

The splitting of a vertex v is best visualized as the continuous morphing of v into an edge e. Depending on the specification of the splitting, an incident face of v gains e on its boundary or an incident edge of v is split into a triangular face, see Figure 17.6. The incident edges of v are displaced and it is assumed that no self-intersection occurs within the PSLG during the splitting. The contraction of an edge e is the inverse of the vertex split. We also assume that no self-intersection occurs during the edge contraction. If e is incident on a triangular face, that face will disappear after the contraction of e.

Not every edge can be contracted. Consider an edge ab. If the PSLG contains a cycle abv that is not the boundary of any face incident to ab, we call the edge ab *non-contractible* because its contraction is not cleanly defined. Figure 17.7 shows an example. In the figure, after the contraction, there is an ambiguity whether dv should be incident on the face f_1 or the face f_2. In fact, one would expect the edge dv to behave like av and bv and be incident on both f_1 and f_2 after the contraction. However, this is impossible.

The vertex split and edge contraction have been used in clustering and hierarchical drawing of maximal planar graphs [6].

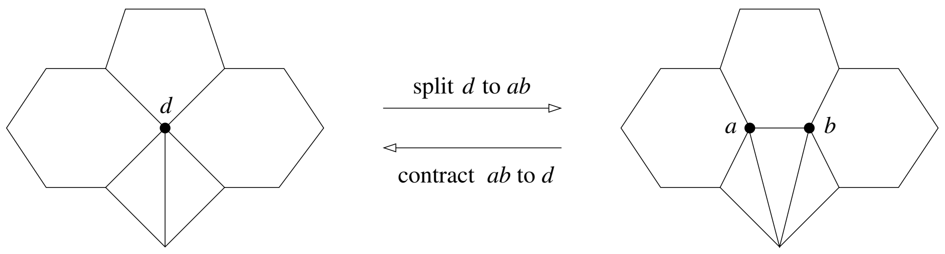


FIGURE 17.6: Vertex split and edge contraction.

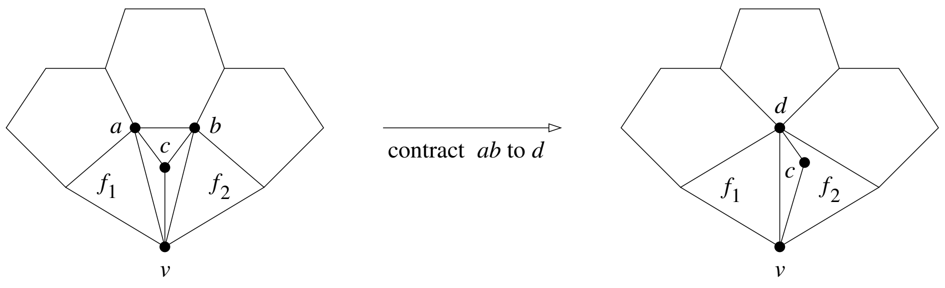


FIGURE 17.7: Non-contractible edge.

17.4 Winged-Edge

The winged-edge data structure was introduced by Baumgart [1] and it predates the halfedge and quadedge data structures. There are three kinds of records: *vertex records*, *edge records*, and *face records*. Each vertex record keeps a reference to one incident edge of the vertex. Each face record keeps a reference to one boundary edge of the face. Each edge e is stored as an oriented edge with the following references (see Figure 17.8):

- The *origin* endpoint $e.org$ and the *destination* endpoint $e.dest$ of e. The convention is that e is directed from $e.org$ to $e.dest$.
- The faces $e.left$ and $e.right$ on the left and right of e, respectively.
- The two edges $e.lcw$ and $e.lccw$ adjacent to e that bound the face $e.left$. The edge $e.lcw$ is incident to $e.org$ and the edge $e.lccw$ is incident to $e.dest$. Note that $e.lcw$ (resp. $e.lccw$) succeeds e if the boundary of $e.left$ is traversed in the clockwise (resp. anti-clockwise) direction from e.
- The two edges $e.rcw$ and $e.rccw$ adjacent to e that bound the face $e.right$. The edge $e.rcw$ is incident to $e.dest$ and the edge $e.rccw$ is incident to $e.org$. Note that $e.rcw$ (resp. $e.rccw$) succeeds e if the boundary of $e.right$ is traversed in the clockwise (resp. anti-clockwise) direction from e.

The information in each edge record can be retrieved in constant time. Given a vertex v, an edge e, and a face f, we can thus answer in constant time whether v is incident on e and e is incident on f. Given a vertex v, we can traverse the edges incident to v in clockwise order as follows. We output the edge e kept at the vertex record for v. We perform $e := e.rccw$ if $v = e.org$ and $e := e.lccw$ otherwise. Then we output e and repeat the above. Given a face f, we can traverse its boundary edges in clockwise order as follows. We output the edge e kept at the face record for f. We perform $e := e.lcw$ if $f = e.left$ and $e := e.rcw$ otherwise.

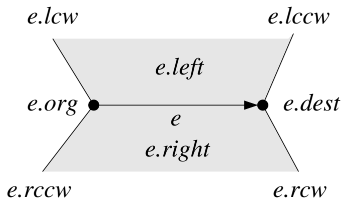


FIGURE 17.8: Winged-edge data structure.

Then we output e and repeat the above.

Note that an edge reference does not carry information about the orientation of the edge. Also, the orientations of the boundary edges of a face need not be consistent with either the clockwise or anti-clockwise traversal. Thus, the manipulation of the data structure is often complicated by case distinctions. We illustrate this with the insertion of an edge e. Assume that $e.org = u$, $e.dest = v$, and both u and v already exist. The input also specifies two edges e_1 and e_2 incident to u and v, respectively. The new edge e is supposed to immediately succeed e_1 (resp. e_2) in the anti-clockwise ordering of edges around u (resp. v). The insertion routine works as follows.

1. If $u = v_{\text{inf}}$ and it is isolated, we need to store the reference to e in the vertex record for u. We update the vertex record for v similarly.
2. Let e_3 be the incident edge of u following e_1 such that e is to be inserted between e_1 and e_3. Note that e_3 succeeds e_1 in anti-clockwise order. We insert e between e_1 and e_3 as follows.

```
$e.rccw := e_1$; $e.lcw := e_3$;
if $e.org = e_1.org$ then $e_1.lcw := e$; else $e_1.rcw := e$;
if $e.org = e_3.org$ then $e_3.rccw := e$; else $e_3.lccw := e$;
```

3. Let e_4 be the incident edge of v following e_2 such that e is to be inserted between e_2 and e_4. Note that e_4 succeeds e_2 in anti-clockwise order. We insert e between e_2 and e_4 as follows.

```
$e.lccw := e_2$; $e.rcw := e_4$;
if $e.dest = e_2.dest$ then $e_2.rcw := e$; else $e_2.lcw := e$;
if $e.dest = e_4.dest$ then $e_4.lccw := e$; else $e_4.rccw := e$;
```

4. The insertion of e has split a face into two. So we create a new face f and make $e.left$ reference it. Also, we store a reference to e in the face record for f. There are further ramifications. First, we make $e.right$ reference the old face.

```
if $e.org = e_1.org$ then $e.right := e_1.left$; else $e.right := e_1.right$;
```

Second, we make the *left* or *right* fields of the boundary edges of f reference f.

```
$e' := e$; $w := e.org$;
repeat
    if $e'.org = w$ then $e'.left := f$; $w := e'.dest$; $e' := e'.lccw$
                    else $e'.right := f$; $w := e'.org$; $e' := e'.rccw$
until $e' = e$;
```

Notice the inconvenient case distinctions needed in steps 2, 3, and 4. The halfedge data structure is designed to keep both orientations of the edges and link them properly. This eliminates most of these case distinctions as well as simplifies the storage scheme.

17.5 Halfedge

In the halfedge data structure, for each edge in the PSLG, there are two symmetric edge records for the two possible orientations of the edge [15]. This solves the orientation problem in the winged-edge data structure. The halfedge data structure is also known as the *doubly connected edge list* [3]. We remark that the name doubly connected edge list was first used to denote the variant of the winged-edge data structure in which the *lccw* and *rccw* fields are omitted [12, 13].

There are three kinds of records: *vertex records*, *halfedge records*, and *face records*. Let e be a halfedge. The following information is kept at the record for e (see Figure 17.9).

- The reference $e.sym$ to the symmetric version of e.
- The *origin* endpoint $e.org$ of e. We do not need to store the destination endpoint of e since it can be accessed as $e.sym.org$. The convention is that e is directed from $e.org$ to $e.sym.org$.
- The face $e.left$ on the left of e.
- The next edge $e.succ$ and the previous edge $e.pred$ in the anti-clockwise traversal around the face $e.left$.

For each vertex v, its record keeps a reference to one halfedge $v.edge$ such that $v = v.edge.org$. For each face f, its record keeps a reference to one halfedge $f.edge$ such that $f = f.edge.left$.

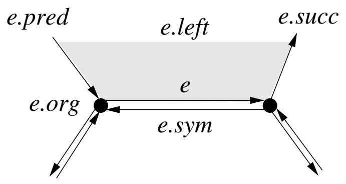


FIGURE 17.9: Halfedge data structure.

We introduce two basic operations `make_halfedges` and `half_splice` which will be needed for implementing the operations on PSLGs. These two operations are motivated by the operations `make_edge` and `splice` introduced by Guibas and Stolfi [8] for the quadedge data structure. We can also do without `make_halfedges` and `half_splice`, but they make things simpler.

- `make_halfedges`(u, v): Return two halfedges e and $e.sym$ connecting the points u and v. The halfedges e and $e.sym$ are initialized such that they represent a new PSLG with e and $e.sym$ as the only halfedges. That is, $e.succ = e.sym = e.pred$ and $e.sym.succ = e = e.sym.pred$. Also, e is the halfedge directed from u to v. If u and v are omitted, it means that the actual coordinates of $e.org$ and $e.sym.org$ are unimportant.
- `half_splice`(e_1, e_2): Given two halfedges e_1 and e_2, `half_splice` swaps the contents of $e_1.pred$ and $e_2.pred$ and the contents of $e_1.pred.succ$ and $e_2.pred.succ$. The effects are:
  - Let $v = e_2.org$. If $e_1.org \neq v$, the incident halfedges of $e_1.org$ and $e_2.org$ are merged into one circular list (see Figure 17.10(a)). The vertex v is now

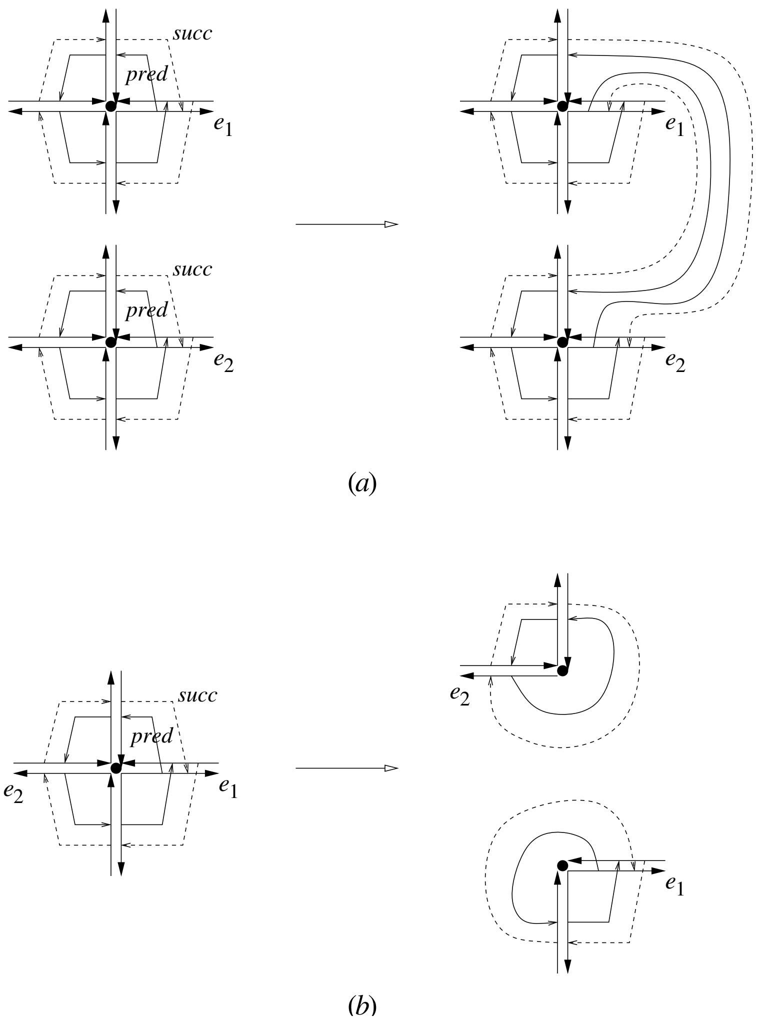


FIGURE 17.10: The effects of `half_splice`.

redundant and we finish the merging as follows.

```
$e' := e_2$;
repeat
    $e'.org := e_1.org$; $e' := e'.sym.succ$;
until $e' = e_2$;
delete the vertex record for $v$;
```

– Let $v = e_2.org$. If $e_1.org = v$, the incident halfedges of v are separated into two circular lists (see Figure 17.10(b)). We create a new vertex u for $e_2.org$ with the coordinates of u left uninitialized. Then we finish the separation as follows.

```
$u.edge := e_2$; $e' := e_2$;
repeat
    $e'.org := u$; $e' := e'.sym.succ$;
until $e' = e_2$.
```

The behavior of `half_splice` is somewhat complex even in the following special cases. If e is an isolated halfedge, `half_splice`(e_1, e) deletes the vertex record for $e.org$ and makes e a halfedge incident to $e_1.org$ following e_1 in anti-clockwise order. If $e_1 = e.sym.succ$, `half_splice`(e_1, e) detaches e from the vertex $e_1.org$ and creates a new vertex record for $e.org$. If $e_1 = e$, `half_splice`(e, e) has no effect at all.

Access functions

The information in each halfedge record can be retrieved in constant time. Given a vertex v, a halfedge e, and a face f, we can thus answer the following adjacency queries:

1: Is v incident on e? This is done by checking if $v = e.org$ or $e.sym.org$.
2: Is e incident on f? This is done by checking if $f = e.left$.
3: List the halfedges with origin v in clockwise order. Let $e = v.edge$. Output e, perform $e := e.sym.succ$, and then repeat until we return to $v.edge$.
4: List the boundary halfedges of f in anti-clockwise order. Let $e = f.edge$. Output e, perform $e := e.succ$, and then repeat until we return to $f.edge$.

Other adjacency queries (e.g., listing the boundary vertices of a face) can be answered similarly.

Edge insertion and deletion

The edge insertion routine takes two vertices u and v and two halfedges e_1 and e_2. If u is a new vertex, e_1 is ignored; otherwise, we assume that $e_1.org = u$. Similarly, if v is a new vertex, e_2 is ignored; otherwise, we assume that $e_2.org = v$. The general case is that an edge connecting u and v is inserted between e_1 and $e_1.pred.sym$ and between e_2 and $e_2.pred.sym$. The two new halfedges e and $e.sym$ are returned with the convention that e is directed from u to v.

```
Algorithm insert($u, v, e_1, e_2$)
1.  $(e, e.sym)$ := make_halfedges($u, v$);
2.  if $u$ is not new
3.     then half_splice($e_1, e$);
4.             $e.left := e_1.left$;
5.             $e.sym.left := e_1.left$;
6.  if $v$ is not new
7.     then half_splice($e_2, e.sym$);
8.             $e.left := e_2.left$;
9.             $e.sym.left := e_2.left$;
10. if neither $u$ nor $v$ is new
11.    then /* A face has been split */
12.            $e_2.left.edge := e$;
13.            create a new face $f$;
14.            $f.edge := e.sym$;
15.            $e' := e.sym$;
16.            repeat
17.                $e'.left := f$;
18.                $e' := e'.succ$;
19.            until $e' = e.sym$;
20. return $(e, e.sym)$;
```

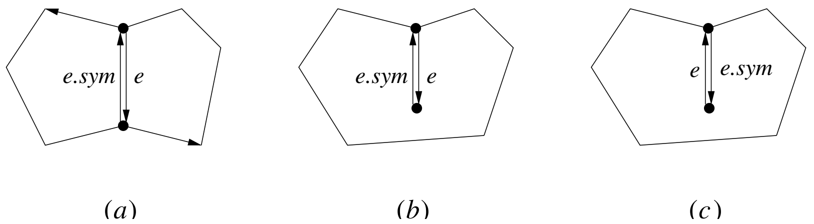


FIGURE 17.11: Cases in deletion.

The following deletion algorithm takes the two halfedges e and $e.sym$ corresponding to the edge to be deleted. If the edge to be deleted borders two adjacent faces, they have to be merged after the deletion.

```
Algorithm delete($e, e.sym$)
1.  if $e.left \neq e.sym.left$
2.     then /* Figure 17.11(a) */
3.           /* the faces adjacent to $e$ and $e.sym$ are to be merged */
4.           delete the face record for $e.sym.left$;
5.           $e' := e.sym$;
6.           repeat
7.                $e'.left := e.left$;
8.                $e' := e'.succ$;
9.           until $e' = e.sym$;
10.          $e.left.edge := e.succ$;
11.          half_splice($e.sym.succ, e$);
12.          half_splice($e.succ, e.sym$);
13.    else  if $e.succ = e.sym$
14.             then /* Figure 17.11(b) */
15.                   $e.left.edge := e.pred$;
16.                   half_splice($e.sym.succ, e$);
17.             else  /* Figure 17.11(c) */
18.                   $e.left.edge := e.succ$;
19.                   half_splice($e.succ, e.sym$);
20. /* $e$ becomes an isolated edge */
21. delete the vertex record for $e.org$ if $e.org \neq v_{\mathrm{inf}}$;
22. delete the vertex record for $e.sym.org$ if $e.sym.org \neq v_{\mathrm{inf}}$;
23. delete the halfedges $e$ and $e.sym$;
```

Vertex split and edge contraction

Recall that each face is assumed to be a simple polygon for the vertex split and edge contraction operations. The vertex split routine takes two points (p, q) and (x, y) and four halfedges e_1, e_2, e_3, and e_4 in anti-clockwise order around the common origin v. It is required that either $e_1 = e_2$ or $e_1.pred = e_2.sym$ and either $e_3 = e_4$ or $e_3.pred = e_4.sym$. The routine splits v into an edge e connecting the points (p, q) and (x, y). Also, e borders the faces bounded by e_1 and e_2 and by e_3 and e_4. Note that if $e_1 = e_2$, we create a new face bounded by e_1, e_2, and e. Similarly, a new face is created if $e_3 = e_4$. The following is the vertex split algorithm.

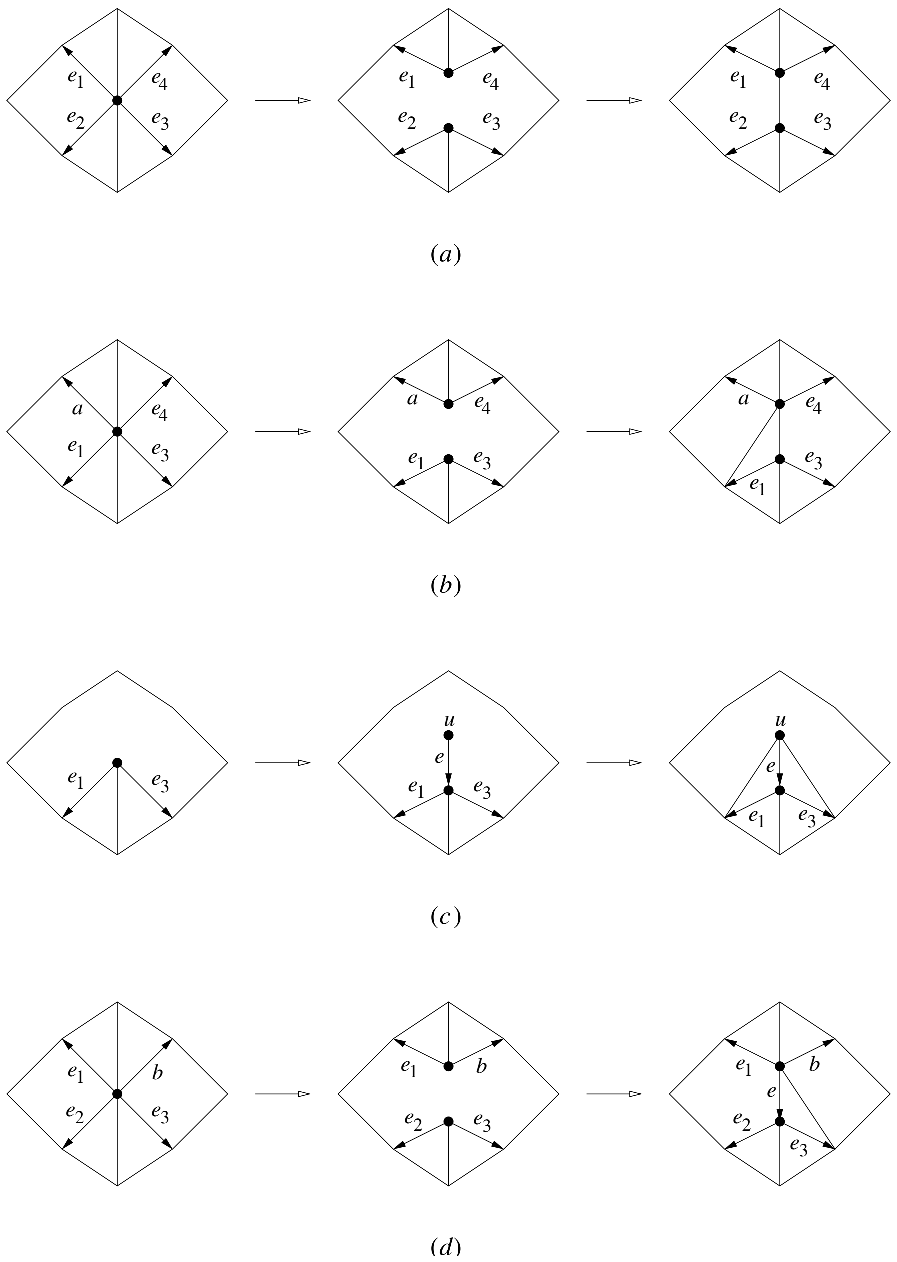


FIGURE 17.12: Cases for `split`.

```
Algorithm split$(p, q, x, y, e_1, e_2, e_3, e_4)$
1.  if $e_1 \neq e_2$ and $e_3 \neq e_4$
2.      then /* Figure 17.12(a) */
3.          half_splice$(e_1, e_3)$;
4.          insert$(e_1.org, e_3.org, e_1, e_3)$;
5.          set the coordinates of $e_3.org$ to $(x, y)$;
6.          set the coordinates of $e_1.org$ to $(p, q)$;
```

```
7.    else  if $e_1 = e_2$
8.          then $a := e_1.sym.succ$;
9.               if $a \neq e_3$
10.                then /* Figure 17.12(b) */
11.                     half_splice($a, e_3$);
12.                     insert($a.org, e_3.org, a, e_3$);
13.                     insert($a.org, e_1.sym.org, a, e_1.sym$);
14.                     set the coordinates of $a.org$ to $(x, y)$;
15.                else /* Figure 17.12(c) */
16.                     let $u$ be a new vertex at $(x, y)$;
17.                     $(e, e.sym)$ := insert($u, e_1.org, \cdot, e_3$);
18.                     insert($u, e_1.sym.org, e, e_1.sym$);
19.                     insert($u, e_3.sym.org, e, e_3.succ$);
20.              set the coordinates of $e_1.org$ to $(p, q)$;
21.         else $b := e_3.pred.sym$;
22.              /* since $e_1 \neq e_2$, $b \neq e_2$ */
23.              /* Figure 17.12(d) */
24.              half_splice($e_1, e_3$);
25.              $(e, e.sym)$ := insert($b.org, e_3.org, e_1, e_3$);
26.              insert($b.org, e_3.sym.org, e, e_3.succ$);
27.              set the coordinates of $b.org$ to $(x, y)$;
28.              set the coordinates of $e_3.org$ to $(p, q)$;
```

The following algorithm contracts an edge to a point (x, y), assuming that the edge contractibility has been checked.

```
Algorithm contract($e, e.sym, x, y$)
1.  $e_1 := e.succ$;
2.  $e_2 := e.pred.sym$;
3.  $e_3 := e.sym.succ$;
4.  $e_4 := e.sym.pred.sym$;
5.  delete($e, e.sym$);
6.  if $e_1.succ \neq e_2.sym$ and $e_3.succ \neq e_4.sym$
7.    then /* Figure 17.13(a) */
8.         half_splice($e_1, e_3$);
9.    else  if $e_1.succ = e_2.sym$ and $e_3.succ \neq e_4.sym$
10.         then /* Figure 17.13(b) */
11.              delete($e_2, e_2.sym$);
12.              half_splice($e_1, e_3$);
13.         else  if $e_1.succ \neq e_2.sym$ and $e_3.succ = e_4.sym$
14.                 then /* symmetric to Figure 17.13(b) */
15.                      delete($e_4, e_4.sym$);
16.                      half_splice($e_1, e_3$);
17.                 else /* Figure 17.13(c) */
18.                      $a := e_3.sym.succ$;
19.                      delete($e_3, e_3.sym$);
20.                      if $a \neq e_2$
21.                        then delete($e_2, e_2.sym$);
22.                             half_splice($e_1, a$);
23.                        else delete($e_2, e_2.sym$);
24. set the coordinates of $e_1.org$ to $(x, y)$;
```

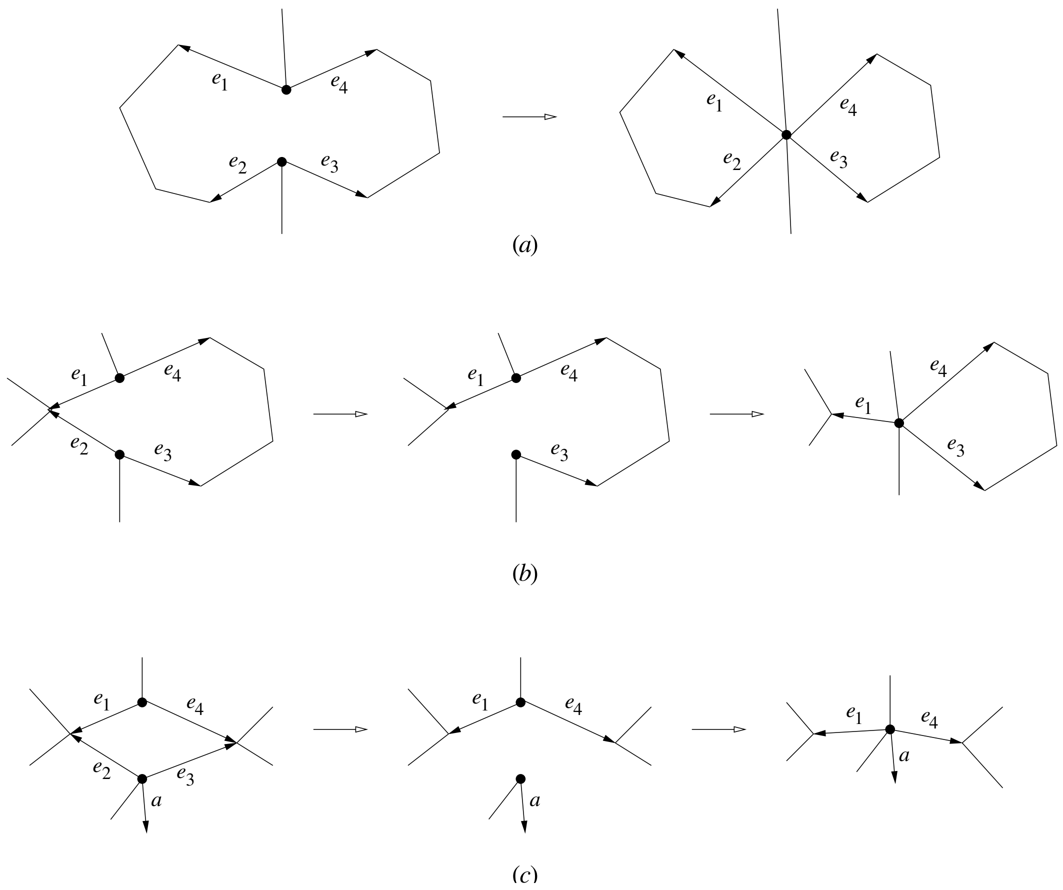


FIGURE 17.13: Cases for `contract`.

17.6 Quadedge

The quadedge data structure was introduced by Guibas and Stolfi [8]. It represents the planar subdivision and its dual simultaneously. The dual S^* of a PSLG S is constructed as follows. For each face of S, put a dual vertex inside the face. For each edge of S bordering the faces f and f', put a dual edge connecting the dual vertices of f and f'. The dual of a vertex v in S is a face and this face is bounded by the dual of the incident edges of v. Figure 17.14 shows an example. The dual may have loops and two vertices may be connected by more than one edge, so the dual may not be a PSLG. Nevertheless, the quadedge data structure is expressive enough to represent the dual. In fact, it is powerful enough to represent subdivisions of both orientable and non-orientable surfaces. We describe a simplified version sufficient for our purposes.

Each edge e in the PSLG is represented by four quadedges $e[i]$, where $i \in \{0, 1, 2, 3\}$. The quadedges $e[0]$ and $e[2]$ are the two oriented versions of e. The quadedges $e[1]$ and $e[3]$ are the two oriented versions of the dual of e. These four quadedges are best viewed as a cross such as $e[i + 1]$ is obtained by rotating $e[i]$ for $\pi/2$ in the anti-clockwise direction. This is illustrated in Figure 17.15. The quadedge $e[i]$ has a *next* field referencing the quadedge that has the same origin as $e[i]$ and follows $e[i]$ in anti-clockwise order. In effect, the *next* fields form a circular linked list of quadedges with a common origin. This is called an *edge ring*.

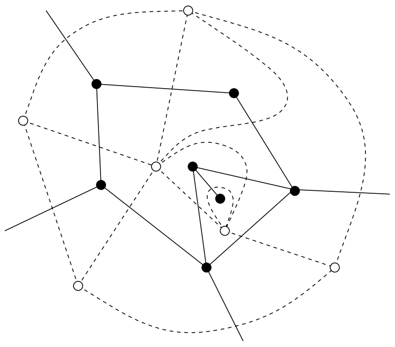


FIGURE 17.14: The solid lines and black dots show a PSLG and the dashed lines and the white dots denote the dual.

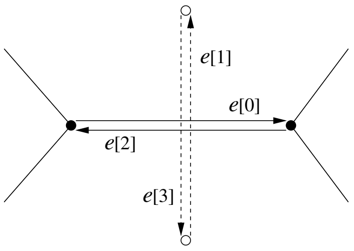


FIGURE 17.15: Quadedges.

The following primitives are needed.

- $\texttt{rot}(e, i)$: Return $e[(i + 1) \bmod 4]$.
- $\texttt{rot}^{-1}(e, i)$: Return $e[(i + 3) \bmod 4]$.
- $\texttt{sym}(e, i)$: This function returns the quadedge with the opposite orientation of $e[i]$. This is done by returning $\texttt{rot}(\texttt{rot}(e, i))$.
- $\texttt{onext}(e, i)$: Return $e[i].next$.
- $\texttt{oprev}(e, i)$: This function gives the quadedge that has the same origin as $e[i]$ and follows $e[i]$ in clockwise order. This is done by returning $\texttt{rot}(e[(i+1) \bmod 4].next)$.

The quadedge data structure is entirely edge based and there are no explicit vertex and face records.

The following two basic operations make_edge and splice are central to the operations on PSLGs supported by the quadedge data structure. Our presentation is slightly different from that in the original paper [8].

- $\texttt{make\_edge}(u, v)$: Return an edge e connecting the points u and v. The quadedges $e[i]$ where $0 \leq i \leq 3$ are initialized such that they represent a new PSLG with e as the only edge. Also, $e[0]$ is the quadedge directed from u to v. If u and v are omitted, it means that the actual coordinates of the endpoints of are unimportant.
- $\texttt{splice}(a, i, b, j)$: Given two quadedges $a[i]$ and $b[j]$, let $(c, k) = \texttt{rot}(a[i].next)$ and $(d, l) = \texttt{rot}(b[j].next)$, splice swaps the contents of $a[i].next$ and $b[j].next$ and the contents of $c[k].next$ and $d[l].next$. The effects on the edge rings of the origins of $a[i]$ and $b[j]$ and the edge rings of the origins of $c[k]$ and $d[l]$ are:

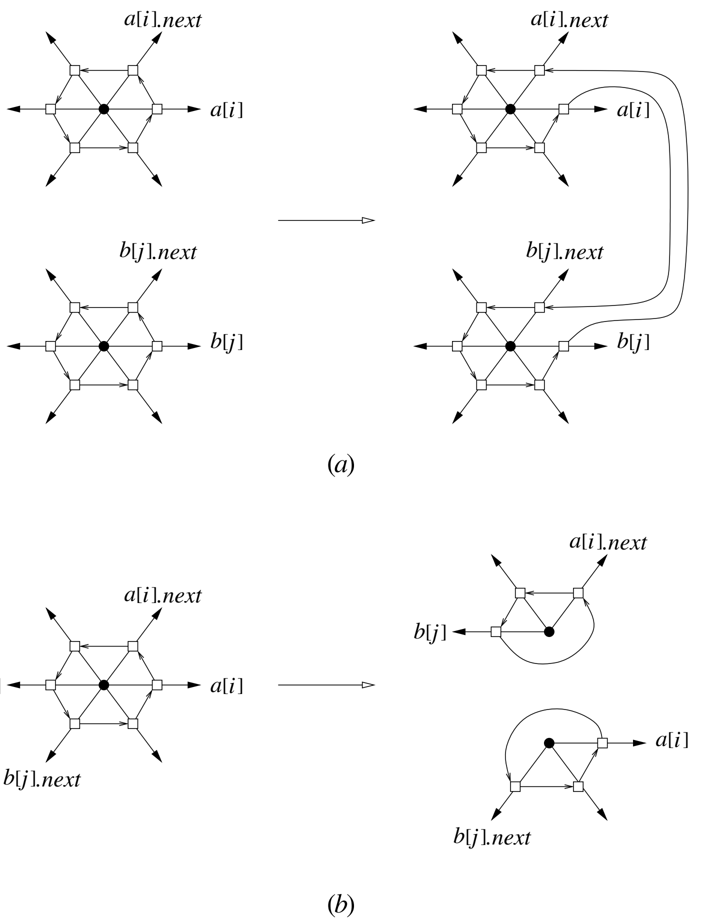


FIGURE 17.16: The effects of `splice`.

– If the two rings are different, they are merged into one (see Figure 17.16(a)).

– If the two rings are the same, it will be split into two separate rings (see Figure 17.16(b)).

Notice that `make_edge` and `splice` are similar to the operations `make_halfedges` and `half_splice` introduced for the halfedge data structure in the previous section. As mentioned before, they inspire the definitions of `make_halfedges` and `half_splice`. Due to this similarity, one can easily adapt the edge insertion, edge deletion, vertex split, and edge contraction algorithms in the previous section for the quadedge data structure.

17.7 Further Remarks

We have assumed that each face in the PSLG has exactly one boundary. This requirement can be relaxed for the winged-edge and the halfedge data structures. One method works as follows. For each face f, pick one edge from each boundary and keep a list of references to these edges at the face record for f. Also, the edge that belongs to outer boundary of f is specially tagged. With this modification, one can traverse the boundaries of a face

f consistently (e.g., keeping f on the left of traversal direction). The edge insertion and deletion algorithms also need to be enhanced. Since a face f may have several boundaries, inserting an edge may combine two boundaries without splitting f. If the insertion indeed splits f, one needs to distribute the other boundaries of f into the two faces resulting from the split. The reverse effects of edge deletion should be taken care of similarly.

The halfedge data structure has also been used for representing orientable polyhedral surfaces [10]. The full power of the quadedge data structure is only realized when one deals with both subdivisions of orientable and non-orientable surfaces. To this end, one needs to introduce a *flip bit* to allow viewing the surface from the above or below. The primitives need to be enhanced for this purpose. The correctness of the data structure is proven formally using *edge algebra*. The details are in the Guibas and Stolfi's original paper [8].

The vertex split and edge contraction are also applicable for polyhedral surfaces. The edge contractibility criteria carries over straightforwardly. Edge contraction is a popular primitive for surface simplification algorithms [4, 7, 9]. The edge contractibility criteria for non-manifolds has also been studied [5].

17.8 Glossary

Arrangements. Given a collection of lines, we split each line into edges by inserting a vertex at every intersection on the line. The resulting PSLG is called the *arrangement of lines*. The *arrangement of line segments* is similarly defined.

Voronoi diagram. Let S be a set of points in the plane. For each point $p \in S$, the Voronoi region of p is defined to be $\{x \in R^2 : \|p - x\| \leq \|q - x\|, \forall q \in S\}$. The *Voronoi diagram* of S is the collection of all Voronoi regions (including their boundaries).

Triangulation. Let S be a set of points in the plane. Any maximal PSLG with the points in S as vertices is a *triangulation* of S.

Delaunay triangulation. Let S be a set of points in the plane. For any three points p, q, and r in S, if the circumcircle of the triangle pqr does not strictly enclose any point in S, we call pqr a *Delaunay triangle*. The *Delaunay triangulation* of S is the collection of all Delaunay triangles (including their boundaries). The Delaunay triangulation of S is the dual of the Voronoi diagram of S.

Acknowledgment

This work was supported, in part, by the Research Grant Council, Hong Kong, China (HKUST 6190/02E).

References

[1] B.G. Baumgart, A polyhedron representation for computer vision, *National Computer conference*, 589–596, Anaheim, CA, 1975, AFIPS.

[2] M.S. Bazaraa, J.J. Jarvis, and H.D. Sherali, *Linear Programming and Network Flows*, Wiley, 1990.

[3] M. deBerg, M. van Kreveld, M. Overmars, and O. Schwarzkopf, *Computational Geometry – Algorithms and Applications*, Springer, 2000.

[4] S.-W. Cheng, T. K. Dey, and S.-H. Poon, Hierarchy of Surface Models and Irreducible

Triangulation, *Computational Geometry: Theory and Applications*, 27(2004), 135–150.

[5] T.K. Dey, H. Edelsbrunner, S. Guha, and D.V. Nekhayev, Topology preserving edge contraction, *Publ. Inst. Math. (Beograd) (N.S.)*, 66 (1999), 23–45.

[6] C.A. Duncan, M.T. Goodrich, and S.G. Kobourov, Planarity-preserving clustering and embedding for large graphs, *Proc. Graph Drawing*, Lecture Notes Comput. Sci., Springer-Verlag, vol. 1731, 1999, 186–196.

[7] M. Garland and P.S. Heckbert. Surface simplification using quadric error metrics. *Proc. SIGGRAPH '97*, 209–216.

[8] L. Guibas and J. Stolfi, Primitives for the manipulation of general subdivisions and the computation of Voronoi diagrams, *ACM Transactions on Graphics*, 4 (1985), 74–123.

[9] H. Hoppe, T. DeRose, T. Duchamp, J. McDonald and W. Stuetzle, Mesh optimization, *Proc. SIGGRAPH '93*, 19–26.

[10] L. Kettner, Using generic programming for designing a data structure for polyhedral surfaces, *Computational Geometry - Theory and Applications*, 13 (1999), 65–90.

[11] E. Mücke, I. Saias, and B. Zhu, Fast randomized point location without preprocessing in two and three-dimensional Delaunay triangulations, *Computational Geometry: Theory and Applications*, 12(1999), 63-83, 1999.

[12] D.E. Muller and F.P. Preparata, Finding the intersection of two convex polyhedra, *Theoretical Computer Science*, 7 (1978), 217–236.

[13] F.P. Preparata and M.I. Shamos, *Computational Geometry: An Introduction*, Springer-Verlag, New York, 1985.

[14] S. Sahni, *Data Structures, Algorithms, and Applications in Java*, McGraw Hill, NY, 2000.

[15] K. Weiler, Edge-based data structures for solid modeling in curved-surface environments. *IEEE Computer Graphics and Application*, 5 (1985), 21–40.

18

Interval, Segment, Range, and Priority Search Trees

D. T. Lee
Academia Sinica

18.1 Introduction

In this chapter we introduce four basic data structures that are of fundamental importance and have many applications as we will briefly cover them in later sections. They are *interval trees*, *segment trees*, *range trees*, and *priority search trees*. Consider for example the following problems. Suppose we have a set of *iso-oriented rectangles* in the planes. A set of rectangles are said to be *iso-oriented* if their edges are parallel to the coordinate axes. The subset of iso-oriented rectangles define a *clique*, if their common intersection is nonempty. The *largest* subset of rectangles whose common intersection is non-empty is called a *maximum clique*. The problem of finding this largest subset with a non-empty common intersection is referred to as the *maximum clique problem* for a rectangle intersection graph[14, 16].[*] The k-dimensional, $k \geq 1$, analog of this problem is defined similarly. In 1-dimensional case we will have a set of *intervals* on the real line, and an *interval intersection graph*, or simply *interval graph*. The maximum clique problem for interval graphs is to find a largest subset of intervals whose common intersection is non-empty. The cardinality of the maximum clique is sometimes referred to as the *density* of the set of intervals.

The problem of finding a subset of objects that satisfy a certain property is often referred to as *searching problem*. For instance, given a set of numbers $S = \{x_1, x_2, \ldots, x_n\}$, where

[*]A rectangle intersection graph is a graph $G = (V, E)$, in which each vertex in V corresponds to a rectangle, and two vertices are connected by an edge in E, if the corresponding rectangles intersect.

$x_i \in \Re$, $i = 1, 2, \ldots, n$, the problem of finding the subset of numbers that lie between a range $[\ell, r]$, i.e., $F = \{x \in S | \ell \leq x \leq r\}$, is called a (1D) *range search* problem[5, 22].

To deal with this kind of geometric searching problem, we need to have appropriate data structures to support efficient searching algorithms. The data structure is assumed to be *static*, i.e., the input set of objects is given *a priori*, and *no* insertions or deletions of the objects are allowed. If the searching problem satisfies *decomposability property*, i.e., if they are *decomposable*†, then there are general *dynamization* schemes available[21], that can be used to convert static data structures into *dynamic* ones, where *insertions* and *deletions* of objects are permitted. Examples of decomposable searching problems include the *membership* problem in which one queries if a point p in S. Let S be partitioned into two subsets S_1 and S_2, and Member(p, S) returns *yes*, if $p \in S$, and *no* otherwise. It is easy to see that Member(p, S)= OR(Member(p, S_1), Member(p, S_2)), where OR is a boolean operator.

18.2 Interval Trees

Consider a set S of intervals, $S = \{I_i | i = 1, 2, \ldots, n\}$, each of which is specified by an ordered pair, $I_i = [\ell_i, r_i], \ell_i, r_i \in \Re, \ell_i \leq r_i, i = 1, 2, \ldots, n$.

An *interval tree*[8, 9], *Interval_Tree*(S), for S is a rooted augmented binary search tree, in which each node v has a key value, $v.key$, two tree pointers $v.left$ and $v.right$ to the left and right subtrees, respectively, and an auxiliary pointer, $v.aux$ to an augmented data structure, and is recursively defined as follows:

- The root node v associated with the set S, denoted Interval_Tree_root(S), has key value $v.key$ equal to the median of the $2 \times |S|$ endpoints. This key value $v.key$ divides S into three subsets S_ℓ, S_r and S_m, consisting of sets of intervals lying totally to the *left* of $v.key$, lying totally to the *right* of $v.key$ and containing $v.key$ respectively. That is, $S_\ell = \{I_i | r_i < v.key\}$, $S_r = \{I_j | v.key < \ell_j\}$ and $S_m = \{I_k | \ell_k \leq v.key \leq r_k\}$.
- Tree pointer $v.left$ points to the left subtree rooted at Interval_Tree_root(S_ℓ), and tree pointer $v.right$ points to the right subtree rooted at Interval_Tree_root(S_r).
- Auxiliary pointer $v.aux$ points to an augmented data structure consisting of two sorted arrays, SA$(S_m.left)$ and SA$(S_m.right)$ of the set of left endpoints of the intervals in S_m and the set of right endpoints of the intervals in S_m respectively. That is, $S_m.left = \{\ell_i | I_i \in S_m\}$ and $S_m.right = \{r_i | I_i \in S_m\}$.

18.2.1 Construction of Interval Trees

The following is a pseudo code for the recursive construction of the interval tree of a set S of n intervals. Without loss of generality we shall assume that the endpoints of these n intervals are all distinct. See Fig. 18.1(a) for an illustration.

```
function Interval_Tree($S$)
/* It returns a pointer $v$ to the root, Interval_Tree_root($S$), of the interval tree for a set $S$
of intervals. */
```

†A searching problem is said to be *decomposable* if and only if $\forall x \in T_1, A, B \in 2^{T_2}, Q(x, A \cup B) = \bigcirc(Q(x, A), Q(x, B))$ for some efficiently computable associative operator $\bigcirc$ on the elements of T_3, where Q is a mapping from $T_1 \times 2^{T_2}$ to T_3.[1, 3]

Input: A set S of n intervals, $S = \{I_i | i = 1, 2, \ldots, n\}$ and each interval $I_i = [\ell_i, r_i]$, where ℓ_i and r_i are the left and right endpoints, respectively of I_i, $\ell_i, r_i \in \Re$, and $\ell_i \leq r_i, i = 1, 2, \ldots, n$.

Output: An interval tree, rooted at Interval_Tree_root(S).

Method:

```
1. if $S = \emptyset$, return nil.
2. Create a node $v$ such that $v.key$ equals $x$, where $x$ is the middle point of
   the set of endpoints so that there are exactly $|S|/2$ endpoints less than $x$
   and greater than $x$ respectively. Let $S_\ell = \{I_i | r_i < x\}$, $S_r = \{I_j | x < \ell_j\}$ and
   $S_m = \{I_k | \ell_k \leq x \leq r_k\}$.
3. Set $v.left$ equal to Interval_Tree($S_\ell$).
4. Set $v.right$ equal to Interval_Tree($S_r$)
5. Create a node $w$ which is the root node of an auxiliary data structure
   associated with the set $S_m$ of intervals, such that $w.left$ and $w.right$ point to
   two sorted arrays, SA($S_m.left$) and SA($S_m.right$), respectively. SA($S_m.left$)
   denotes an array of left endpoints of intervals in $S_m$ in ascending order, and
   SA($S_m.right$) an array of right endpoints of intervals in $S_m$ in descending
   order.
6. Set $v.aux$ equal to node $w$.
```

Note that this recursively built interval tree structure requires $O(n)$ space, where n is the cardinality of S, since each interval is either in the left subtree, the right subtree or the middle augmented data structure.

18.2.2 Example and Its Applications

Fig. 18.1(b) illustrates an example of an interval tree of a set of intervals, spread out as shown in Fig. 18.1(a).

The interval trees can be used to handle quickly queries of the following form.

Enclosing Interval Searching Problem [11, 15] Given a set S of n intervals and a query point, q, report all those intervals containing q, i.e., find a subset $F \subseteq S$ such that $F = \{I_i | \ell_i \leq q \leq r_i\}$.

Overlapping Interval Searching Problem [4, 8, 9] Given a set S of n intervals and a query interval Q, report all those intervals in S overlapping Q, i.e., find a subset $F \subseteq S$ such that $F = \{I_i | I_i \cap Q \neq \emptyset\}$.

The following pseudo code solves the **Overlapping Interval Searching Problem** in $O(\log n) + |F|$ time. It is invoked by a call to Overlapping_Interval_Search(v, Q, F), where v is Interval_Tree_root(S), and F, initially set to be $\emptyset$, will contain the set of intervals overlapping query interval Q.

procedure Overlapping_Interval_Search(v, Q, F)

Input: A set S of n intervals, $S = \{I_i | i = 1, 2, \ldots, n\}$ and each interval $I_i = [\ell_i, r_i]$, where ℓ_i and r_i are the left and right endpoints, respectively of I_i, $\ell_i, r_i \in \Re$, and $\ell_i \leq r_i, i = 1, 2, \ldots, n$ and a query interval $Q = [\ell, r], \ell, r \in \Re$.

Output: A subset $F = \{I_i | I_i \cap Q \neq \emptyset\}$.

Method:

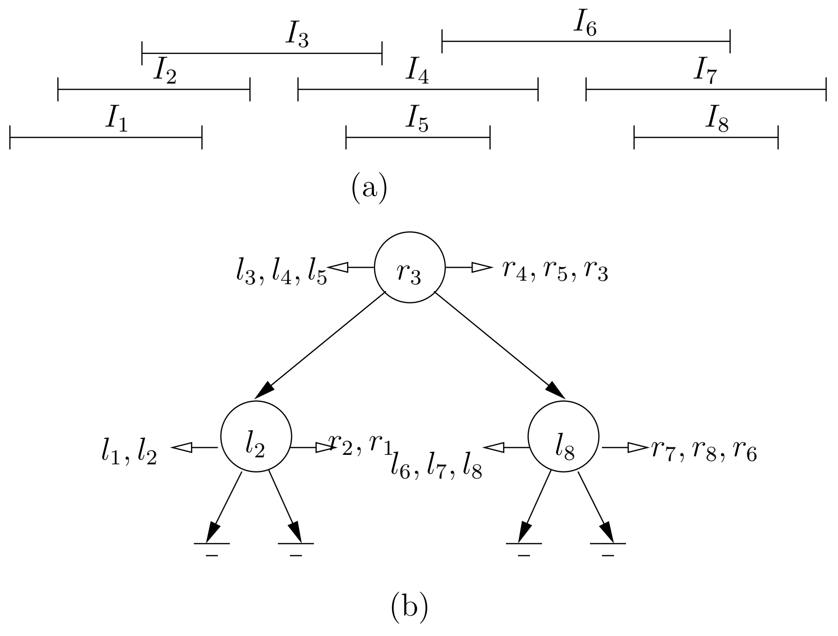


FIGURE 18.1: Interval tree for $S = \{I_1, I_2, \ldots, I_8\}$ and its interval models.

```
1. Set $F = \emptyset$ initially.
2. if $v$ is nil return.
3. if ($v.key \in Q$) then
       for each interval $I_i$ in the augmented data structure pointed to by $v.aux$
       do $F = F \cup \{I_i\}$
       Overlapping_Interval_Search($v.left$, $Q$, $F$)
       Overlapping_Interval_Search($v.right$, $Q$, $F$)
4. if ($r < v.key$) then
   for each left endpoint $\ell_i$ in the sorted array pointed to by $v.aux.left$ such
   that $\ell_i \geq r$ do
   (a) $F = F \cup \{I_i\}$
   (b) Overlapping_Interval_Search($v.left$, $Q$, $F$)
5. if ($\ell > v.key$) then
   for each right endpoint $r_i$ in the sorted array pointed to by $v.aux.right$ such
   that $r_i \geq \ell$ do
   (a) $F = F \cup \{I_i\}$
   (b) Overlapping_Interval_Search($v.right$, $Q$, $F$)
```

It is obvious to see that an interval I in S overlaps a query interval $Q = [\ell, r]$ if (i) Q contains the left endpoint of I, (ii) Q contains the right endpoint of I, or (iii) Q is totally contained in I. Step 3 reports those intervals I that contain a point $v.key$ which is also contained in Q. Step 4 reports intervals in either case (i) or (iii) and Step 5 reports intervals in either case (ii) or (iii).

Note the special case of **procedure** Overlapping_Interval_Search(v, Q, F) when we set the query interval $Q = [\ell, r]$ so that its left and right endpoints coincide, i.e., $\ell = r$ will report all the intervals in S containing a query point, solving the *Enclosing Interval Searching Problem.*

However, if one is interested in the problem of finding a special type of overlapping intervals, *e.g.*, all intervals containing or contained in a given query interval[11, 15], the interval tree data structure does not necessarily yield an efficient solution. Similarly, the interval tree does not provide an effective method to handle queries about the set of intervals, *e.g.,* the maximum clique, or the *measure*, the total length of the union of the intervals[10, 17].

We conclude with the following theorem.

THEOREM 18.1 *The Enclosing Interval Searching Problem and Overlapping Interval Searching Problem for a set S of n intervals can both be solved in $O(\log n)$ time (plus time for output) and in linear space.*

18.3 Segment Trees

The segment tree structure, originally introduced by Bentley[5, 22], is a data structure for intervals whose endpoints are *fixed* or *known a priori.* The set $S = \{I_1, I_2, \ldots, I_n\}$ of n intervals, each of which represented by $I_i = [\ell_i, r_i]$, $\ell_i, r_i \in \Re$, $\ell_i \leq r_i$, is represented by a data array, Data_Array(S), whose entries correspond to the endpoints, ℓ_i or r_i, and are sorted in non-decreasing order. This sorted array is denoted SA[1..N], $N = 2n$. That is, $SA[1] \leq SA[2] \leq \ldots \leq SA[N]$, $N = 2n$. We will in this section use the indexes in the range $[1, N]$ to refer to the entries in the sorted array SA[1..N]. For convenience we will be working in the transformed domain using indexes, and a comparison involving a point $q \in \Re$ and an index $i \in \aleph$, unless otherwise specified, is performed in the original domain in $\Re$. For instance, $q < i$ is interpreted as $q <$SA[i].

The segment tree structure, as will be demonstrated later, can be useful in finding the *measure* of a set of intervals. That is, the length of the union of a set of intervals. It can also be used to find the maximum clique of a set of intervals. This structure can be generalized to higher dimensions.

18.3.1 Construction of Segment Trees

The segment tree, as the interval tree discussed in Section 18.2 is a rooted augment binary search tree, in which each node v is associated with a range of integers $v.range = [v.B, v.E]$, $v.B, v.E \in \aleph$, $v.B < v.E$, representing a range of indexes from $v.B$ to $v.E$, a key, $v.key$ that split $v.range$ into two subranges, each of which is associated with each child of v, two tree pointers $v.left$ and $v.right$ to the left and right subtrees, respectively, and an auxiliary pointer, $v.aux$ to an augmented data structure. Given integers s and t, with $1 \leq s < t \leq N$, the segment tree, denoted Segment_Tree(s, t), is recursively described as follows.

- The root node v, denoted Segment_Tree_root(s, t), is associated with the range $[s, t]$, and $v.B = s$ and $v.E = t$.
- If $s + 1 = t$ then we have a leaf node v with $v.B = s, v.E = t$ and $v.key =$ nil.
- Otherwise (*i.e., $s + 1 < t$*), let m be the mid-point of s and t, or $m = \lfloor \frac{(v.B+v.E)}{2} \rfloor$. Set $v.key = m$.
- Tree pointer $v.left$ points to the left subtree rooted at Segment_Tree_root(s, m),

and tree pointer $v.right$ points to the right subtree rooted at Segment_Tree_root(m, t).

- Auxiliary pointer $v.aux$ points to an augmented data structure, associated with the range $[s, t]$, whose content depends on the usage of the segment tree.

The following is a pseudo code for the construction of a segment tree for a range $[s, t]$ $s < t, s, t \in \aleph$, and the construction of a set of n intervals whose endpoints are indexed by an array of integers in the range $[1, N], N = 2n$ can be done by a call to Segment_Tree$(1, N)$. See Fig. 18.2(b) for an illustration.

```
function Segment_Tree($s, t$)
/* It returns a pointer $v$ to the root, Segment_Tree_root($s, t$), of the segment tree for the
range [$s, t$].*/
Input: A set $\mathcal{N}$ of integers, $\{s, s+1, \ldots, t\}$ representing the indexes of the endpoints
    of a subset of intervals.
Output: A segment tree, rooted at Segment_Tree_root($s, t$).
Method:
    1. Let $v$ be a node, $v.B = s, v.E = t$, $v.left = v.right = nil$, and $v.aux$ to be
       determined.
    2. if $s + 1 = t$ then return.
    3. Let $v.key = m = \lfloor \frac{(v.B+v.E)}{2} \rfloor$.
    4. $v.left$ = Segment_Tree_root($s, m$)
    5. $v.right$ = Segment_Tree_root($m, t$)
```

The parameters $v.B$ and $v.E$ associated with node v define a range $[v.B, v.E]$, called a *standard range* associated with v. The standard range associated with a leaf node is also called an *elementary range*. It is straightforward to see that Segment_Tree(s, t) constructed in **function** Segment_Tree(s, t) described above is balanced, and has height, denoted Segment_Tree.height, equal to $\lceil \log_2(t - s) \rceil$.

We now introduce the notion of *canonical covering* of a range $[s, t]$, where $s, t \in \aleph$ and $1 \leq s < t \leq N$. A node v in Segment_Tree$(1, N)$ is said to be in the *canonical covering* of $[s, t]$ if its associated standard range satisfies this property $[v.B, v.E] \subseteq [s, t]$, while that of its parent node does not. It is obvious that if a node v is in the canonical covering, then its *sibling node*, *i.e.*, the node with the same parent node as the present one, is not, for otherwise the common parent node would have been in the canonical covering. Thus at each level there are *at most* two nodes that belong to the canonical covering of $[s, t]$.

Thus, for each range $[s, t]$ the number of nodes in its canonical covering is at most $\lceil \log_2(t - s) \rceil + \lfloor \log_2(t - s) \rfloor - 2$. In other words, a range $[s, t]$ (or respectively an interval $[s, t]$) can be decomposed into at most $\lceil \log_2(t - s) \rceil + \lfloor \log_2(t - s) \rfloor - 2$ standard ranges (or respectively subintervals)[5, 22].

To identify the nodes in a segment tree T that are in the canonical covering of an interval $I = [b, e]$, representing a range $[b, e]$, we perform a call to Interval_Insertion(v, b, e, Q), where v is Segment_Tree_root(S). The procedure Interval_Insertion(v, b, e, Q) is defined below.

```
procedure Interval_Insertion($v, b, e, Q$)
/* It returns a queue $Q$ of nodes $q \in T$ such that $[v.B, v.E] \subseteq [b, e]$ and its parent node $u$
whose $[u.B, u.E] \not\subseteq [b, e]$.*/
Input: A segment tree $T$ pointed to by its root node, $v$ = Segment_Tree_root($1, N$),
    for a set $S$ of intervals.
```

Output: A queue Q of nodes in T that are in the canonical covering of $[b, e]$
Method:

```
1. Initialize an output queue $Q$, which supports insertion ($\Rightarrow Q$) and deletion
   ($\Leftarrow Q$) in constant time.
2. if ($[v.B, v.E] \subseteq [b, e]$) then append $[b, e]$ to $v$, $v \Rightarrow Q$, and return.
3. if ($b < v.key$) then Interval_Insertion($v.left, b, e, Q$)
4. if ($v.key < e$) then Interval_Insertion($v.right, b, e, Q$)
```

To **append** $[b, e]$ to a node v means to insert interval $I = [b, e]$ into the auxiliary structure associated with node v to indicate that node v whose standard range is totally contained in I is in the canonical covering of I. If the auxiliary structure $v.aux$ associated with node v is an array, the operation **append** $[b, e]$ to a node v can be implemented as $v.aux[j{+}{+}] = I$,

procedure Interval_Insertion(v, b, e, Q) described above can be used to represent a set S of n intervals in a segment tree by performing the insertion operation n times, one for each interval. As each interval I can have at most $O(\log n)$ nodes in its canonical covering, and hence we perform at most $O(\log n)$ **append** operations for each insertion, the total amount of space required in the auxiliary data structures reflecting all the nodes in the canonical covering is $O(n \log n)$.

Deletion of an interval represented by a range $[b, e]$ can be done similarly, except that the **append** operation will be replaced by its corresponding inverse operation **remove** that removes the node from the list of canonical covering nodes.

THEOREM 18.2 *The segment tree for a set S of n intervals can be constructed in $O(n \log n)$ time, and if the auxiliary structure for each node v contains a list of intervals containing v in the canonical covering, then the space required is $O(n \log n)$.*

18.3.2 Examples and Its Applications

Fig. 18.2(b) illustrates an example of a segment tree of the set of intervals, as shown in Fig. 18.2(a). The integers, if any, under each node v represent the indexes of intervals that contain the node in its canonical covering. For example, Interval I_2 contains nodes labeled by standard ranges $[2, 4]$ and $[4, 7]$.

We now describe how segment trees can be used to solve the *Enclosing Interval Searching Problem* defined before and the *Maximum Clique Problem* of a set of intervals, which is defined below.

Maximum Density or Maximum Clique of a set of Intervals [12, 16, 23] Given a set S of n intervals, find a maximum subset $C \subseteq S$ such that the common intersection of intervals in C is non-empty. That is, $\bigcap_{I_i \in C \subseteq S} I_i \neq \emptyset$ and $|C|$ is maximized. $|C|$ is called the *density* of the set.

The following pseudo code solves the **Enclosing Interval Searching Problem** in $O(\log n)+|F|$ time, where F is the output set. It is invoked by a call Point_in_Interval_Search(v, q, F), where v is Segment_Tree_root(S), and F initially set to be $\emptyset$, will contain the set of intervals containing a query point q.

procedure Point_in_Interval_Search(v, q, F)
/* It returns in F the list of intervals stored in the segment tree pointed to by v and containing query point q */

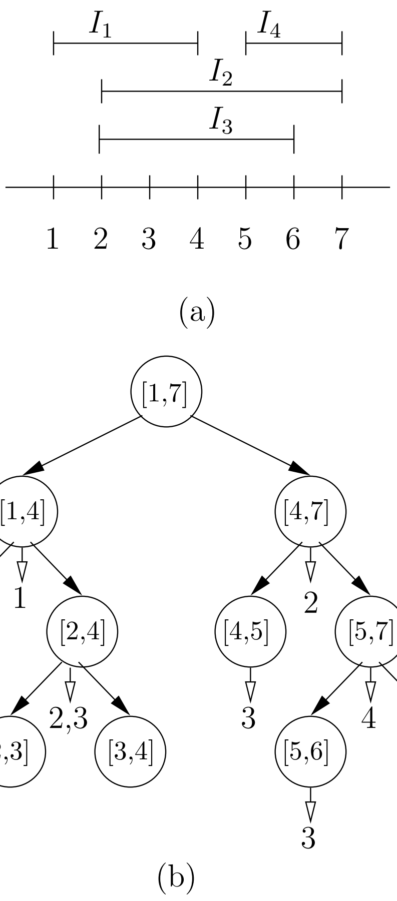


FIGURE 18.2: Segment tree of $S = \{I_1, I_2, I_3, I_4\}$ and its interval models.

Input: A segment tree representing a set S of n intervals, $S = \{I_i | i = 1, 2, \ldots, n\}$ and a query point $q \in \Re$. The auxiliary structure $v.aux$ associated with each node v is a list of intervals $I \in S$ that contain v in their canonical covering.

Output: A subset $F = \{I_i | \ell_i \leq q \leq r_i\}$.

Method:

```
1. if $v$ is nil or ($q < v.B$ or $q > v.E$) then return.
2. if ($v.B \leq q \leq v.E$) then
   for each interval $I_i$ in the auxiliary data structure pointed to by $v.aux$ do
   $F = F \cup \{I_i\}$.
3. if ($q \leq v.key$) then Point_in_Interval_Search($v.left, q, F$)
4. else (/* i.e., $q > v.key$ */) Point_in_Interval_Search($v.right, q, F$)
```

We now address the problem of finding the maximum clique of a set S of intervals, $S = \{I_1, I_2, \ldots, I_n\}$, where each interval $I_i = [\ell_i, r_i]$, and $\ell_i \leq r_i, \ell_i, r_i \in \Re, i = 1, 2, \ldots, n$. There are other approaches, such as plane-sweep [12, 16, 22, 23] that solve this problem within the same complexity.

For this problem we introduce an auxiliary data structure to be stored at each node v. $v.aux$ will contain two pieces of information: one is the *number* of intervals containing v

in the canonical covering, denoted $v.\sharp$, and the other is the *clique size*, denoted $v.clq$. The *clique size* of a node v is the size of the maximum clique whose common intersection is contained in the standard range associated with v. It is defined to be equal to the *larger* of the two numbers: $v.left.\sharp + v.left.clq$ and $v.right.\sharp + v.right.clq$. For a leaf node v, $v.clq = 0$. The size of the maximum clique for the set of intervals will then be stored at the root node Segment_Tree_root(S) and is equal to the sum of $v.\sharp$ and $v.clq$, where $v =$ Segment_Tree_root(S). It is obvious that the space needed for this segment tree is linear.

As this data structure supports insertion of intervals incrementally, it can be used to answer the maximum clique of the current set of intervals as the intervals are inserted into (or deleted from) the segment tree T. The following pseudo code finds the size of the maximum clique of a set of intervals.

```
function Maximum_Clique($S$)
/* It returns the size of the maximum clique of a set $S$ of intervals. */
Input: A set $S$ of $n$ intervals and the segment tree $T$ rooted at Segment_Tree_root($S$).
Output: An integer, which is the size of the maximum clique of $S$.
Method: Assume that $S = \{I_1, I_2, \ldots, I_n\}$ and that the endpoints of the intervals are
        represented by the indexes of a sorted array containing these endpoints.
     1. Initialize $v.clq = v.\sharp = 0$ for all nodes $v$ in $T$.
     2. for each interval $I_i = [\ell_i, r_i] \in S$, $i = 1, 2, \ldots, n$ do
        /* Insert $I_i$ into the tree and update $v.\sharp$ and $v.clq$ for all visited nodes and
        those nodes in the canonical covering of $I_i$ */
     3. begin
     4. $s$ = Find_split-node($v, \ell_i, r_i$), where $v$ is Segment_Tree_root($S$). (See below)
        Let the root-to-split-node($s$)-path be denoted $P$.
     5. /* Find all the canonical covering nodes in the left subtree of $s$ */
        Traverse along the left subtree from $s$ following the left tree pointer, and
        find a leftward path, $s_{\ell_1}, s_{\ell_2}, \ldots$ till node $s_{\ell_L}$ such that $s_{\ell_1} = s.left$, $s_{\ell_k} =$
        $s_{\ell_{k-1}}.left$, for $k = 2, \ldots, L$. Note that the standard ranges of all these nodes
        overlap $I_i$, but the standard range associated with $s_{\ell_L}.left$ is totally disjoint
        from $I_i$. $s_{\ell_L}$ is $s_{\ell_1}$ only if the standard range of $s_{\ell_1}$ is totally contained in
        $I_i$, i.e., $s_{\ell_1}$ is in the canonical covering of $I_i$. Other than this, the right child
        of each node on this leftward path belongs to the canonical covering of $I_i$.
     6. Increment $u.\sharp$ for all nodes $u$ that belong to the canonical covering of $I_i$.
     7. Update $s_{\ell_j}.clq$ according to the definition of clique size for all nodes on the
        leftward path in reverse order, i.e., starting from node $s_L$ to $s_{\ell_1}$.
     8. /* Find all the canonical covering nodes in the right subtree of $s$ */
        Similarly we traverse along the right subtree from $s$ along the right tree
        pointer, and find a rightward path. Perform Steps 5 to 7.
     9. Update $s.clq$ for the split node $s$ after the clique sizes of both left and right
        child of node $s$ have been updated.
    10. Update $u.clq$ for all the nodes $u$ on the root-to-split-node-path $P$ in reverse
        order, starting from node $s$ to the root.
    11. end
    12. return $(v.\sharp + v.clq)$, where $v$ = Segment_Tree_root($S$).
```

function Find_split-node(v, b, e)
/* Given a segment tree T rooted at v and an interval $I = [b, e] \subseteq [v.B, v.E]$, this procedure returns the *split-node* s such that either $[s.B, s.E] = [b, e]$ or $[s_\ell.B, s_\ell.E] \cap [b, e] \neq \emptyset$ and $[s_r.B, s_r.E] \cap [b, e] \neq \emptyset$, where s_ℓ and s_r are the left child and right child of s respectively. */

```
1. if $[v.B, v.E] = [b, e]$ then return $v$.
2. if $(b < v.key)$ and $(e > v.key)$ then return $v$.
3. if $(e \leq v.key)$ then return Find_split-node$(v.left, b, e)$
4. if $(b \geq v.key)$ then return Find_split-node$(v.right, b, e)$
```

Note that in **procedure** Maximum_Clique(S) it takes $O(\log n)$ time to process each interval. We conclude with the following theorem.

THEOREM 18.3 *Given a set $S = \{I_1, I_2, \ldots, I_n\}$ of n intervals, the maximum clique of $S_i = \{I_1, I_2, \ldots, I_i\}$ can be found in $O(i \log i)$ time and linear space, for each $i = 1, 2, \ldots, n$, by using a segment tree.*

We note that the above procedure can be adapted to find the maximum clique of a set of *hyperrectangles* in k-dimensions for $k > 2$ in time $O(n^k)$.[16]

18.4 Range Trees

Consider a set S of points in k-dimensional space $\Re^k$. A range tree for this set S of points is a data structure that supports general range queries of the form $[x^1_\ell, x^1_r] \times [x^2_\ell, x^2_r] \times \ldots \times [x^k_\ell, x^k_r]$, where each range $[x^i_\ell, x^i_r], x^i_\ell, x^i_r \in \Re, x^i_\ell \leq x^i_r$ for all $i = 1, 2, \ldots, k$, denotes an interval in $\Re$. The cartesian product of these k ranges is referred to as a *kD* range. In 2-dimensional space, a 2D range is simply an axes-parallel rectangle in $\Re^2$. The *range search problem* is to find all the points in S that satisfy any range query. In 1-dimension, the range search problem can be easily solved in logarithmic time using a sorted array or a balanced binary search tree. The 1D range is simply an interval $[x_\ell, x_r]$. We first do a binary search using x_ℓ as searched key to find the first node v whose key is no less than x_ℓ. Once v is located, the rest is simply to retrieve the nodes, one at a time, until the node u whose key is greater than x_r. We shall in this section describe an augmented binary search tree which is easily generalized to higher dimensions.

18.4.1 Construction of Range Trees

A range tree is primarily a binary search tree augmented with an auxiliary data structure. The root node v, denoted Range_Tree_root(S), of a kD-range tree[5, 18, 22, 24] for a set S of points in k-dimensional space $\Re^k$, *i.e.,* $S = \{p_i = (x^1_i, x^2_i, \ldots, x^k_i), i = 1, 2, \ldots, n\}$, where $p_i.x^j = x^j_i \in \Re$ is the jth-coordinate value of point p_i, for $j = 1, 2, \ldots, k$, is associated with the entire set S. The key stored in $v.key$ is to partition S into two approximately equal subsets S_ℓ and S_r, such that all the points in S_ℓ and in S_r lie to the left and to the right, respectively of the hyperplane $H^k : x^k = v.key$. That is, we will store the median of the kth coordinate values of all the points in S in $v.key$ of the root node v, *i.e.,* $v.key = p_j.x^k$ for some point p_j such that S_ℓ contains points p_ℓ, $p_\ell.x^k \leq v.key$, and S_r contains points p_r, $p_r.x^k > v.key$. Each node v in the kD-range tree, as before, has two tree pointers, $v.left$ and $v.right$, to the roots of its left and right subtrees respectively. The node pointed to by *v.left* will be associated with the set S_ℓ and the node pointed to by *v.right* will be associated with the set S_r. The auxiliary pointer *v.aux* will point to an augmented data structure, in our case a $(k-1)$D-range tree.

A $1D$-range tree is a sorted array of all the points $p_i \in S$ such that the entries are drawn from the set $\{x_i^1 | i = 1, 2, \ldots, n\}$ sorted in nondecreasing order. This 1D-range tree supports the 1D range search in logarithmic time.

The following is a pseudo code for a kD-range tree for a set S of n points in k-dimensional space. See Fig. 18.3(a) and (b) for an illustration. Fig. 18.4(c) is a schematic representation of a kD-range tree.

```
function kD_Range_Tree($k, S$)
/* It returns a pointer $v$ to the root, kD_Range_Tree_root($k, S$), of the kD-range tree for a
set $S \subseteq \Re^k$ of points, $k \geq 1$. */
Input: A set $S$ of $n$ points in $\Re^k$, $S = \{p_i = (x_i^1, x_i^2, \ldots, x_i^k), i = 1, 2, \ldots, n\}$, where
    $x_i^j \in \Re$ is the $j$th-coordinate value of point $p_i$, for $j = 1, 2, \ldots, k$.
Output: A kD-range tree, rooted at kD_Range_Tree_root($k, S$).
Method:
    1. if $S = \emptyset$, return nil.
    2. if ($k = 1$) create a sorted array SA($S$) pointed to by a node $v$ containing
       the set of the 1st coordinate values of all the points in $S$, i.e., SA($1, S$) has
       $\{p_i.x^1 | i = 1, 2, \ldots, n\}$ in nondecreasing order. return ($v$).
    3. Create a node $v$ such that $v.key$ equals the median of the set $\{p_i.x^k|$ kth
       coordinate value of $p_i \in S, i = 1, 2, \ldots, n\}$. Let $S_\ell$ and $S_r$ denote the
       subset of points whose kth coordinate values are not greater than and are
       greater than $v.key$ respectively. That is, $S_\ell = \{p_i \in S\} | p_i.x^k \leq v.key\}$ and
       $S_r = \{p_j \in S\} | p_j.x^k > v.key\}$.
    4. $v.left$ = kD_Range_Tree($k, S_\ell$)
    5. $v.right$ = kD_Range_Tree($k, S_r$)
    6. $v.aux$ = kD_Range_Tree($k-1, S$)
```

As this is a recursive algorithm with two parameters, k and $|S|$, that determine the recursion depth, it is not immediately obvious how much time and how much space are needed to construct a kD-range tree for a set of n points in k-dimensional space.

Let $T(n, k)$ denote the time taken and $S(n, k)$ denote the space required to build a kD-range tree of a set of n points in $\Re^k$. The following are recurrence relations for $T(n, k)$ and $S(n, k)$ respectively.

$$T(n,k) = \begin{cases} O(1) & \text{if } n = 1 \\ O(n \log n) & \text{if } k = 2 \\ 2T(n/2, k) + T(n, k-1) + O(n) & \text{otherwise} \end{cases}$$

$$S(n,k) = \begin{cases} O(1) & \text{if } n = 1 \\ O(n) & \text{if } k = 1 \\ 2S(n/2, k) + S(n, k-1) + O(1) & \text{otherwise} \end{cases}$$

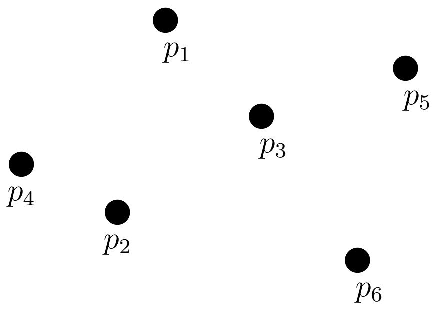
Note that in 1-dimension, we need to have the points sorted and stored in a sorted array, and thus $T(n, 1) = O(n \log n)$ and $S(n, 1) = O(n)$. The solutions of $T(n, k)$ and $S(n, k)$ to the above recurrence relations are $T(n, k) = O(n \log^{k-1} n + n \log n)$ for $k \geq 1$ and $S(n, k) = O(n \log^{k-1} n)$ for $k \geq 1$. For a general multidimensional divide-and-conquer scheme, and solutions to the recurrence relation, please refer to Bentley[2] and Monier[20] respectively.

We conclude that

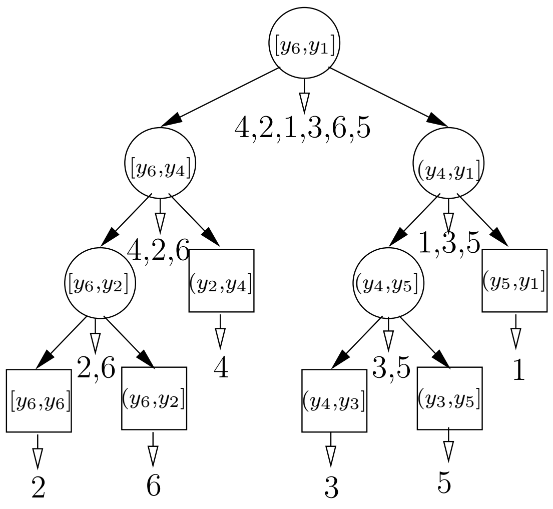
THEOREM 18.4 *The kD-range tree for a set of n points in k-dimensional space can be constructed in $O(n \log^{k-1} n + n \log n)$ time and $O(n \log^{k-1} n)$ space for $k \geq 1$.*

18.4.2 Examples and Its Applications

Fig. 18.3(b) illustrates an example of a range tree for a set of points in 2-dimensions shown in Fig. 18.3(a). This list of integers under each node represents the indexes of points in ascending x-coordinates. Fig. 18.4 illustrates a general schematic representation of a kD-range tree, which is a *layered* structure[5, 22].



(a)



(b)

FIGURE 18.3: 2D-range tree of $S = \{p_1, p_2, \ldots, p_6\}$, where $p_i = (x_i, y_i)$.

We now discuss how we make use of a range tree to solve the range search problem. We shall use 2D-range tree as an example for illustration purposes. It is rather obvious

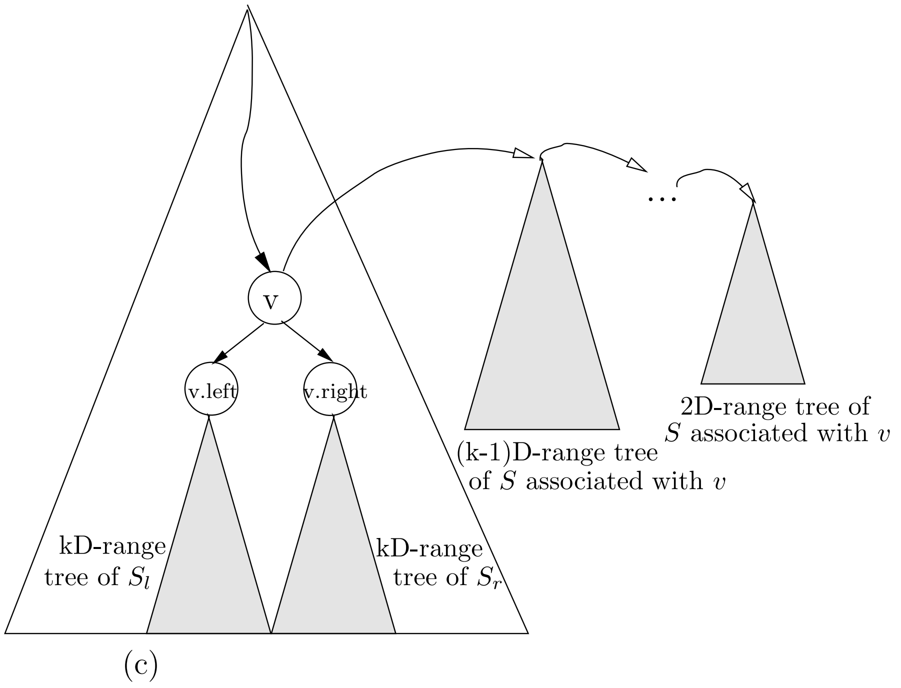

v
v.left
v.right
kD-range tree of S_l
kD-range tree of S_r
(k-1)D-range tree of S associated with v
...
2D-range tree of S associated with v
(c)

FIGURE 18.4: A schematic representation of a (layered) kD-range tree, where S is the set associated with node v.

to generalize it to higher dimensions. Recall we have a set S of n points in the plane $\Re^2$ and 2D range query $Q = [x_\ell, x_r] \times [y_\ell, y_r]$. Let us assume that a 2D-range tree rooted at 2D_Range_Tree_root(S) is available. Recall also that associated with each node v in the range tree there is a *standard range* for the set S_v of points represented in the subtree rooted at node v, in this case $[v.B, v.E]$ where $v.B = \min\{p_i.y\}$ and $v.E = \max\{p_i.y\}$ for all $p_i \in S_v$. $v.key$ will split the standard range into two standard subranges $[v.B, v.key]$ and $[v.key, v.E]$ each associated with the root nodes $v.left$ and $v.right$ of the left and right subtrees of v respectively.

The following pseudo code reports in F the set of points in S that lie in the range $Q = [x_\ell, x_r] \times [y_\ell, y_r]$. It is invoked by 2D_Range_Search($v, x_\ell, x_r, y_\ell, y_r, F$), where v is the root, 2D_Range_Tree_root(S), and F, initially empty will return all the points in S that lie in the range $Q = [x_\ell, x_r] \times [y_\ell, y_r]$.

```
**procedure** 2D_Range_Search($v, x_\ell, x_r, y_\ell, y_r, F$)
/* It returns $F$ containing all the points in the range tree rooted at node $v$ that lie in
$[x_\ell, x_r] \times [y_\ell, y_r]$. */
**Input:** A set $S$ of $n$ points in $\Re^2$, $S = \{p_i | i = 1, 2, \ldots, n\}$ and each point $p_i =$
        $(p_i.x, p_i.y)$, where $p_i.x$ and $p_i.y$ are the $x$- and $y$-coordinates of $p_i$, $p_i.x, p_i.y \in$
        $\Re, i = 1, 2, \ldots, n$.
**Output:** A set $F$ of points in $S$ that lie in $[x_\ell, x_r] \times [y_\ell, y_r]$.
**Method:**
        1. Start from the root node $v$ to find the split-node $s$, $s$ = Find_split-node($v, y_\ell, y_r$),
           such that $s.key$ lies in $[y_\ell, y_r]$.
```

```
2. if $s$ is a leaf, then 1D_Range_Search($s.aux, x_\ell, x_r, F$) that checks in the
   sorted array pointed to by $s.aux$, which contains just a point $p$, to see if its
   $x$-coordinate $p.x$ lies in the $x$-range $[x_\ell, x_r]$
3. $v = s.left$.
4. while $v$ is not a leaf do
   if ($y_\ell \leq v.key$) then
       1D_Range_Search($v.right.aux, x_\ell, x_r, F$)
       $v = v.left$
   else $v = v.right$
5. (/* $v$ is a leaf, and check node $v.aux$ directly */)
   1D_Range_Search($v.aux, x_\ell, x_r, F$)
6. $v = s.right$
7. while $v$ is not a leaf do
   if ($y_r > v.key$) then
       1D_Range_Search($v.left.aux, x_\ell, x_r, F$)
       $v = v.right$
   else $v = v.left$
8. (/* $v$ is a leaf, and check node $v.aux$ directly */)
   1D_Range_Search($v.aux, x_\ell, x_r, F$)
```

procedure 1D_Range_Search(v, x_ℓ, x_r, F) is very straightforward. v is a pointer to a sorted array SA. We first do a binary search in SA looking for the first element no less than x_ℓ and then start to report in F those elements no greater than x_r. It is obvious that **procedure** 2D_Range_Search finds all the points in Q in $O(\log^2 n)$ time. Note that there are $O(\log n)$ nodes for which we need to invoke 1D_Range_Search in their auxiliary sorted arrays. These nodes v are in the canonical covering[‡] of the y-range $[y_\ell, y_r]$, since its associated standard range $[v.B, v.E]$ is totally contained in $[y_\ell, y_r]$, and the 2D-range search problem is now reduced to the 1D-range search problem.

This is not difficult to see that the 2D-range search problem can be answered in time $O(\log^2 n)$ plus time for output, as there are $O(\log n)$ nodes in the canonical covering of a given y-range and for each node in the canonical covering we spend $O(\log n)$ time for dealing with the 1D-range search problem.

However, with a modification to the auxiliary data structure, one can achieve an optimal query time of $O(\log n)$, instead of $O(\log^2 n)$[6, 7, 24]. This is based on the observation that in each of the 1D-range search subproblem associated with each node in the canonical covering, we perform the same query, reporting points whose x-coordinates lie in the x-range $[x_\ell, x_r]$. More specifically we are searching for the smallest element no less than x_ℓ.

The modification is performed on the sorted array associated with each of the node in the 2D_Range_Tree(S).

Consider the root node v. As it is associated with the entire set of points, $v.aux$ points to the sorted array containing the x-coordinates of *all* the points in S. Let this sorted array be denoted $SA(v)$ and the entries, $SA(v)_i, i = 1, 2, \ldots, |S|$, are sorted in nondecreasing order of the x-coordinate values. In addition to the x-coordinate value, each entry also contains

[‡]See the definition of the canonical covering defined in Section 18.3.1.

the index of the corresponding point. That is, $SA(v)_i.key$ and $SA(v)_i.index$ contain the x-coordinate of p_j respectively, where $SA(v)_i.index = j$ and $SA(v)_i.key = p_j.x$.

We shall augment each entry $SA(v)_i$ with two pointers, $SA(v)_i.left$ and $SA(v)_i.right$. They are defined as follows. Let v_ℓ and v_r denote the roots of the left and right subtrees of v, *i.e.,* $v.left = v_\ell$ and $v.right = v_r$. $SA(v)_i.left$ points to the entry $SA(v_\ell)_j$ such that entry $SA(v_\ell)_j.key$ is the smallest among all key values $SA(v_\ell)_j.key \geq SA(v)_i.key$. Similarly, $SA(v)_i.right$ points to the entry $SA(v_r)_k$ such that entry $SA(v_r)_k.key$ is the smallest among all key values $SA(v_r)_k.key \geq SA(v)_i.key$.

These two augmented pointers, $SA(v)_i.left$ and $SA(v)_i.right$, possess the following property: If $SA(v)_i.key$ is the smallest key such that $SA(v)_i.key \geq x_\ell$, then $SA(v_\ell)_j.key$ is also the smallest key such that $SA(v_\ell)_j.key \geq x_\ell$. Similarly $SA(v_r)_k.key$ is the smallest key such that $SA(v_r)_k.key \geq x_\ell$.

Thus if we have performed a binary search in the auxiliary sorted array $SA(v)$ associated with node v locating the entry $SA(v)_i$ whose key $SA(v)_i.key$ is the smallest key such that $SA(v)_i.key \geq x_\ell$, then following the left (respectively right) pointer $SA(v)_i.left$ (respectively $SA(v)_i.right$) to $SA(v_\ell)_j$ (respectively $SA(v_r)_k$), the entry $SA(v_\ell)_j.key$ (respectively $SA(v_r)_k.key$) is also the smallest key such that $SA(v_\ell)_j.key \geq x_\ell$ (respectively $SA(v_r)_k.key \geq x_\ell$). Thus there is no need to perform an additional binary search in the auxiliary sorted array $SA(v.left)$ (respectively $SA(v.right)$).

With this additional modification, we obtain an *augmented* 2D-range tree and the following theorem.

THEOREM 18.5 *The 2D-range search problem for a set of n points in the 2-dimensional space can be solved in time $O(\log n)$ plus time for output, using an augmented 2D-range tree that requires $O(n \log n)$ space.*

The following procedure is generalized from **procedure** 2D_Range_Search$(v, x_\ell, x_r, y_\ell, y_r, F)$ discussed in Section 18.4.2 taken into account the augmented auxiliary data structure. It is invoked by kD_Range_Search(k, v, Q, F), where v is the root kD_Range_Tree_root(S) of the range tree, Q is the k-range, $[x_\ell^1, x_r^1] \times [x_\ell^2, x_r^2] \times \ldots \times [x_\ell^k, x_r^k]$, represented by a two dimensional array, such that $Q_i.\ell = x_\ell^i$ and $Q_i.r = x_r^i$, and F, initially empty, will contain all the points that lie in Q.

procedure kD_Range_Search(k, v, Q, F). /* It returns F containing all the points in the range tree rooted at node v that lie in k-range, $[x_\ell^1, x_r^1] \times [x_\ell^2, x_r^2] \times \ldots \times [x_\ell^k, x_r^k]$, where each range $[x_\ell^i, x_r^i], x_\ell^i = Q_i.\ell, x_r^i = Q_i.r \in \Re, x_\ell^i \leq x_r^i$ for all $i = 1, 2, \ldots, k$, denotes an interval in $\Re$. */

Input: A set S of n points in $\Re^k$, $S = \{p_i | i = 1, 2, \ldots, n\}$ and each point $p_i = (p_i.x^1, p_i.x^2, \ldots, p_i.x^k)$, where $p_i.x^j \in \Re$, are the jth-coordinates of p_i, $j = 1, 2, \ldots, k$.

Output: A set F of points in S that lie in $[x_\ell^1, x_r^1] \times [x_\ell^2, x_r^2] \times \ldots \times [x_\ell^k, x_r^k]$.

Method:

```
1. if ($k > 2$) then
      • Start from the root node $v$ to find the split-node $s$, $s$ = Find_split-node($v, Q_\ell^k, Q_r^k$), such that $s.key$ lies in $[Q_\ell^k, Q_r^k]$.
      • if $s$ is a leaf, then check in the sorted array pointed to by $s.aux$, which contains just a point $p$. $p \Rightarrow F$ if its coordinate values lie in $Q$. return
      • $v = s.left$.
```

```
    • while v is not a leaf do
      if ($Q^k_\ell \leq v.key$)
      then kD_Range_Search($k-1, v.right.aux, Q, F$).
            $v = v.left$
      else $v = v.right$
    • (/* v is a leaf, and check node v.aux directly */)
      Check in the sorted array pointed to by v.aux, which contains just a
      point p. $p \Rightarrow F$ if its coordinate values lie in Q. return
    • $v = s.right$
    • while v is not a leaf do
      if ($Q^k_r > v.key$)
      then kD_Range_Search($k-1, v.left.aux, Q, F$).
            $v = v.right$
      else $v = v.left$
    • (/* v is a leaf, and check node v.aux directly */)
      Check in the sorted array pointed to by v.aux, which contains just a
      point p. $p \Rightarrow F$ if its coordinate values lie in Q. return
2. else /* $k \leq 2$*/
3. if $k = 2$ then
    • Do binary search in sorted array SA(v) associated with node v, using
      $Q_1.\ell$ ($x^1_\ell$) as key to find entry $o_v$ such that $SA(v)_{o_v}$'s key, $SA(v)_{o_v}.key$
      is the smallest such that $SA(v)_{o_v}.key \geq Q_1.\ell$,
    • Find the split-node s, $s$ = Find_split-node($v, x^2_\ell, x^2_r$), such that s.key lies
      in $[x^2_\ell, x^2_r]$. Record the root-to-split-node path from v to s, following left
      or right tree pointers.
    • Starting from entry $o_v$ ($SA(v)_i$) follow pointers $SA(v)_{o_v}.left$ or $SA(v)_{o_v}.right$
      according to the v-to-s path to point to entry $SA(s)_{o_s}$ associated with
      SA(s).
    • if s is a leaf, then check in the sorted array pointed to by s.aux, which
      contains just a point p. $p \Rightarrow F$ if its coordinate values lie in Q. return
    • $v = s.left$, $o_v = SA(s)_{o_s}.left$.
    • while v is not a leaf do
          if ($Q_2.\ell \leq v.key$)
          then $\ell = SA(v)_{o_v}.right$
              while ($SA(v.right)_\ell.key \leq Q_1.r$) do
                  point $p_m \Rightarrow F$, where $m = SA(v.right)_\ell.index$
                  $\ell$++
              $v = v.left$, $o_v = SA(v)_{o_v}.left$
          else $v = v.right$, $o_v = SA(v)_{o_v}.right$
    • (/* v is a leaf, and check node v.aux directly */)
      Check in the sorted array pointed to by v.aux, which contains just a
      point p. $p \Rightarrow F$ if its coordinate values lie in Q.
    • $v = s.right$, $o_v = SA(s)_{o_s}.right$.
    • while v is not a leaf do
          if ($Q_2.r > v.key$)
          then $\ell = SA(v)_{o_v}.left$
```

```
            while ($SA(v.left)_\ell.key \leq Q_1.r$) do
                point $p_m \Rightarrow F$, where $m = SA(v.left)_\ell.index$
                $\ell$++
        else $v = v.left$, $o_v = SA(v)_{o_v}.left$
```

- (/* v is a leaf, and check node $v.aux$ directly */)
  Check in the sorted array pointed to by $v.aux$, which contains just a point p. $p \Rightarrow F$ if its coordinate values lie in Q.

The following recurrence relation for the query time $Q(n,k)$ of the kD-range search problem, can be easily obtained:

$$Q(n,k) = \begin{cases} O(1) & \text{if } n = 1 \\ O(\log n) + \mathcal{F} & \text{if } k = 2 \\ \Sigma_{v \in CC} Q(n_v, k-1) + O(\log n) & \text{otherwise} \end{cases}$$

where $\mathcal{F}$ denotes the output size, and n_v denotes the size of the subtree rooted at node v that belongs to the canonical covering CC of the query. The solution is $Q(n,k) = O(\log^{k-1} n) + \mathcal{F}$[5, 22].

We conclude with the following theorem.

THEOREM 18.6 *The kD-range search problem for a set of n points in the k-dimensional space can be solved in time $O(\log^{k-1} n)$ plus time for output, using an augmented kD-range tree that requires $O(n \log^{k-1} n)$ space for $k \geq 1$.*

18.5 Priority Search Trees

The priority search tree was originally introduced by McCreight[19]. It is a hybrid of two data structures, binary search tree and a priority queue.[13] A *priority queue* is a queue and supports the following operations: insertion of an item and deletion of the minimum (highest priority) item, so called *delete_min* operation. Normally the delete_min operation takes constant time, while updating the queue so that the minimum element is readily accessible takes logarithmic time. However, searching for an element in a priority queue will normally take linear time. To support efficient searching, the priority queue is modified to be a priority search tree. We will give a formal definition and its construction later. As the priority search tree represents a set S of elements, each of which has two pieces of information, one being a key from a totally ordered set, say the set $\Re$ of real numbers, and the other being a notion of priority, also from a totally ordered set, for each element, we can model this set S as a set of points in 2-dimensional space. The x- and y-coordinates of a point p represent the key and the priority respectively. For instance, consider a set of jobs $S = \{J_1, J_2, \ldots, J_n\}$, each of which has a release time $r_i \in \Re$ and a priority $p_i \in \Re, i = 1, 2, \ldots, n$. Then each job J_i can be represented as a point q such that $q.x = r_i, q.y = p_i$.

The priority search tree can be used to support queries of the form, find, among a set S of n points, the point p with minimum $p.y$ such that its x-coordinate lies in a given range $[\ell, r]$, *i.e.,* $\ell \leq p.x \leq r$. As can be shown later, this query can be answered in $O(\log n)$ time.

18.5.1 Construction of Priority Search Trees

As before, the root node, Priority_Search_Tree_root(S), represents the entire set S of points. Each node v in the tree will have a key $v.key$, an auxiliary data $v.aux$ containing the index of the point and its priority, and two pointers $v.left$ and $v.right$ to its left and right subtrees respectively such that all the key values stored in the left subtree are less than $v.key$, and all the key values stored in the right subtree are greater than $v.key$. The following is a pseudo code for the recursive construction of the priority search tree of a set S of n points in $\Re^2$. See Fig. 18.5(a) for an illustration.

```
function Priority_Search_Tree($S$)
/* It returns a pointer $v$ to the root, Priority_Search_Tree_root($S$), of the priority search
tree for a set $S$ of points. */
Input: A set $S$ of $n$ points in $\Re^2$, $S = \{p_i | i = 1, 2, \ldots, n\}$ and each point $p_i =$
    $(p_i.x, p_i.y)$, where $p_i.x$ and $p_i.y$ are the $x$- and $y$-coordinates of $p_i$, $p_i.x, p_i.y \in$
    $\Re, i = 1, 2, \ldots, n$.
Output: A priority search tree, rooted at Priority_Search_Tree_root($S$).
Method:
        1. if $S = \emptyset$, return nil.
        2. Create a node $v$ such that $v.key$ equals the median of the set $\{p.x | p \in S\}$, and
           $v.aux$ contains the index $i$ of the point $p_i$ whose $y$-coordinate is the minimum
           among all the $y$-coordinates of the set $S$ of points i.e., $p_i.y = \min\{p.y | p \in S\}$.
        3. Let $S_\ell = \{p \in S \setminus \{p_{v.aux}\} | p.x \leq v.key\}$ and $S_r = \{p \in S \setminus \{p_{v.aux}\} | p.x >$
           $v.key\}$ denote the set of points whose $x$-coordinates are less than or equal
           to $v.key$ and greater than $v.key$ respectively.
        4. $v.left$= Priority_Search_Tree_root($S_\ell$).
        5. $v.right$= Priority_Search_Tree_root($S_r$).
        6. return $v$.
```

Thus, Priority_Search_Tree_root(S) is a minimum heap data structure with respect to the y-coordinates, *i.e.,* the point with minimum y-coordinate can be accessed in constant time, and is a balanced binary search tree for the x-coordinates. Implicitly the root node v is associated with an x-range $[x_\ell, x_r]$ representing the span of the x-coordinate values of all the points in the whole set S. The root of the left subtree pointed to by $v.left$ is associated with the x-range $[x_\ell, v.key]$ representing the span of the x-coordinate values of all the points in the set S_ℓ and the root of the right subtree pointed to by $v.right$ is associated with the x-range $[v.key, x_r]$ representing the span of the x-coordinate values of all the points in the set S_r. It is obvious that this algorithm takes $O(n \log n)$ time and linear space. We summarize this in the following.

THEOREM 18.7 *The priority search tree for a set S of n points in $\Re^2$ can be constructed in $O(n \log n)$ time and linear space.*

18.5.2 Examples and Its Applications

Fig. 18.5 illustrates an example of a priority search tree of the set of points. Note that the root node contains p_6 since its y-coordinate value is the minimum.

We now illustrate a usage of the priority search tree by an example. Consider a so-called *grounded 2D range search problem* for a set of n points in the plane. As defined in Section 18.4.2, a 2D range search problem is to find all the points $p \in S$ such that $p.x$ lies in an x-range $[x_\ell, x_r], x_\ell \leq x_r$ and $p.y$ lies in a y-range $[y_\ell, y_r]$. When the y-range is of the

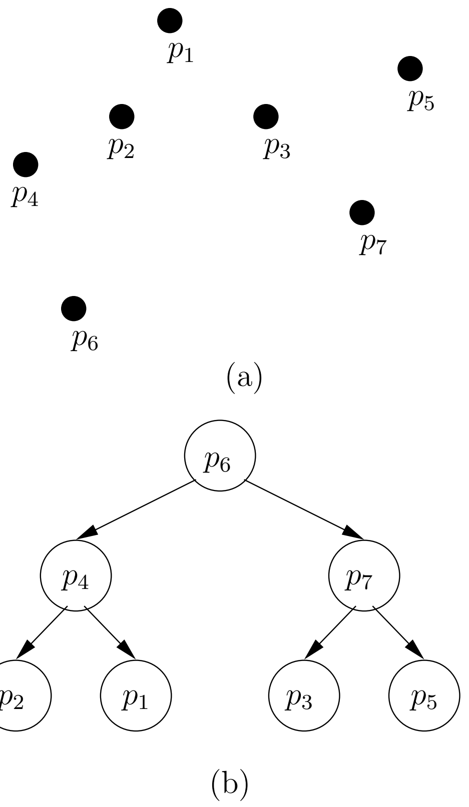


FIGURE 18.5: Priority search tree of $S = \{p_1, p_2, \ldots, p_7\}$.

form $[-\infty, y_r]$ then the 2D range is referred to as *grounded* 2D range or sometimes as *1.5D* range, and the 2D range search problem as *grounded* 2D range search or *1.5D* range search problem.

Grounded 2D Range Search Problem Given a set S of n points in the plane $\Re^2$, with preprocessing allowed, find the subset F of points whose x- and y-coordinates satisfy a grounded 2D range query of the form $[x_\ell, x_r] \times [-\infty, y_r], x_\ell, x_r, y_r \in \Re, x_\ell \leq x_r$.

The following pseudo code solves this problem optimally. We assume that a priority search tree for S has been constructed via procedure Priority_Search_Tree(S). The answer will be obtained in F via an invocation to Priority_Search_Tree_Range_Search(v, x_ℓ, x_r, y_r, F), where v is Priority_Search_Tree_root(S).

procedure Priority_Search_Tree_Range_Search(v, x_ℓ, x_r, y_r, F)
/* v points to the root of the tree, F is a queue and set to nil initially. */

Input: A set S of n points, $\{p_1, p_2, \ldots, p_n\}$, in $\Re^2$, stored in a priority search tree, Priority_Search_Tree(S) pointed to by Priority_Search_Tree_root(S) and a 2D grounded range $[x_\ell, x_r] \times [-\infty, y_r], x_\ell, x_r, y_r \in \Re, x_\ell \leq x_r$.

Output: A subset $F \subseteq S$ of points that lie in the 2D grounded range, *i.e.*, $F = \{p \in S | x_\ell \leq p.x \leq x_r$ and $p.y \leq y_r\}$.

Method:

```
1. Start from the root $v$ finding the first split-node $v_{split}$ such that $v_{split}.x$ lies
   in the $x$-range $[x_\ell, x_r]$.
2. For each node $u$ on the path from node $v$ to node $v_{split}$ **if** the point $p_{u.aux}$
   lies in range $[x_\ell, x_r] \times [\infty, y_r]$ **then** report it by $(p_{u.aux} \Rightarrow F)$.
3. For each node $u$ on the path of $x_\ell$ in the left subtree of $v_{split}$ **do**
   **if** the path goes left at $u$ **then** Priority_Search_Tree_1dRange_Search($u.right, y_r, F$).
4. For each node $u$ on the path of $x_r$ in the right subtree of $v_{split}$ **do**
   **if** the path goes right at $u$ **then** Priority_Search_Tree_1dRange_Search($u.left, y_r, F$).
```

```
**procedure** Priority_Search_Tree_1dRange_Search($v, y_r, F$)
/* Report in $F$ all the points $p_i$, whose $y$-coordinate values are no greater than $y_r$, where
$i = v.aux$. */
1. **if** $v$ is nil **return**.
2. **if** $p_{v.aux}.y \leq y_r$ **then** report it by $(p_{v.aux} \Rightarrow F)$.
3.        Priority_Search_Tree_1dRange_Search($v.left, y_r, F$)
4.        Priority_Search_Tree_1dRange_Search($v.right, y_r, F$)
```

procedure Priority_Search_Tree_1dRange_Search(v, y_r, F) basically retrieves all the points stored in the priority search tree rooted at v such that their y-coordinates are all less than and equal to y_r. The search terminates at the node u whose associated point has a y-coordinate greater than y_r, implying **all** the nodes in the subtree rooted at u satisfy this property. The amount of time required is proportional to the output size. Thus we conclude that

THEOREM 18.8 *The* **Grounded 2D Range Search Problem** *for a set S of n points in the plane $\Re^2$ can be solved in time $O(\log n)$ plus time for output, with a priority search tree structure for S that requires $O(n \log n)$ time and $O(n)$ space.*

Note that the space requirement for the priority search tree in linear, compared to that of a 2D-range tree, which requires $O(n \log n)$ space. That is, the **Grounded 2D Range Search Problem** for a set S of n points can be solved optimally using priority search tree structure.

Acknowledgment

This work was supported in part by the National Science Council under the Grants NSC-91-2219-E-001-001 and NSC91-2219-E-001-002.

References

[1] J. L. Bentley, "Decomposable searching problems," *Inform. Process. Lett.*, vol. 8, 1979, pp. 244- 251.

[2] J. L. Bentley, "Multidimensional divide-and-conquer," *Commun. ACM*, (23,4), 1980, pp. 214-229.

[3] J. L. Bentley and J. B. Saxe, "Decomposable searching problems I: Static-to-dynamic transformation," *J. Algorithms*, vol. 1, 1980, pp. 301-358.

[4] J.-D. Boissonnat and F. P. Preparata, "Robust plane sweep for intersecting segments," *SIAM J. Comput.* (29,5), 2000, pp. 1401-1421.

[5] M. de Berg, M. van Kreveld, M. Overmars and O. Schwarzkopf, *Computational Geometry: Algorithms and Applications*, Springer-Verlag, Berlin, 1997.

[6] B. Chazelle and L. J. Guibas, "Fractional cascading: I. A data structuring technique," *Algorithmica*, (1,3), 1986, pp. 133–162

[7] B. Chazelle and L. J. Guibas, "Fractional cascading: II. Applications," *Algorithmica*, (1,3), 1986, pp. 163-191.

[8] H. Edelsbrunner, "A new approach to rectangle intersections, Part I," *Int'l J. Comput. Math.*, vol. 13, 1983, pp. 209-219.

[9] H. Edelsbrunner, "A new approach to rectangle intersections, Part II," *Int'l J. Comput. Math.*, vol. 13, 1983, pp. 221-229.

[10] M. L. Fredman and B. Weide, "On the complexity of computing the measure of $\bigcup[a_i, b_i]$," *Commun. ACM,* vol. 21, 1978, pp. 540-544.

[11] P. Gupta, R. Janardan, M. Smid and B. Dasgupta, "The rectangle enclosure and point-dominance problems revisited," *Int'l J. Comput. Geom. Appl.*, vol. 7, 1997, pp. 437-455.

[12] U. Gupta, D. T. Lee and J. Y-T. Leung, "An optimal solution for the channel-assignment problem," *IEEE Trans. Comput.*, Nov. 1979, pp. 807-810.

[13] E. Horowitz, S. Sahni and S. Anderson-Freed, *Fundamentals of Data Structures in C*, Computer Science Press, 1993.

[14] H. Imai and Ta. Asano, "Finding the connected components and a maximum clique of an intersection graph of rectangles in the plane," *J. Algorithms*, vol. 4, 1983, pp. 310-323.

[15] D. T. Lee and F.P. Preparata, "An improved algorithm for the rectangle enclosure problem," *J. Algorithms,* 3,3 Sept. 1982, pp. 218-224.

[16] D. T. Lee, "Maximum clique problem of rectangle graphs," in *Advances in Computing Research,* Vol. 1, ed. F.P. Preparata, JAI Press Inc., 1983, 91-107.

[17] J. van Leeuwen and D. Wood, "The measure problem for rectangular ranges in d-space," *J. Algorithms*, vol. 2, 1981, pp. 282-300.

[18] G. S. Lueker and D. E. Willard, "A data structure for dynamic range queries," *Inform. Process. Lett.*, (15,5), 1982, pp. 209-213.

[19] E. M. McCreight, "Priority search trees," *SIAM J. Comput.*, (14,1), 1985, pp. 257-276.

[20] L. Monier, "Combinatorial solutions of multidimensional divide-and-conquer recurrences," *J. Algorithms*, vol. 1, 1980, pp. 60-74.

[21] M. H. Overmars and J. van Leeuwen, "Two general methods for dynamizing decomposable searching problems," *Computing*, vol. 26, 1981, pp. 155-166.

[22] F. P. Preparata and M. I. Shamos, *Computational Geometry: An Introduction*, 3rd edition, Springer-Verlag, 1990.

[23] M. Sarrafzadeh and D. T. Lee, "Restricted track assignment with applications," *Int'l J. Comput. Geom. Appl.*, vol. 4, 1994, pp. 53-68.

[24] D. E. Willard, "New data structures for orthogonal range queries," *SIAM J. Comput.*, vol. 14, 1985, pp. 232-253.

19
Quadtrees and Octrees

Srinivas Aluru
Iowa State University

19.1 Introduction

Quadtrees are hierarchical spatial tree data structures that are based on the principle of recursive decomposition of space. The term *quadtree* originated from representation of two dimensional data by recursive decomposition of space using separators parallel to the coordinate axis. The resulting split of a region into four regions corresponding to southwest, northwest, southeast and northeast quadrants is represented as four children of the node corresponding to the region, hence the term"quad"tree. In a three dimensional analogue, a region is split into eight regions using planes parallel to the coordinate planes. As each internal node can have eight children corresponding to the 8-way split of the region associated with it, the term *octree* is used to describe the resulting tree structure. Analogous data structures for representing spatial data in higher than three dimensions are called *hyperoctrees*. It is also common practice to use the term *quadtrees* in a generic way irrespective of the dimensionality of the spatial data. This is especially useful when describing algorithms that are applicable regardless of the specific dimensionality of the underlying data.

Several related spatial data structures are described under the common rubric of quadtrees. Common to these data structures is the representation of spatial data at various levels of granularity using a hierarchy of regular, geometrically similar regions (such as cubes, hyperrectangles etc.). The tree structure allows quick focusing on regions of interest, which facilitates the design of fast algorithms. As an example, consider the problem of finding all points in a data set that lie within a given distance from a query point, commonly known as the *spherical region query*. In the absence of any data organization, this requires checking the distance from the query point to each point in the data set. If a quadtree of the data is available, large regions that lie outside the spherical region of interest can be quickly discarded from consideration, resulting in great savings in execution time. Furthermore, the unit aspect ratio employed in most quadtree data structures allows geometric arguments useful in designing fast algorithms for certain classes of applications.

In constructing a quadtree, one starts with a square, cubic or hypercubic region (depending on the dimensionality) that encloses the spatial data under consideration. The different variants of the quadtree data structure are differentiated by the principle used in the recursive decomposition process. One important aspect of the decomposition process is if the decomposition is guided by input data or is based on the principle of equal subdivision of the space itself. The former results in a tree size proportional to the size of the input. If all the input data is available a priori, it is possible to make the data structure height balanced. These attractive properties come at the expense of difficulty in making the data structure dynamic, typically in accommodating deletion of data. If the decomposition is based on equal subdivision of space, the resulting tree depends on the distribution of spatial data. As a result, the tree is height balanced and is linear in the size of input only when the distribution of the spatial data is uniform, and the height and size properties deteriorate with increase in nonuniformity of the distribution. The beneficial aspect is that the tree structure facilitates easy update operations and the regularity in the hierarchical representation of the regions facilitates geometric arguments helpful in designing algorithms.

Another important aspect of the decomposition process is the termination condition to stop the subdivision process. This identifies regions that will not be subdivided further, which will be represented by leaves in the quadtree. Quadtrees have been used as fixed resolution data structures, where the decomposition stops when a preset resolution is reached, or as variable resolution data structures, where the decomposition stops when a property based on input data present in the region is satisfied. They are also used in a hybrid manner, where the decomposition is stopped when either a resolution level is reached or when a property is satisfied.

Quadtrees are used to represent many types of spatial data including points, line segments, rectangles, polygons, curvilinear objects, surfaces, volumes and cartographic data. Their use is pervasive spanning many application areas including computational geometry, computer aided design (Chapter 52), computer graphics (Chapter 54), databases (Chapter 60), geographic information systems (Chapter 55), image processing (Chapter 57), pattern recognition, robotics and scientific computing. Introduction of the quadtree data structure and its use in applications involving spatial data dates back to the early 1970s and can be attributed to the work of Klinger [20], Finkel and Bentley [3], and Hunter [16]. Due to extensive research over the last three decades, a large body of literature is available on quadtrees and its myriad applications. For a detailed study on this topic, the reader is referred to the classic textbooks by Samet [29, 30]. Development of quadtree like data structures, algorithms and applications continues to be an active research area with significant research developments in recent years. In this chapter, we attempt a coverage of some of the classical results together with some of the more recent developments in the design and analysis of algorithms using quadtrees and octrees.

19.2 Quadtrees for Point Data

We first explore quadtrees in the context of the simplest type of spatial data – multidimensional points. Consider a set of n points in d dimensional space. The principal reason a spatial data structure is used to organize multidimensional data is to facilitate queries

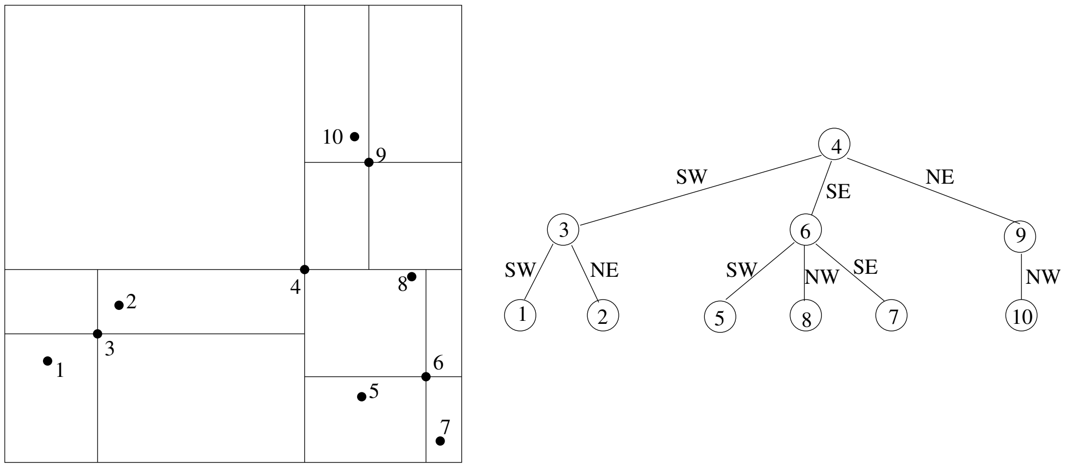


FIGURE 19.1: A two dimensional set of points and a corresponding point quadtree.

requiring spatial information. A number of such queries can be identified for point data. For example:

1. *Range query:* Given a range of values for each dimension, find all the points that lie within the range. This is equivalent to retrieving the input points that lie within a specified hyperrectangular region. Such a query is often useful in database information retrieval.
2. *Spherical region query:* Given a query point p and a radius r, find all the points that lie within a distance of r from p. In a typical molecular dynamics application, spherical region queries centered around each of the input points is required.
3. *All nearest neighbor query:* Given n points, find the nearest neighbor of each point within the input set.

While quadtrees are used for efficient execution of such spatial queries, one must also design algorithms for the operations required of almost any data structure such as constructing the data structure itself, and accommodating searches, insertions and deletions. Though such algorithms will be covered first, it should be kept in mind that the motivation behind the data structure is its use in spatial queries. If all that were required was search, insertion and deletion operations, any one dimensional organization of the data using a data structure such as a binary search tree would be sufficient.

19.2.1 Point Quadtrees

The point quadtree is a natural generalization of the binary search tree data structure to multiple dimensions. For convenience, first consider the two dimensional case. Start with a square region that contains all of the input points. Each node in the point quadtree corresponds to an input point. To construct the tree, pick an arbitrary point and make it the root of the tree. Using lines parallel to the coordinate axis that intersect at the selected point (see Figure 19.1), divide the region into four subregions corresponding to the southwest, northwest, southeast and northeast quadrants, respectively. Each of the subregions is recursively decomposed in a similar manner to yield the point quadtree. For points that lie at the boundary of two adjacent regions, a convention can be adopted to treat the points as belonging to one of the regions. For instance, points lying on the left and bottom edges of a region may be considered included in the region, while points lying on the top and right edges are not. When a region corresponding to a node in the tree contains a single point, it is considered a leaf node. Note that point quadtrees are not unique and their structure depends on the selection of points used in region subdivisions. Irrespective of the choices made, the resulting tree will have n nodes, one corresponding to each input point.

If all the input points are known in advance, it is easy to choose the points for subdivision so as to construct a height balanced tree. A simple way to do this is to sort the points with one of the coordinates, say x, as the primary key and the other coordinate, say y, as the secondary key. The first subdivision point is chosen to be the median of this sorted data. This will ensure that none of the children of the root node receives more than half the points. In $O(n)$ time, such a sorted list can be created for each of the four resulting subregions. As the total work at every level of the tree is bounded by $O(n)$, and there are at most $O(\log n)$ levels in the tree, a height balanced point quadtree can be built in $O(n \log n)$ time. Generalization to d dimensions is immediate, with $O(dn \log n)$ run time.

The recursive structure of a point quadtree immediately suggests an algorithm for searching. To search for a point, compare it with the point stored at the root. If they are different, the comparison immediately suggests the subregion containing the point. The search is directed to the corresponding child of the root node. Thus, search follows a path in the quadtree until either the point is discovered, or a leaf node is reached. The run time is bounded by $O(dh)$, where h is the height of the tree.

To insert a new point not already in the tree, first conduct a search for it which ends in a leaf node. The leaf node now corresponds to a region containing two points. One of them is chosen for subdividing the region and the other is inserted as a child of the node corresponding to the subregion it falls in. The run time for point insertion is also bounded by $O(dh)$, where h is the height of the tree after insertion. One can also construct the tree itself by repeated insertions using this procedure. Similar to binary search trees, the run time under a random sequence of insertions is expected to be $O(n \log n)$ [6]. Overmars and van Leeuwen [24] present algorithms for constructing and maintaining optimized point quadtrees irrespective of the order of insertions.

Deletion in point quadtrees is much more complex. The point to be deleted is easily identified by a search for it. The difficulty lies in identifying a point in its subtree to take the place of the deleted point. This may require nontrivial readjustments in the subtree underneath. The reader interested in deletion in point quadtrees is referred to [27]. An analysis of the expected cost of various types of searches in point quadtrees is presented by Flajolet *et al.* [7].

For the remainder of the chapter, we will focus on quadtree data structures that use equal subdivision of the underlying space, called *region quadtrees.* This is because we regard Bentley's multidimensional binary search trees [3], also called *k-d* trees, to be superior to point quadtrees. The *k-d* tree is a binary tree where a region is subdivided into two based only on one of the dimensions. If the dimension used for subdivision is cyclically rotated at consecutive levels in the tree, and the subdivision is chosen to be consistent with the point quadtree, then the resulting tree would be equivalent to the point quadtree but without the drawback of large degree (2^d in d dimensions). Thus, it can be argued that point quadtrees are contained in *k-d* trees. Furthermore, recent results on compressed region quadtrees indicate that it is possible to simultaneously achieve the advantages of both region and point quadtrees. In fact, region quadtrees are the most widely used form of quadtrees despite their dependence on the spatial distribution of the underlying data. While their use posed theoretical inconvenience — it is possible to create as large a worst-case tree as

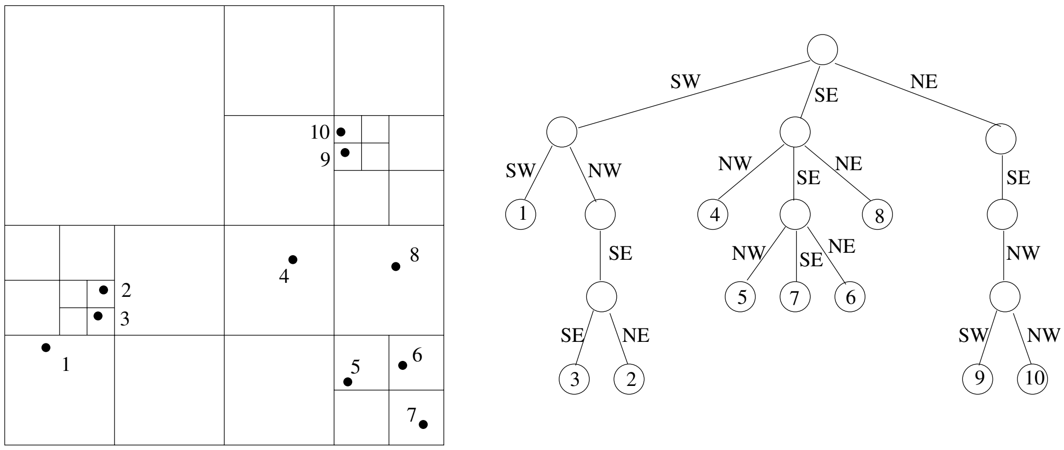


FIGURE 19.2: A two dimensional set of points and the corresponding region quadtree.

desired with as little as three points — they are widely acknowledged as the data structure of choice for practical applications. We will outline some of these recent developments and outline how good practical performance and theoretical performance guarantees can both be achieved using region quadtrees.

19.2.2 Region Quadtrees

The region quadtree for n points in d dimensions is defined as follows: Consider a hypercube large enough to enclose all the points. This region is represented by the root of the d-dimensional quadtree. The region is subdivided into 2^d subregions of equal size by bisecting along each dimension. Each of these regions containing at least one point is represented as a child of the root node. The same procedure is recursively applied to each child of the root node. The process is terminated when a region contains only a single point. This data structure is also known as the *point region quadtree*, or *PR-quadtree* for short [31]. At times, we will simply use the term *quadtree* when the tree implied is clear from the context. The region quadtree corresponding to a two dimensional set of points is shown in Figure 19.2. Once the enclosing cube is specified, the region quadtree is unique. The manner in which a region is subdivided is independent of the specific location of the points within the region. This makes the size of the quadtree sensitive to the spatial distribution of the points.

Before proceeding further, it is useful to establish a terminology to describe the type of regions that correspond to nodes in the quadtree. Call a hypercubic region containing all the points the *root cell*. Define a hierarchy of cells by the following: The root cell is in the hierarchy. If a cell is in the hierarchy, then the 2^d equal-sized cubic subregions obtained by bisecting along each dimension of the cell are also called *cells* and belong to the hierarchy (see Figure 19.3 for an illustration of the cell hierarchy in two dimensions). We use the term *subcell* to describe a cell that is completely contained in another. A cell containing the subcell is called a *supercell*. The subcells obtained by bisecting a cell along each dimension are called the *immediate subcells* with respect to the bisected cell. Also, a cell is the *immediate supercell* of any of its immediate subcells. We can treat a cell as a set of all points in space contained in the cell. Thus, we use $C \subseteq D$ to indicate that the cell C is a subcell of the cell D and $C \subset D$ to indicate that C is a subcell of D but $C \neq D$. Define the *length of a cell* C, denoted $length(C)$, to be the span of C along any dimension.

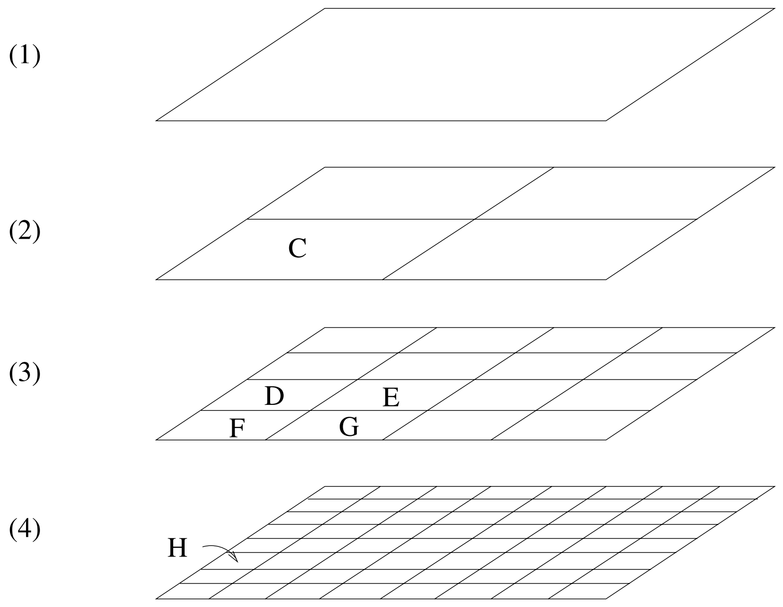


FIGURE 19.3: Illustration of hierarchy of cells in two dimensions. Cells D, E, F and G are immediate subcells of C. Cell H is an immediate subcell of D, and is a subcell of C.

An important property of the cell hierarchy is that, given two arbitrary cells, either one is completely contained in the other or they are disjoint. cells are considered disjoint if they are adjacent to each other and share a boundary.

Each node in a quadtree corresponds to a subcell of the root cell. Leaf nodes correspond to largest cells that contain a single point. There are as many leaf nodes as the number of points, n. The size of the quadtree cannot be bounded as a function of n, as it depends on the spatial distribution. For example, consider a data set of 3 points consisting of two points very close to each other and a faraway point located such that the first subdivision of the root cell will separate the faraway point from the other two. Then, depending on the specific location and proximity of the other two points, a number of subdivisions may be necessary to separate them. In principle, the location and proximity of the two points can be adjusted to create as large a worst-case tree as desired. In practice, this is an unlikely scenario due to limits imposed by computer precision.

From this example, it is intuitively clear that a large number of recursive subdivisions may be required to separate points that are very close to each other. In the worst case, the recursive subdivision continues until the cell sizes are so small that a single cell cannot contain both the points irrespective of their location. Subdivision is never required beyond this point, but the points may be separated sooner depending on their actual location. Let s be the smallest distance between any pair of points and D be the length of the root cell. An upper bound on the height of the quadtree is obtained by considering the worst-case path needed to separate a pair of points which have the smallest pairwise distance. The length of the smallest cell that can contain two points s apart in d dimensions is $\frac{s}{\sqrt{d}}$ (see Figure 19.4 for a two and three dimensional illustration). The paths separating the closest points may contain recursive subdivisions until a cell of length smaller than $\frac{s}{\sqrt{d}}$ is reached. Since each subdivision halves the length of the cells, the maximum path length is given by the smallest k for which $\frac{D}{2^k} < \frac{s}{\sqrt{d}}$, or $k = \lceil \log \frac{\sqrt{d}D}{s} \rceil$. For a fixed number of dimensions, the

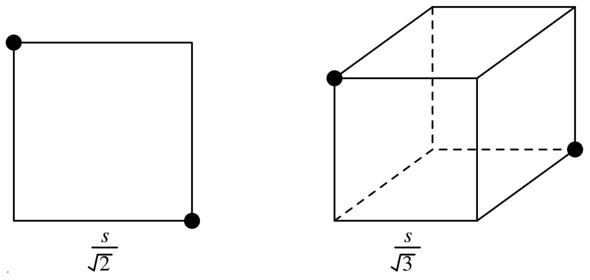


FIGURE 19.4: Smallest cells that could possibly contain two points that are a distance s apart in two and three dimensions.

worst-case path length is $O(\log \frac{D}{s})$. Since the tree has n leaves, the number of nodes in the tree is bounded by $O(n \log \frac{D}{s})$. In the worst case, D is proportional to the largest distance between any pair of points. Thus, the height of a quadtree is bounded by the logarithm of the ratio of the largest pairwise distance to the smallest pairwise distance. This ratio is a measure of the degree of nonuniformity of the distribution.

Search, insert and delete operations in region quadtrees are rather straightforward. To search for a point, traverse a path from root to a leaf such that each cell on the path encloses the point. If the leaf contains the point, it is in the quadtree. Otherwise, it is not. To insert a point not already in the tree, search for the point which terminates in a leaf. The leaf node corresponds to a region which originally had one point. To insert a new point which also falls within the region, the region is subdivided as many times as necessary until the two points are separated. This may create a chain of zero or more length below the leaf node followed by a branching to separate the two points. To delete a point present in the tree, conduct a search for the point which terminates in a leaf. Delete the leaf node. If deleting the node leaves its parent with a single child, traverse the path from the leaf to the root until a node with at least two children is encountered. Delete the path below the level of the child of this node. Each of the search, insert and delete operations takes $O(h)$ time, where h is the height of the tree. Construction of a quadtree can be done either through repeated insertions, or by constructing the tree level by level starting from the root. In either case, the worst case run time is $O\left(n \log \frac{D}{s}\right)$. We will not explore these algorithms further in favor of superior algorithms to be described later.

19.2.3 Compressed Quadtrees and Octrees

In an n-leaf tree where each internal node has at least two children, the number of nodes is bounded by $2n - 1$. The size of quadtrees is distribution dependent because there can be internal nodes with only one child. In terms of the cell hierarchy, a cell may contain all its points in a small volume so that, recursively subdividing it may result in just one of the immediate subcells containing the points for an arbitrarily large number of steps. Note that the cells represented by nodes along such a path have different sizes but they all enclose the same points. In many applications, all these nodes essentially contain the same information as the information depends only on the points the cell contains. This prompted the development of compressed quadtrees, which are obtained by compressing each such path into a single node. Therefore, each node in a compressed quadtree is either a leaf or has at least two children. The compressed quadtree corresponding to the quadtree of Figure 19.2 is depicted in Figure 19.5. Compressed quadtrees originated from the work

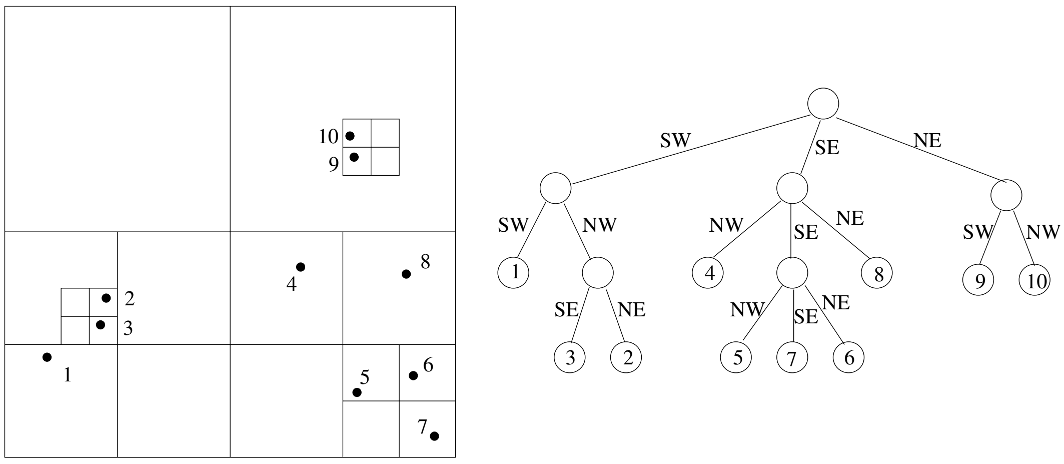


FIGURE 19.5: The two-dimensional set of points from Figure 19.2, and the corresponding compressed quadtree.

of Clarkson [4] in the context of the all nearest neighbors problem and further studied by Aluru and Sevilgen [2].

A node v in the compressed quadtree is associated with two cells, *large cell of* v ($L(v)$) and *small cell of* v ($S(v)$). They are the largest and smallest cells that enclose the points in the subtree of the node, respectively. When $S(v)$ is subdivided, it results in at least two non-empty immediate subcells. For each such subcell C resulting from the subdivision, there is a child u such that $L(u) = C$. Therefore, $L(u)$ at a node u is an immediate subcell of $S(v)$ at its parent v. A node is a leaf if it contains a single point and the small cell of a leaf node is the hypothetical cell with zero length containing the point.

The size of a compressed quadtree is bounded by $O(n)$. The height of a compressed quadtree has a lower bound of $\Omega(\log n)$ and an upper bound of $O(n)$. Search, insert and delete operations on compressed quadtrees take $O(h)$ time. In practice, the height of a compressed quadtree is significantly smaller than suggested by the upper bound because a) computer precision limits restrict the ratio of largest pairwise distance to smallest pairwise distance that can be represented, and b) the ratio of length scales represented by a compressed quadtree of height h is at least $2^h : 1$. In most practical applications, the height of the tree is so small that practitioners use representation schemes that allow only trees of constant height [12, 37] or even assume that the height is constant in algorithm analysis [11]. For instance, a compressed octree of height 20 allows potentially $8^{20} = 2^{60}$ leaf nodes and a length scale of $2^{20} : 1 \approx 10^6 : 1$.

Though compressed quadtrees are described as resulting from collapsing chains in quadtrees, such a procedure is not intended for compressed quadtree construction. Instead, algorithms for direct construction of compressed quadtrees in $O(dn \log n)$ time will be presented, which can be used to construct quadtrees efficiently if necessary. To obtain a quadtree from its compressed version, identify each node whose small cell is not identical to its large cell and replace it by a chain of nodes corresponding to the hierarchy of cells that lead from the large cell to the small cell.

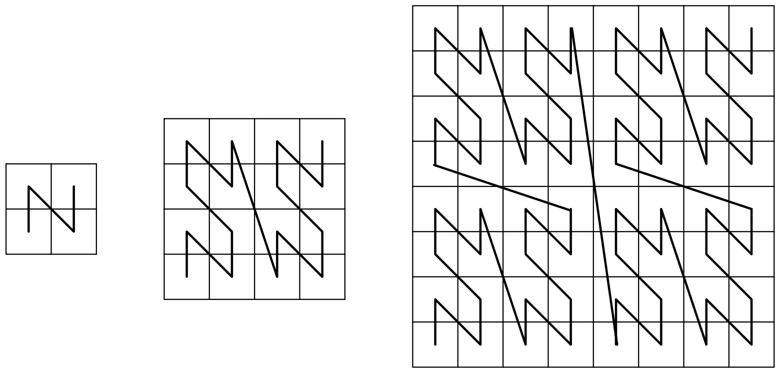


FIGURE 19.6: Z-curve for 2×2, 4×4 and 8×8 cell decompositions.

19.2.4 Cell Orderings and Space-Filling Curves

We explore a suitable one dimensional ordering of cells and use it in conjunction with spatial ordering to develop efficient algorithms for compressed quadtrees. First, define an ordering for the immediate subcells of a cell. In two dimensions, we use the order SW, NW, SE and NE. The same ordering has been used to order the children of a node in a two dimensional quadtree (Figure 19.2 and Figure 19.5). Now consider ordering two arbitrary cells. If one of the cells is contained in the other, the subcell precedes the supercell. If the two cells are disjoint, the smallest supercell enclosing both the cells contains them in different immediate subcells of it. Order the cells according to the order of the immediate subcells containing them. This defines a total order on any collection of cells with a common root cell. It follows that the order of leaf regions in a quadtree corresponds to the left-or-right order in which the regions appear in our drawing scheme. Similarly, the ordering of all regions in a quadtree corresponds to the postorder traversal of the quadtree. These concepts naturally extend to higher dimensions. Note that any ordering of the immediate subcells of a cell can be used as foundation for cell orderings.

Ordering of cells at a particular resolution in the manner described above can be related to space filling curves. Space filling curves are proximity preserving mappings from a multi-dimensional uniform cell decomposition to a one dimensional ordering. The path implied in the multidimensional space by the linear ordering, i.e., the sequence in which the multidimensional cells are visited according to the linear ordering, forms a non-intersecting curve. Of particular interest is Morton ordering, also known as the Z-space filling curve [22]. The Z-curves for 2×2, 4×4 and 8×8 cell decompositions are shown in Figure 19.6. Consider a square two dimensional region and its $2^k \times 2^k$ cell decomposition. The curve is considered to originate in the lower left corner and terminate in the upper right corner. The curve for a $2^k \times 2^k$ grid is composed of four $2^{k-1} \times 2^{k-1}$ grid curves one in each quadrant of the $2^k \times 2^k$ grid and the tail of one curve is connected to the head of the next as shown in the figure. The order in which the curves are connected is the same as the order of traversal of the 2×2 curve. Note that Morton ordering of cells is consistent with the cell ordering specified above. Other space filling curves orderings such as graycode [5] and Hilbert [13] curve can be used and quadtree ordering schemes consistent with these can also be utilized. We will continue to utilize the Z-curve ordering as it permits a simpler bit interleaving scheme which will be presented and exploited later.

Algorithms on compressed quadtree rely on the following operation due to Clarkson [4]:

LEMMA 19.1 Let R be the product of d intervals $I_1 \times I_2 \times \ldots \times I_d$, i.e., R is a hyper-rectangular region in d dimensional space. The smallest cell containing R can be found in $O(d)$ time, which is constant for any fixed d.

The procedure for computing the smallest cell uses floor, logarithm and bitwise exclusive-or operations. An extended RAM model is assumed in which these are considered constant time operations. The reader interested in proof of Lemma 19.1 is referred to [1, 4]. The operation is useful in several ways. For example, the order in which two points appear in the quadtree as per our ordering scheme is independent of the location of other points. To determine the order for two points, say $(x_1, x_2, \ldots, x_d)$ and $(y_1, y_2, \ldots, y_d)$, find the smallest cell that contains $[x_1, y_1] \times [x_2, y_2] \times \ldots \times [x_d, y_d]$. The points can then be ordered according to its immediate subcells that contain the respective points. Similarly, the smallest cell containing a pair of other cells, or a point and a cell, can be determined in $O(d)$ time.

19.2.5 Construction of Compressed Quadtrees

A Divide-and-Conquer Construction Algorithm

Let T_1 and T_2 be two compressed quadtrees representing two distinct sets S_1 and S_2 of points. Let r_1 (respectively, r_2) be the root node of T_1 (respectively, T_2). Suppose that $L(r_1) = L(r_2)$, i.e., both T_1 and T_2 are constructed starting from a cell large enough to contain $S_1 \cup S_2$. A compressed quadtree T for $S_1 \cup S_2$ can be constructed in $O(|S_1| + |S_2|)$ time by merging T_1 and T_2.

To merge T_1 and T_2, start at their roots and merge the two trees recursively. Suppose that at some stage during the execution of the algorithm, node u in T_1 and node v in T_2 are being considered. An invariant of the merging algorithm is that $L(u)$ and $L(v)$ cannot be disjoint. Furthermore, it can be asserted that $S(u) \cup S(v) \subseteq L(u) \cap L(v)$. In merging two nodes, only the small cell information is relevant because the rest of the large cell $(L(v) - S(v))$ is empty. For convenience, assume that a node may be empty. If a node has less than 2^d children, we may assume empty nodes in place of the absent children. Four distinct cases arise:

- Case I: If a node is empty, the result of merging is simply the tree rooted at the other node.
- Case II: If $S(u) = S(v)$, the corresponding children of u and v have the same large cells which are the the immediate subcells of $S(u)$ (or equivalently, $S(v)$). In this case, merge the corresponding children one by one.
- Case III: If $S(v) \subset S(u)$, the points in v are also in the large cell of one of the children of u. Thus, this child of u is merged with v. Similarly, if $S(u) \subset S(v)$, u is merged with the child of v whose large cell contains $S(u)$.
- Case IV: If $S(u) \cap S(v) = \emptyset$, then the smallest cell containing both $S(u)$ and $S(v)$ contains $S(u)$ and $S(v)$ in different immediate subcells. In this case, create a new node in the merged tree with its small cell as the smallest cell that contains $S(u)$ and $S(v)$ and make u and v the two children of this node. Subtrees of u and v are disjoint and need not be merged.

LEMMA 19.2 Two compressed quadtrees with a common root cell can be merged in time proportional to the sum of their sizes.

Proof The merging algorithm presented above performs a preorder traversal of each compressed quadtree. The whole tree may not need to be traversed because in merging a node, it may be determined that the whole subtree under the node directly becomes a subtree of the resulting tree. In every step of the merging algorithm, we advance on one of the trees after performing at most $O(d)$ work. Thus, the run time is proportional to the sum of the sizes of the trees to be merged.

To construct a compressed quadtree for n points, scan the points and find the smallest and largest coordinate along each dimension. Find a region that contains all the points and use this as the root cell of every compressed quadtree constructed in the process. Recursively construct compressed quadtrees for $\lfloor \frac{n}{2} \rfloor$ points and the remaining $\lceil \frac{n}{2} \rceil$ points and merge them in $O(dn)$ time. The compressed quadtree for a single point is a single node v with the root cell as $L(v)$. The run time satisfies the recurrence

$$T(n) = T\left(\lfloor \frac{n}{2} \rfloor\right) + T\left(\lceil \frac{n}{2} \rceil\right) + O(dn)$$

resulting in $O(dn \log n)$ run time.

Bottom-up Construction

To perform a bottom-up construction, first compute the order of the points in $O(dn \log n)$ time using any optimal sorting algorithm and the ordering scheme described previously. The compressed quadtree is then incrementally constructed starting from the single node tree for the first point and inserting the remaining points as per the sorted list. During the insertion process, keep track of the most recently inserted leaf. Let p be the next point to be inserted. Starting from the most recently inserted leaf, traverse the path from the leaf to the root until the first node v such that $p \in L(v)$ is encountered. Two possibilities arise:

- Case I: If $p \notin S(v)$, then p is in the region $L(v) - S(v)$, which was empty previously. The smallest cell containing p and $S(v)$ is a subcell of $L(v)$ and contains p and $S(v)$ in different immediate subcells. Create a new node u between v and its parent and insert p as a child of u.
- Case II: If $p \in S(v)$, v is not a leaf node. The compressed quadtree presently does not contain a node that corresponds to the immediate subcell of $S(v)$ that contains p, i.e., this immediate subcell does not contain any of the points previously inserted. Therefore, it is enough to insert p as a child of v corresponding to this subcell.

Once the points are sorted, the rest of the algorithm is identical to a post-order walk on the final compressed quadtree with $O(d)$ work per node. The number of nodes visited per insertion is not bounded by a constant but the number of nodes visited over all insertions is $O(n)$, giving $O(dn)$ run time. Combined with the initial sorting of the points, the tree can be constructed in $O(dn \log n)$ time.

19.2.6 Basic Operations

Fast algorithms for operations on quadtrees can be designed by simultaneously keeping track of spatial ordering and one dimensional ordering of cells in the compressed quadtree. The spatial ordering is given by the compressed quadtree itself. In addition, a balanced binary search tree (BBST) is maintained on the large cells of the nodes to enable fast cell searches. Both the trees consist of the same nodes and this can be achieved by allowing each node to have pointers corresponding to compressed quadtree structure and pointers corresponding to BBST structure.

Point and Cell Queries

Point and cell queries are similar since a point can be considered to be a zero length cell. A node v is considered to represent cell C if $S(v) \subseteq C \subseteq L(v)$. The node in the compressed quadtree representing the given cell is located using the BBST. Traverse the path in the BBST from the root to the node that is being searched in the following manner: To decide which child to visit next on the path, compare the query cell with the large and small cells at the node. If the query cell precedes the small cell in cell ordering, continue the search with the left child. If it succeeds the large cell in cell ordering, continue with the right child. If it lies between the small cell and large cell in cell ordering, the node represents the query cell. As the height of a BBST is $O(\log n)$, the time taken for a point or cell query is $O(d \log n)$.

Insertions and Deletions

As points can be treated as cells of zero length, insertion and deletion algorithms will be discussed in the context of cells. These operations are meaningful only if a cell is inserted as a leaf node or deleted if it is a leaf node. Note that a cell cannot be deleted unless all its subcells are previously deleted from the compressed quadtree.

Cell Insertion

To insert a given cell C, first check whether it is represented in the compressed quadtree. If not, it should be inserted as a leaf node. Create a node v with $S(v) = C$ and first insert v in the BBST using a standard binary search tree insertion algorithm. To insert v in the compressed quadtree, first find the BBST successor of v, say u. Find the smallest cell D containing C and the $S(u)$. Search for cell D in the BBST and identify the corresponding node w. If w is not a leaf, insert v as a child of w in compressed quadtree. If w is a leaf, create a new node w' such that $S(w') = D$. Nodes w and v become the children of w' in the compressed quadtree. The new node w' should be inserted in the BBST. The overall algorithm requires a constant number of insertions and searches in the BBST, and takes $O(d \log n)$ time.

Cell Deletion

As in insertion, the cell should be deleted from the BBST and the compressed quadtree. To delete the cell from BBST, the standard deletion algorithm is used. During the execution of this algorithm, the node representing the cell is found. The node is deleted from the BBST only if it is present as a leaf node in the compressed quadtree. If the removal of this node from the compressed quadtree leaves its parent with only one child, the parent is deleted as well. Since each internal node has at least two children, the delete operation cannot propagate to higher levels in the compressed quadtree.

19.2.7 Practical Considerations

In most practical applications, the height of a region quadtree is rather small because the spatial resolution provided by a quadtree is exponential in its height. This can be used to design schemes that will greatly simplify and speedup quadtree algorithms.

Consider numbering the 2^{2k} cells of a $2^k \times 2^k$ two dimensional cell decomposition in the order specified by the Z-curve using integers $0 \ldots 4^k - 1$ (Figure 19.6). Represent each cell in the cell space using a coordinate system with k bits for each coordinate. From the definition of the Z-curve, it follows that the number of a cell can be obtained by interleaving the bits representing the x and y coordinates of the cell, starting from the x coordinate. For example, $(3, 5) = (011, 101)$ translates to $011011 = 27$. The procedure can be extended to higher dimensions. If $(x_1, x_2, \ldots x_d)$ represents the location of a d dimensional cell, the corresponding number can be obtained by taking bit representations of $x_1, x_2, \ldots x_d$, and interleaving them.

The same procedure can be described using uncompressed region quadtrees. Label the 2^d edges connecting a node to its children with bit strings of length d. This is equivalent to describing the immediate subcells of a cell using one bit for each coordinate followed by bit interleaving. A cell can then be described using concatenation of the bits along the path from the root to the cell. This mechanism can be used to simultaneously describe cells at various length scales. The bit strings are conveniently stored in groups of 32 or 64 bits using integer data types. This creates a problem in distinguishing certain cells. For instance, consider distinguishing cell "000" from cell "000000" in an octree. To ensure each bit string translates to a unique integer when interpreted as a binary number, it is prefixed by a 1. It also helps in easy determination of the length of a bit string by locating the position of the leftmost 1. This leads to a representation that requires $dk + 1$ bits for a d dimensional quadtree with a height of at most k. Such a representation is very beneficial because primitive operations on cells can be implemented using bit operations such as and, or, exclusive-or etc. For example, 128 bits (4 integers on a 32 bit computer and 2 integers on a 64 bit computer) are sufficient to describe an octree of height 42, which allows representation of length scales $2^{42} : 1 > 4 \times 10^{12} : 1$.

The bit string based cell representation greatly simplifies primitive cell operations. In the following, the name of a cell is also used to refer to its bit representation:

- *Check if $C_1 \subseteq C_2$.* If C_2 is a prefix of C_1, then C_1 is contained in C_2, otherwise not.
- *Find the smallest cell enclosing C_1 and C_2.* This is obtained by finding the longest common prefix of C_1 and C_2 whose length is 1 *mod* d.
- *Find the immediate subcell of C_1 that contains C_2.* If $dl + 1$ is the number of bits representing C_1, the required immediate subcell is given by the first $(d + 1)l + 1$ bits of C_2.

Consider n points in a root cell and let k denote the largest resolution to be used. Cells are not subdivided further even if they contain multiple points. From the coordinates of a point, it is easy to compute the leaf cell containing it. Because of the encoding scheme used, if cell C should precede cell D in the cell ordering, the number corresponding to the binary interpretation of the bit string representation of C is smaller than the corresponding number for D. Thus, cells can be sorted by simply treating them as numbers and ordering them accordingly.

Finally, binary search trees and the attendant operations on them can be completely avoided by using hashing to directly access a cell. An n leaf compressed quadtree has at most $2n - 1$ nodes. Hence, an array of that size can be conveniently used for hashing. If all cells at the highest resolution are contained in the quadtree, i.e., $n = d^k$, then an array of size $2n - 1$ can be used to directly index cells. Further details of such representation are left as an exercise to the reader.

19.3 Spatial Queries with Region Quadtrees

In this section, we consider a number of spatial queries involving point data, and algorithms for them using compressed region quadtrees.

19.3.1 Range Query

Range queries are commonly used in database applications. Database records with d keys can be represented as points in d-dimensional space. In a range query, ranges of values are specified for all or a subset of keys with the objective of retrieving all the records that satisfy the range criteria. Under the mapping of records to points in multidimensional space, the ranges define a (possibly open-ended) hyperrectangular region. The objective is to retrieve all the points that lie in this query region.

As region quadtrees organize points using a hierarchy of cells, the range query can be answered by finding a collection $\mathcal{C}$ of cells that are both fully contained in the query region and completely encompass the points in it. This can be achieved by a top-down traversal of the compressed region quadtree starting from the root. To begin with, $\mathcal{C}$ is empty. Consider a node v and its small cell $S(v)$. If $S(v)$ is outside the query region, the subtree underneath it can be safely discarded. If $S(v)$ is completely inside the query region, it is added to $\mathcal{C}$. All points in the subtree of $S(v)$ are within the query region and reported as part of the output. If $S(v)$ overlaps with the query region but is not contained in it, each child of v is examined in turn.

If the query region is small compared to the size of the root cell, it is likely that a path from the root is traversed where each cell on the path completely contains the query region. To avoid this problem, first compute the smallest cell encompassing the query region. This cell can be searched in $O(d \log n)$ time using the cell search algorithm described before, or perhaps in $O(d)$ time if the hashing/indexing technique is applicable. The top-down traversal can start from the identified cell. When the cell is subdivided, at least two of its children represent cells that overlap with the query region. Consider a child cell and the part of the query region that is contained in it. The same idea can be recursively applied by finding the smallest cell that encloses this part of the query region and directly finding this cell rather than walking down a path in the tree to reach there. This ensures that the number of cells examined during the algorithm is $O(|\mathcal{C}|)$. To see why, consider the cells examined as organized into a tree based on subcell-supercell relationships. The leaves of the tree are the collection of cells $\mathcal{C}$. Each cell in the tree is the smallest cell that encloses a subregion of the query region. Therefore, each internal node has at least two children. Consequently, the size of the tree, or the number of cells examined in answering the range query is $O(|\mathcal{C}|)$. For further study of range queries and related topics, see [5, 23, 25].

Next, we turn our attention to a number of spatial queries which we categorize as group queries. In group queries, a query to retrieve points that bear a certain relation to a query point is specified. The objective of the group query is to simultaneously answer n queries with each input point treated as a query point. For example, given n points, finding the nearest neighbor of each point is a group query. While the run time of performing the query on an individual point may be large, group queries can be answered more efficiently by intelligently combining the work required in answering queries for the individual points. Instead of presenting a different algorithm for each query, we show that the same generic framework can be used to solve a number of such queries. The central idea behind the group query algorithm is to realize run time savings by processing queries together for nearby points. Consider a cell C in the compressed quadtree and the (as yet uncomputed) set of points that result from answering the query for each point in C. The algorithm

Algorithm 1 Group-query (v)

```
$P$ = active set at $v$'s parent
$A$ = active set at $v = \emptyset$
While $P \neq \emptyset$ do
    $u$ = *Select (P)*
    $P = P - \{u\}$
    decision = *Status (v, u)*
    If decision = PROCESS
        *Process (v, u)*
    If decision = UNKNOWN
        If $S(u) \subseteq S(v)$
            $A = A \cup \{u\}$
        Else $P = P \cup children(u)$
For each child $u$ of $v$
    **Group-query** ($u$)
```

FIGURE 19.7: Unified algorithm for the group queries.

keeps track of a collection of cells of size as close to C as possible that is guaranteed to contain these points. The algorithm proceeds in a hierarchical manner by computing this information for cells of decreasing sizes (see Figure 19.7).

A node u is said to be *resolved* with respect to node v if, either all points in $S(u)$ are in the result of the query for all points in $S(v)$ or none of the points in $S(u)$ is in the result of the query for any point in $S(v)$. Define the *active set* of a node v to be the set of nodes u that cannot be resolved with respect to v such that $S(u) \subseteq S(v) \subseteq L(u)$. The algorithm uses a depth first search traversal of the compressed quadtree. The active set of the root node contains itself. The active set of a node v is calculated by traversing portions of the subtrees rooted at the nodes in the active set of its parent. The functions *Select, Status* and *Process* used in the algorithm are designed based on the specific group query. When considering the status of a node u with respect to the node v, the function *Status(v,u)* returns one of the following three values:

- PROCESS – If $S(u)$ is in the result of the query for all points in $S(v)$
- DISCARD – If $S(u)$ is not in the result of the query for all points in $S(v)$
- UNKNOWN – If neither of the above is true

If the result is either PROCESS or DISCARD, the children of u are not explored. Otherwise, the size of $S(u)$ is compared with the size of $S(v)$. If $S(u) \subseteq S(v)$, then u is added to the set of active nodes at v for consideration by v's children. If $S(u)$ is larger, then u's children are considered with respect to v. It follows that the active set of a leaf node is empty, as the length of its small cell is zero. Therefore, entire subtrees rooted under the active set of the parent of a leaf node are explored and the operation is completed for the point inhabiting that leaf node. The function *Process(v,u)* reports all points in $S(u)$ as part of the result of query for each point in $S(v)$.

The order in which nodes are considered is important for proving the run time of some operations. The function *Select* is used to accommodate this.

19.3.2 Spherical Region Queries

Given a query point and a distance $r > 0$, the spherical region query is to find all points that lie within a distance of r from the query point. The group version of the spherical region query is to take n points and a distance $r > 0$ as input, and answer spherical region queries with respect to each of the input points.

A cell D may contain points in the spherical region corresponding to some points in cell C only if the smallest distance between D and C is less than r. If the largest distance between D and C is less than r, then all points in D are in the spherical region of every point in C. Thus, the function *Status(v,u)* is defined as follows: If the largest distance between $S(u)$ and $S(v)$ is less than r, return PROCESS. If the smallest distance between $S(u)$ and $S(v)$ is greater than r, return DISCARD. Otherwise, return UNKNOWN. Processing u means including all the points in u in the query result for each point in v. For this query, no special selection strategy is needed.

19.3.3 k-Nearest Neighbors

For computing the k-nearest neighbors of each point, some modifications to the algorithm presented in Figure 19.7 are necessary. For each node w in P, the algorithm keeps track of the largest distance between $S(v)$ and $S(w)$. Let d_k be the k^{th} smallest of these distances. If the number of nodes in P is less than k, then d_k is set to ∞. The function *Status(v,u)* returns DISCARD if the smallest distance between $S(v)$ and $S(u)$ is greater than d_k. The option PROCESS is never used. Instead, for a leaf node, all the points in the nodes in its active set are examined to select the k nearest neighbors. The function *Select* picks the largest cell in P, breaking ties arbitrarily.

Computing k-nearest neighbors is a well-studied problem [4, 35]. The algorithm presented here is equivalent to Vaidya's algorithm [35, 36], even though the algorithms appear to be very different on the surface. Though Vaidya does not consider compressed quadtrees, the computations performed by his algorithm can be related to traversal on compressed quadtrees and a proof of the run time of the presented algorithm can be established by a correspondence with Vaidya's algorithm. The algorithm runs in $O(kn)$ time. The proof is quite elaborate, and omitted for lack of space. For details, see [35, 36]. The special case of $n = 1$ is called the all nearest neighbor query, which can be computed in $O(n)$ time.

19.4 Image Processing Applications

Quadtrees are ubiquitously used in image processing applications. Consider a two dimensional square array of pixels representing a binary image with black foreground and white background (Figure 19.8). As with region quadtrees used to represent point data, a hierarchical cell decomposition is used to represent the image. The pixels are the smallest cells used in the decomposition and the entire image is the root cell. The root cell is decomposed into its four immediate subcells and represented by the four children of the root node. If a subcell is completely composed of black pixels or white pixels, it becomes a leaf in the quadtree. Otherwise, it is recursively decomposed. If the resolution of the image is $2^k \times 2^k$, the height of the resulting region quadtree is at most k. A two dimensional image and its corresponding region quadtree are shown in Figure 19.8. In drawing the quadtree, the same ordering of the immediate subcells of a cell into SW, NW, SE and NE quadrants is followed. Each node in the tree is colored black, white, or gray, depending on if the cell consists of all black pixels, all white pixels, or a mixture of both black and white pixels, respectively. Thus, internal nodes are colored gray and leaf nodes are colored either black or white.

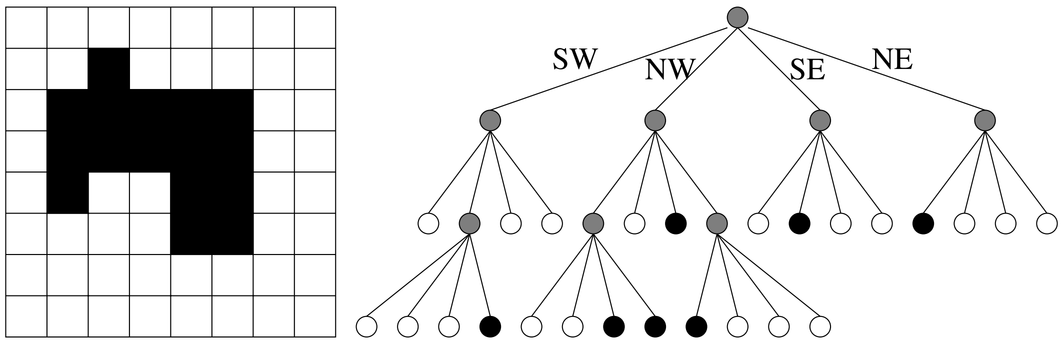


FIGURE 19.8: A two dimensional array of pixels, and the corresponding region quadtree.

Each internal node in the image quadtree has four children. For an image with n leaf nodes, the number of internal nodes is $\frac{(n-1)}{3}$. For large images, the space required for storing the internal nodes and the associated pointers may be expensive and several space-efficient storage schemes have been investigated. These include storing the quadtree as a collection of leaf cells using the Morton numbering of cells [21], or as a collection of black leaf cells only [8, 9], and storing the quadtree as its preorder traversal [19]. Iyengar *et al.* introduced a number of space-efficient representations of image quadtrees including forests of quadtrees [10, 26], translation invariant data structures [17, 32, 33] and virtual quadtrees [18].

The use of quadtrees can be easily extended to grayscale images and color images. Significant space savings can be realized by choosing an appropriate scheme. For example, 2^r gray levels can be encoded using r bits. The image can be represented using r binary valued quadtrees. Because adjacent pixels are likely to have gray levels that are closer, a gray encoding of the 2^r levels is advantageous over a binary encoding [19]. The gray encoding has the property that adjacent levels differ by one bit in the gray code representation. This should lead to larger blocks having the same value for a given bit position, leading to shallow trees.

19.4.1 Construction of Image Quadtrees

Region quadtrees for image data can be constructed using an algorithm similar to the bottom-up construction algorithm for point region quadtrees described in Subsection 19.2.5. In fact, constructing quadtrees for image data is easier because the smallest cells in the hierarchical decomposition are given by the pixels and all the pixels can be read by the algorithm as the image size is proportional to the number of pixels. Thus, quadtree for region data can be built in time linear in the size of the image, which is optimal. The pixels of the image are scanned in Morton order which eliminates the need for the sorting step in the bottom-up construction algorithm. It is wasteful to build the complete quadtree with all pixels as leaf nodes and then compact the tree by replacing maximal subtrees having all black or all white pixels with single leaf nodes, though such a method would still run in linear time and space. By reading the pixels in Morton order, a maximal black or white cell can be readily identified and the tree can be constructed using only such maximal cells as leaf nodes [27]. This will limit the space used by the algorithm to the final size of the quadtree.

19.4.2 Union and Intersection of Images

The union of two images is the overlay of one image over another. In terms of the array of pixels, a pixel in the union is black if the pixel is black in at least one of the images. In the region quadtree representation of images, the quadtree corresponding to the union of two images should be computed from the quadtrees of the constituent images. Let I_1 and I_2 denote the two images and T_1 and T_2 denote the corresponding region quadtrees. Let T denote the region quadtree of the union of I_1 and I_2. It is computed by a preorder traversal of T_1 and T_2 and examining the corresponding nodes/cells. Let v_1 in T_1 and v_2 in T_2 be nodes corresponding to the same region. There are three possible cases:

- Case I: If v_1 or v_2 is black, the corresponding node is created in T and is colored black. If only one of them is black and the other is gray, the gray node will contain a subtree underneath. This subtree need not be traversed.
- Case II: If v_1 (respectively, v_2) is white, v_2 (respectively, v_1) and the subtree underneath it (if any) is copied to T.
- Case III: If both v_1 and v_2 are gray, then the corresponding children of v_1 and v_2 are considered.

The tree resulting from the above merging algorithm may consist of unnecessary subdivisions of a cell consisting of completely black pixels. For example, if a region is marked gray in both T_1 and T_2 but each of the four quadrants of the region is black in at least one of T_1 and T_2, then the node corresponding to the region in T will have four children colored black. This is unnecessary as the node itself can be colored black and should be considered a leaf node. Such adjustments need not be local and may percolate up the tree. For instance, consider a checker board image of $2^k \times 2^k$ pixels with half black and half white pixels such that the north, south, east and west neighbors of a black pixel are white pixels and vice versa. Consider another checker board image with black and white interchanged. The quadtree for each of these images is a full 4-ary tree of level k. But overlaying one image over another produces one black square of size $2^k \times 2^k$. The corresponding quadtree should be a single black node. However, merging initially creates a full 4-ary tree of depth k. To compact the tree resulting from merging to create the correct region quadtree, a bottom-up traversal is performed. If all children of a node are black, then the children are removed and the node is colored black.

The intersection of two images can be computed similarly. A pixel in the intersection of two images is black only if the corresponding pixels in both the images are black. For instance, the intersection of the two complementary checkerboard images described above is a single white cell of size $2^k \times 2^k$. The algorithm for intersection can be obtained by interchanging the roles of black and white in the union algorithm. Similarly, a bottom-up traversal is used to detect nodes all of whose children are white, remove the children, and color the node white. Union and intersection algorithms can be easily generalized to multiple images.

19.4.3 Rotation and Scaling

Quadtree representation of images facilitates certain types of rotation and scaling operations. Rotations on images are often performed in multiples of 90 degrees. Such a rotation has the effect of changing the quadrants of a region. For example, a 90 degree clockwise rotation changes the SW, NW, SE, NE quadrants to NW, NE, SW, SE quadrants, respectively. This change should be recursively carried out for each region. Hence, a rotation that is a multiple of 90 degrees can be effected by simply reordering the child pointers of each node. Similarly, scaling by a power of two is trivial using quadtrees since it is simply a loss of resolution. For other linear transformations, see the work of Hunter [14, 15].

19.4.4 Connected Component Labeling

Two black pixels of a binary image are considered adjacent if they share a horizontal or vertical boundary. A pair of black pixels is said to be connected if there is a sequence of adjacent pixels leading from one to the other. A connected component is a maximal set of black pixels where every pair of pixels is connected. The connected component labeling problem is to find the connected components of a given image and give each connected component a unique label. Identifying the connected components of an image is useful in object counting and image understanding. Samet developed algorithms for connected component labeling using quadtrees [28].

Let B and W denote the number of black nodes and white nodes, respectively, in the quadtree. Samet's connected component labeling algorithm works in three stages:

1. Establish the adjacency relationships between pairs of black pixels.
2. Identify a unique label for each connected component. This can be thought of as computing the transitive closure of the adjacency relationships.
3. Label each black cell with the corresponding connected component.

The first stage is carried out using a postorder traversal of the quadtree during which adjacent black pixels are discovered and identified by giving them the same label. To begin with, all the black nodes are unlabeled. When a black region is considered, black pixels adjacent to it in two of the four directions, say north and east, are searched. The post order traversal of the tree traverses the regions in Morton order. Thus when a region is encountered, its south and west adjacent black regions (if any) would have already been processed. At that time, the region would have been identified as a neighbor. Thus, it is not necessary to detect adjacency relationships between a region and its south or west adjacent neighbors.

Suppose the postorder traversal is currently at a black node or black region R. Adjacent black regions in a particular direction, say north, are identified as follows. First, identify the adjacent region of the same size as R that lies to the north of R. Traverse the quadtree upwards to the root to identify the first node on the path that contains both the regions, or equivalently, find the lowest common ancestor of both the regions in the quadtree. If such a node does not exist, R has no north neighbor and it is at the boundary of the image. Otherwise, a child of this lowest common ancestor will be a region adjacent to the north boundary of R and of the same size as or larger than R. If this region is black, it is part of the same connected component as R. If neither the region nor R is currently labeled, create a new label and use it for both. If only one of them is labeled, use the same label for the other. If they are both already labeled with the same label, do nothing. If they are both already labeled with different labels, the label pair is recorded as corresponding to an adjacency relationship. If the region is gray, it is recursively subdivided and adjacent neighbors examined to find all black regions that are adjacent to R. For each black neighbor, the labeling procedure described above is carried out.

The run time of this stage of the algorithm depends on the time required to find the north and east neighbors. This process can be improved using the smallest-cell primitive described earlier. However, Samet proved that the average of the maximum number of nodes visited by the above algorithm for a given black region and a given direction is smaller than or equal to 5, under the assumption that a black region is equally likely to occur at any position and level in the quadtree [28]. Thus, the algorithm should take time proportional to the size of the quadtree, i.e., $O(W + B)$ time.

Stage two of the algorithm is best performed using the union-find data structure [34]. Consider all the labels that are used in stage one and treat them as singleton sets. For every pair of labels recorded in stage one, perform a find operation to find the two sets that contain the labels currently, and do a union operation on the sets. At the end of stage two, all labels used for the same connected component will be grouped together in one set. This can be used to provide a unique label for each set (called the set label), and subsequently identify the set label for each label. The number of labels used is bounded by B. The amortized run time per operation using the union-find data structure is given by the inverse Ackermann's function [34], a constant (≤ 4) for all practical purposes. Therefore, the run time of stage two can be considered to be $O(B)$. Stage three of the algorithm can be carried out by another postorder traversal of the quadtree to replace the label of each black node with the corresponding set label in $O(B + W)$ time. Thus, the entire algorithm runs in $O(B + W)$ time.

19.5 Scientific Computing Applications

Region quadtrees are widely used in scientific computing applications. Most of the problems are three dimensional, prompting the development of octree based methods. In fact, some of the practical techniques and shortcuts presented earlier for storing region quadtrees and bit string representation of cells owe their origin to applications in scientific computing.

Octree are used in many ways in scientific computing applications. In many scientific computing applications, the behavior of a physical system is captured through either 1) a discretization of the space using a finite grid, followed by determining the quantities of interest at each of the grid cells, or 2) computing the interactions between a system of (real or virtual) particles which can be related to the behavior of the system. The former are known as grid-based methods and typically correspond to the solution of differential equations. The latter are known as particle-based methods and often correspond to the solution of integral equations. Both methods are typically iterative and the spatial information, such as the hierarchy of grid cells or the spatial distribution of the particles, often changes from iteration to iteration.

Examples of grid-based methods include finite element methods, finite difference methods, multigrid methods and adaptive mesh refinement methods. Many applications use a cell decomposition of space as the grid or grid hierarchy and the relevance of octrees for such applications is immediate. Algorithms for construction of octrees and methods for cell insertions and deletions are all directly applicable to such problems. It is also quite common to use other decompositions, especially for the finite-element method. For example, a decomposition of a surface into triangular elements is often used. In such cases, each basic element can be associated with a point (for example, the centroid of a triangle), and an octree can be built using the set of points. Information on the neighboring elements required by the application is then related to neighborhood information on the set of points.

Particle based methods include such simulation techniques as molecular dynamics, smoothed particle hydrodynamics, and N-body simulations. These methods require many of the spatial queries discussed in this chapter. For example, van der Waal forces between atoms in molecular dynamics fall off so rapidly with increasing distance (inversely proportional to the sixth power of the distance between atoms) that a cutoff radius is employed. In computing the forces acting on an individual atom, van der Waal forces are taken into account only for atoms that lie within the cutoff radius. The list of such atoms is typically referred to as the neighbor list. Computing neighbor lists for all atoms is just a group spherical region query, already discussed before. Similarly, k-nearest neighbors are useful in applications such as smoothed particle hydrodynamics. In this section, we further explore the use of octrees in scientific computing by presenting an optimal algorithm for the N-body problem.

19.5.1 The N-body Problem

The N-body problem is defined as follows: Given n bodies and their positions, where each pair of bodies interact with a force inversely proportional to the square of the distance between them, compute the force on each body due to all other bodies. A direct algorithm for computing all pairwise interactions requires $O(n^2)$ time. Greengard's fast multipole method [11], which uses an octree data structure, reduces this complexity by approximating the interaction between clusters of bodies instead of computing individual interactions. For each cell in the octree, the algorithm computes a multipole expansion and a local expansion. The multipole expansion at a cell C, denoted $\phi(C)$, is the effect of the bodies within C on distant bodies. The local expansion at C, denoted $\psi(C)$, is the effect of all distant bodies on bodies within C.

The N-body problem can be solved in $O(n)$ time using a compressed octree [2]. For two cells C and D which are not necessarily of the same size, define a predicate *well-separated*(C, D) to be true if D's multipole expansion converges at any point in C, and false otherwise. If two cells are not well-separated, they are *proximate*. Similarly, two nodes v_1 and v_2 in the compressed octree are said to be *well-separated* if and only if $S(v_1)$ and $S(v_2)$ are well-separated. Otherwise, we say that v_1 and v_2 are *proximate*.

For each node v in the compressed octree, the multipole expansion $\phi(v)$ and the local expansion $\psi(v)$ need to be computed. Both $\phi(v)$ and $\psi(v)$ are with respect to the cell $S(v)$. The multipole expansions can be computed by a simple bottom-up traversal in $O(n)$ time. For a leaf node, its multipole expansion is computed directly. At an internal node v, $\phi(v)$ is computed by aggregating the multipole expansions of the children of v.

The algorithm to compute the local expansions is given in Figure 19.9. The astute reader will notice that this is the same group query algorithm described in Section 19.3, reproduced here again with slightly different notation for convenience. The computations are done using a top-down traversal of the tree. To compute local expansion at node v, consider the set of nodes that are proximate to its parent, which is the *proximity set*, $P(parent(v))$. The proximity set of the root node contains only itself. Recursively decompose these nodes until each node is either 1) well-separated from v or 2) proximate to v and the length of the small cell of the node is no greater than the length of $S(v)$. The nodes satisfying the first condition form the *interaction set* of v, $I(v)$ and the nodes satisfying the second condition are in the proximity set of v, $P(v)$. In the algorithm, the set $E(v)$ contains the nodes that are yet to be processed. Local expansions are computed by combining parent's local expansion and the multipole expansions of the nodes in $I(v)$. For the leaf nodes, potential calculation is completed by using the direct method.

The following terminology is used in analyzing the run time of the algorithm. The set of cells that are proximate to C and having same length as C is called *the proximity set of C* and is defined by $P^{=}(C) = \{D \mid length(C) = length(D), \neg well\text{-}separated(C, D)\}$. The superscript "=" is used to indicate that cells of the same length are being considered. For node v, define the *proximity set P(v)* as the set of all nodes proximate to v and having the small cell no greater than and large cell no smaller than $S(v)$. More precisely, $P(v) = \{w \mid \neg well\text{-}separated(S(v), S(w)),\ length(S(w)) \leq length(S(v)) \leq length(L(w))\}$. The *interaction set I(v)* of v is defined as $I(v) = \{w \mid well\text{-}separated(S(v), S(w)),\ [\ w \in P(parent(v)) \vee \{\exists u \in P(parent(v)),\ w$ is a descendant of $u,\ \neg well\text{-}sep(v, parent(w)),\ length(S(v)) <$

Algorithm 2 Compute-Local-Exp (v)

```
I. Find the proximity set $P(v)$ and interaction set $I(v)$ for $v$
    $E(v) = P(parent(v))$
    $I(v) = \emptyset$; $P(v) = \emptyset$
    While $E(v) \neq \emptyset$ do
        Pick some $u \in E(v)$
        $E(v) = E(v) - \{u\}$
        If $well\text{-}separated(S(v), S(u))$
            $I(v) = I(v) \cup \{u\}$
        Else if $S(u) \subseteq S(v)$
            $P(v) = P(v) \cup \{u\}$
        Else $E(v) = E(v) \cup children(u)$
II. Calculate the local expansion at $v$
    Assign shifted $\psi(parent(v))$ to $\psi(v)$
    For each node $u \in I(v)$
        Add shifted $\phi(u)$ to $\psi(v)$
III. Calculate the local expansions at the children of $v$ with recursive calls
    For each child $u$ of $v$
        Compute-Local-Exp ($u$)
```

FIGURE 19.9: Algorithm for calculating local expansions of all nodes in the tree rooted at v.

$length(S(parent(w)))\}]\}$. We use $parent(w)$ to denote the parent of the node w.

The algorithm achieves $O(n)$ run time for any predicate *well-separated* that satisfies the following three conditions:

C1. The relation *well-separated* is symmetric for equal length cells, that is, $length(C) = length(D) \Rightarrow well\text{-}separated(C, D) = well\text{-}separated(D, C)$.
C2. For any cell C, $|P^{=}(C)|$ is bounded by a constant.
C3. If two cells C and D are not well-separated, any two cells C' and D' such that $C \subseteq C'$ and $D \subseteq D'$ are not well-separated as well.

These three conditions are respected by the various well-separatedness criteria used in N-body algorithms and in particular, Greengard's algorithm. In N-body methods, the well-separatedness decision is solely based on the geometry of the cells and their relative distance and is oblivious to the number of bodies or their distribution within the cells. Given two cells C and D of the same length, if D can be approximated with respect to C, then C can be approximated with respect to D as well, as stipulated by Condition C1. The size of the proximity sets of cells of the same length should be $O(1)$ as prescribed by Condition C2 in order that an $O(n)$ algorithm is possible. Otherwise, an input that requires processing the proximity sets of $\Omega(n)$ such cells can be constructed, making an $O(n)$ algorithm impossible. Condition C3 merely states that two cells C' and D' are not well-separated unless every subcell of C' is well-separated from every subcell of D'.

LEMMA 19.3 For any node v in the compressed octree, $|P(v)| = O(1)$.

Proof Consider any node v. Each $u \in P(v)$ can be associated with a unique cell $C \in P^{=}(S(v))$ such that $S(u) \subseteq C$. This is because any subcell of C which is not a subcell of $S(u)$ is not represented in the compressed octree. It follows that $|P(v)| \leq |P^{=}(S(v))| = O(1)$ (by Condition C2).

LEMMA 19.4 The sum of interaction set sizes over all nodes in the compressed octree is linear in the number of nodes in the compressed octree i.e., $\sum_v |I(v)| = O(n)$.

Proof Let v be a node in the compressed octree. Consider any $w \in I(v)$, either $w \in P(parent(v))$ or w is in the subtree rooted by a node $u \in P(parent(v))$. Thus,

$$\begin{aligned}\sum_v |I(v)| &= \sum_v |\{w \mid w \in I(v), w \in P(parent(v))\}| \\ &\quad + \sum_v |\{w \mid w \in I(v), w \notin P(parent(v))\}|.\end{aligned}$$

Consider these two summations separately. The bound for the first summation is easy; From Lemma 19.3, $|P(parent(v))| = O(1)$. So,

$$\sum_v |\{w \mid w \in I(v), w \in P(parent(v))\}| = \sum_v O(1) = O(n).$$

The second summation should be explored more carefully.

$$\sum_v |\{w \mid w \in I(v), w \notin P(parent(v))\}| = \sum_w |\{v \mid w \in I(v), w \notin P(parent(v))\}|$$

In what follows, we bound the size of the set $M(w) = \{v \mid w \in I(v),\ w \notin P(parent(v))\}$ for any node.

Since $w \notin P(parent(v))$, there exists a node $u \in P(parent(v))$ such that w is in the subtree rooted by u. Consider $parent(w)$: The node $parent(w)$ is either u or a node in the subtree rooted at u. In either case, $length(S(parent(w))) \leq length(S(parent(v)))$. Thus, for each $v \in M(w)$, there exists a cell C such that $S(v) \subseteq C \subseteq S(parent(v))$ and $length(S(parent(w))) = length(C)$. Further, since v and $parent(w)$ are not well-separated, C and $S(parent(w))$ are not well-separated as well by Condition C3. That is to say $S(parent(w)) \in P^{=}(C)$ and $C \in P^{=}(S(parent(w)))$ by Condition C1. By Condition C2, we know that $|P^{=}(S(parent(w)))| = O(1)$. Moreover, for each cell $C \in P^{=}(S(parent(w)))$, there are at most 2^d choices of v because $length(C) \leq length(S(parent(v)))$. As a result, $|M(w)| \leq 2^d \times O(1) = O(1)$ for any node w. Thus, $\sum_v |I(v)| = \sum_w |\{v \mid w \in I(v),$ $w \notin P(parent(v))\}| = \sum_w O(1) = O(n)$.

THEOREM 19.1 *Given a compressed octree for n bodies, the N-body problem can be solved in $O(n)$ time.*

Proof Computing the multipole expansion at a node takes constant time and the number of nodes in the compressed octree is $O(n)$. Thus, total time required for the multipole expansion calculation is $O(n)$. The nodes explored during the local expansion calculation at a node v are either in $P(v)$ or $I(v)$. In both cases, it takes constant time to process a node. By Lemma 19.3 and 19.4, the total size of both sets for all nodes in the compressed octree is bounded by $O(n)$. Thus, local expansion calculation takes $O(n)$ time. As a conclusion, the running time of the fast multipole algorithm on the compressed octree takes $O(n)$ time irrespective of the distribution of the bodies.

It is interesting to note that the same generic algorithmic framework is used for spherical region queries, all nearest neighbors, k-nearest neighbors and solving the N-body problem. While the proofs are different, the algorithm also provides optimal solution for k-nearest neighbors and the N-body problem.

While this chapter is focused on applications of quadtrees and octrees in image processing and scientific computing applications, they are used for many other types of spatial data and in the context of numerous other application domains. A detailed study of the design and analysis of quadtree based data structures and their applications can be found in [29, 30].

Acknowledgment

This work was supported, in part, by the National Science Foundation under grant ACI-0306512.

References

[1] S. Aluru, J. L. Gustafson, G. M. Prabhu, and F. E. Sevilgen. Distribution-independent hierarchical algorithms for the N-body problem. *The Journal of Supercomputing*, 12(4):303–323, 1998.

[2] S. Aluru and F. Sevilgen. Dynamic compressed hyperoctrees with application to the N-body problem. In *Springer-Verlag Lecture Notes in Computer Science*, volume 19, 1999.

[3] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.

[4] K. L. Clarkson. Fast algorithms for the all nearest neighbors problem. In *24th Annual Symposium on Foundations of Computer Science (FOCS '83)*, pages 226–232, 1982.

[5] C. Faloutsos. Gray codes for partial match and range queries. *IEEE Transactions on Software Engineering*, 14(10):1381–1393, 1988.

[6] R. A. Finkel and J. L. Bentley. Quad trees: a data structure for retrieval on composite key. *Acta Informatica*, 4(1):1–9, 1974.

[7] P. Flajolet, G. Gonnet, C. Puech, and J. M. Robson. The analysis of multidimensional searching in quad-trees. In Alok Aggarwal, editor, *Proceedings of the 2nd Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '90)*, pages 100–109, San Francisco, CA, USA, 1990.

[8] I. Gargantini. An effective way to represent quadtrees. *Communications of the ACM*, 25(12):905–910, 1982.

[9] I. Gargantini. Translation, rotation and superposition of linear quadtrees. *International Journal of Man-Machine Studies*, 18(3):253–263, 1983.

[10] N. K. Gautier, S. S. Iyengar, N. B. Lakhani, and M. Manohar. Space and time efficiency of the forest-of-quadtrees representation. *Image and Vision Computing*, 3:63–70, 1985.

[11] L. F. Greengard. *The Rapid Evaluation of Potential Fields in Particle Systems.* MIT Press, 1988.

[12] B. Hariharan, S. Aluru, and B. Shanker. A scalable parallel fast multipole method

for analysis of scattering from perfect electrically conducting surfaces. In *SC'2002 Conference CD, http://www.supercomp.org*, 2002.

[13] D. Hilbert. Uber die stegie abbildung einer linie auf flachenstuck. 38:459–460, 1891.

[14] G. M. Hunter and K. Steiglitz. Linear transformation of pictures represented by quad trees. *Computer Graphics and Image Processing*, 10(3):289–296, 1979.

[15] G. M. Hunter and K. Steiglitz. Operations on images using quad trees. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 1(2):143–153, 1979.

[16] G. M. Hunter. *Efficient computation and data structures for graphics.* PhD thesis, Princeton University, Princeton, N.J., USA, 1978.

[17] S. S. Iyengar and H. Gadagkar. Translation invariant data structure for 3-D binary images. *Pattern Recognition Letters*, 7:313–318, 1988.

[18] L. P. Jones and S. S. Iyengar. Space and time efficient virtual quadtrees. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 6:244–247, 1984.

[19] E. Kawaguchi, T. Endo, and J. Matsunaga. Depth-first expression viewed from digital picture processing. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 5(4):373–384, 1983.

[20] A. Klinger. *Optimizing methods in statistics*, chapter Patterns and search statistics, pages 303–337. 1971.

[21] A. Klinger and M. L. Rhodes. Organization and access of image data by areas. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 1(1):50–60, 1979.

[22] G. M. Morton. A computer oriented geodetic data base and a new technique in file sequencing. Technical report, IBM, Ottawa, Canada, 1966.

[23] J. A. Orenstein and T. H. Merrett. A class of data structures for associative searching. In *Proceedings of the Fourth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 181–190, 1984.

[24] M. H. Overmars and J. van Leeuwen. Dynamic multi-dimentional data structures based on quad- and $K-D$ trees. *Acta Informatica*, 17:267–285, 1982.

[25] B. Pagel, H. Six, H. Toben, and P. Widmayer. Toward an analysis of range query performance in spatial data structures. In *Proceedings of the Twelfth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 214–221, 1993.

[26] V. Raman and S. S. Iyengar. Properties and applications of forests of quadtrees for pictorial data representation. *BIT*, 23(4):472–486, 1983.

[27] H. Samet. Deletion in two-dimentional quad trees. *Communications of the ACM*, 23(12):703–710, 1980.

[28] H. Samet. Connected component labeling using quadtrees. *Journal of the ACM*, 28(3):487–501, 1981.

[29] H. Samet. *Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS.* Addison-Wesley, Reading, MA, 1989.

[30] H. Samet. *The Design and Analysis of Spatial Data Structures.* Addison-Wesley, Reading, MA, 1989.

[31] H. Samet, A. Rosenfeld, C. A. Shaffer, and R. E. Webber. Processing geographic data with quadtrees. In *Seventh Int. Conference on Pattern Recognition*, pages 212–215. IEEE Computer Society Press, 1984.

[32] D. S. Scott and S. S. Iyengar. A new data structure for efficient storing of images. *Pattern Recognition Letters*, 3:211–214, 1985.

[33] D. S. Scott and S. S. Iyengar. Tid – a translation invariant data structure for storing images. *Communications of the ACM*, 29(5):418–429, 1986.

[34] R. E. Tarjan. Efficiency of a good but not linear disjoint set union algorithm. *Journal of the ACM*, 22:215–225, 1975.

[35] P. M. Vaidya. An optimal algorithm for the All-Nearest-Neighbors problem. In *27th Annual Symposium on Foundations of Computer Science*, pages 117–122. IEEE, 1986.

[36] P. M. Vaidya. An $O(n \log n)$ algorithm for the All-Nearest-Neighbors problem. *Discrete and Computational Geometry*, 4:101–115, 1989.

[37] M. Warren and J. Salmon. A parallel hashed-octree N-body algorithm. In *Proceedings of Supercomputing '93*, 1993.

20

Binary Space Partitioning Trees

Bruce F. Naylor
University of Texas, Austin

20.1 Introduction

In most applications involving computation with 3D geometric models, manipulating objects and generating images of objects are crucial operations. Performing these operations requires determining for every frame of an animation the spatial relations between objects: how they might intersect each other, and how they may occlude each other. However, the objects, rather than being monolithic, are most often comprised of many pieces, such as by many polygons forming the faces of polyhedra. The number of pieces may be anywhere from the 100's to the 1,000,000's. To compute spatial relations between n polygons by brute force entails comparing every pair of polygons, and so would require $O(n^2)$. For large scenes comprised of 10^5 polygons, this would mean 10^{10} operations, which is much more than necessary.

The number of operations can be substantially reduced to anywhere from $O(n \log_2 n)$ when the objects interpenetrate (and so in our example reduced to 10^6), to as little as constant time, $O(1)$, when they are somewhat separated from each other. This can be accomplished by using Binary Space Partitioning Trees, also called BSP Trees. They provide a computational representation of space that simultaneously provides a search structure and a representation of geometry. The reduction in number of operations occurs because BSP Trees provide a kind of "spatial sorting". In fact, they are a generalization to dimensions > 1 of binary search trees, which have been widely used for representing sorted lists. Figure 20.1 below gives an introductory example showing how a binary tree of lines, instead of points, can be used to "sort" four geometric objects, as opposed to sorting symbolic objects such as names.

Constructing a BSP Tree representation of one or more polyhedral objects involves computing the spatial relations between polygonal faces once and encoding these relations in a binary tree (Figure 20.2). This tree can then be transformed and merged with other trees to very quickly compute the spatial relations (for visibility and intersections) between the polygons of two moving objects.

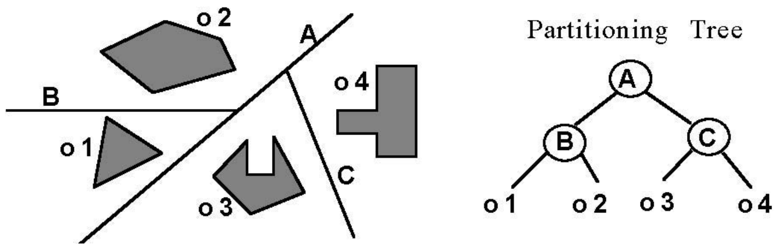


FIGURE 20.1: BSP Tree representation of inter-object spatial relations.

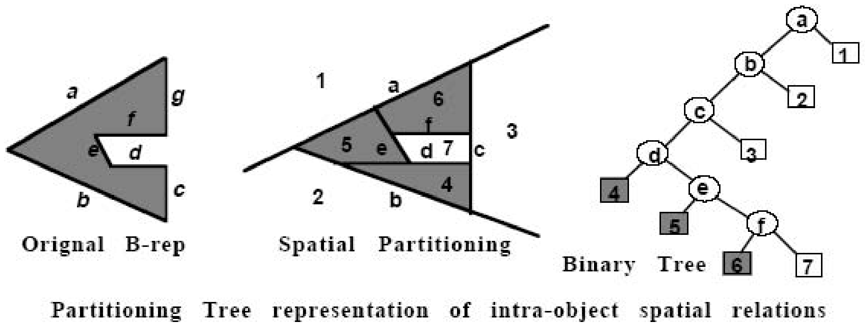


FIGURE 20.2: Partitioning Tree representation of intra-object spatial relations.

As long as the relations encoded by a tree remain valid, which for a rigid body is forever, one can reap the benefits of having generated this tree structure every time the tree is used in subsequent operations. The return on investment manifests itself as substantially faster algorithms for computing intersections and visibility orderings. And for animation and interactive applications, these savings can accrue over hundreds of thousands of frames. BSP Trees achieve an elegant solution to a number of important problems in geometric computation by exploiting two very simple properties occurring whenever a single plane separates (lies between) two or more objects: 1) any object on one side of the plane cannot intersect any object on the other side, 2) given a viewing position, objects on the same side as the viewer can have their images drawn on top of the images of objects on the opposite side (Painter's Algorithm). See Figure 20.3.

These properties can be made dimension independent if we use the term "hyperplane" to refer to planes in 3D, lines in 2D, and in general for d-space, to a $(d-1)$-dimensional subspace defined by a single linear equation. The only operation we will need for constructing BSP Trees is the partitioning of a convex region by a singe hyperplane into two child regions, both of which are also convex as a result (Figure 20.4).

BSP Trees exploit the properties of separating planes by using one very simple but powerful technique to represent any object or collection of objects: recursive subdivision by hyperplanes. A BSP Tree is the recording of this process of recursive subdivision in the form of a binary tree of hyperplanes. Since there is no restriction on what hyperplanes are used, polytopes (polyhedra, polygons, etc.) can be represented exactly. A BSP Tree is a program for performing intersections between the hyperplane's halfspaces and any other geometric entity. Since subdivision generates increasingly smaller regions of space, the order of the hyperplanes is chosen so that following a path deeper into the tree corresponds to adding more detail, yielding a multi-resolution representation. This leads to efficient

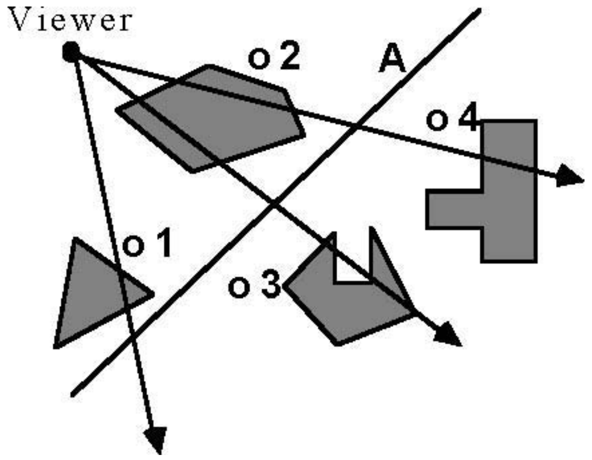


FIGURE 20.3: Plane Power: sorting objects w.r.t a hyperplane.

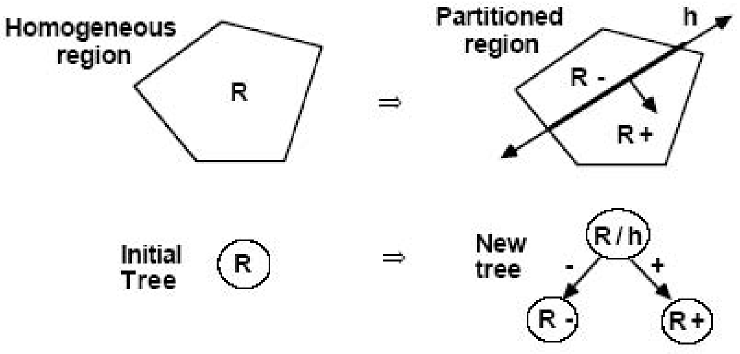


FIGURE 20.4: Elementary operation used to construct BSP Trees.

intersection computations. To determine visibility, all that is required is choosing at each tree node which of the two branches to draw first based solely on which branch contains the viewer. No other single representation of geometry inherently answers questions of intersection and visibility for a scene of 3D moving objects. And this is accomplished in a computationally efficient and parallelizable manner.

20.2 BSP Trees as a Multi-Dimensional Search Structure

Spatial search structures are based on the same ideas that were developed in Computer Science during the 60's and 70's to solve the problem of quickly processing large sets of symbolic data, as opposed to geometric data, such as lists of people's names. It was discovered that by first sorting a list of names alphabetically, and storing the sorted list in an array, one can find out whether some new name is already in the list in $\log_2 n$ operations using a binary search algorithm, instead of $n/2$ expected operations required by a sequential search. This is a good example of extracting structure (alphabetical order) existing in the list of names and exploiting that structure in subsequent operations (looking up a name) to reduce computation. However, if one wishes to permit additions and deletions of names while maintaining a sorted list, then a dynamic data structure is needed, i.e. one using pointers. One of the most common examples of such a data structure is the binary search tree.

A binary search tree (See also Chapter 3) is illustrated in Figure 20.5, where it is being used to represent a set of integers S = { 0, 1, 4, 5, 6, 8 } lying on the real line. We have included both the binary tree and the hierarchy of intervals represented by this tree. To find out whether a number/point is already in the tree, one inserts the point into the tree and follows the path corresponding to the sequence of nested intervals that contain the point. For a balanced tree, this process will take no more than $O(\log n)$ steps; for in fact, we have performed a binary search, but one using a tree instead of an array. Indeed, the tree itself encodes a portion of the search algorithm since it prescribes the order in which the search proceeds.

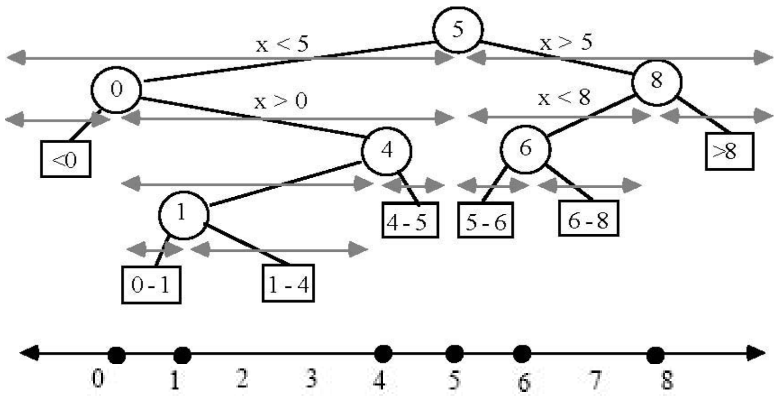


FIGURE 20.5: A binary search tree.

This now brings us back to BSP Trees, for as we said earlier, they are a generalization of binary search trees to dimensions > 1 (in 1D, they are essentially identical). In fact, constructing a BSP Tree can be thought of as a geometric version of Quick Sort. Modifications (insertions and deletions) are achieved by merging trees, analogous to merging sorted lists in Merge Sort. However, since points do not divide space for any dimension > 1, we must use hyperplanes instead of points by which to subdivide. Hyperplanes always partition a region into two halfspaces regardless of the dimension. In 1D, they look like points since they are also 0D sets; the one difference being the addition of a normal denoting the “greater than” side. In Figure 20.6, we show a restricted variety of BSP Trees that most clearly illustrates the generalization of binary search trees to higher dimensions. (You may want to call this a k-d tree, but the standard semantics of k-d trees does not include representing continuous sets of points, but rather finite sets of points.) BSP Trees are also a geometric variety of Decision Trees, which are commonly used for classification (e.g. biological taxonomies), and are widely used in machine learning. Decision trees have also been used for proving lower bounds, the most famous showing that sorting is $\Omega(n \log n)$. They are also the model of the popular “20 questions” game (I’m thinking of something and you have 20 yes/no question to guess what it is). For BSP Trees, the questions become “what side of a particular hyperplane does some piece of geometry lie”.

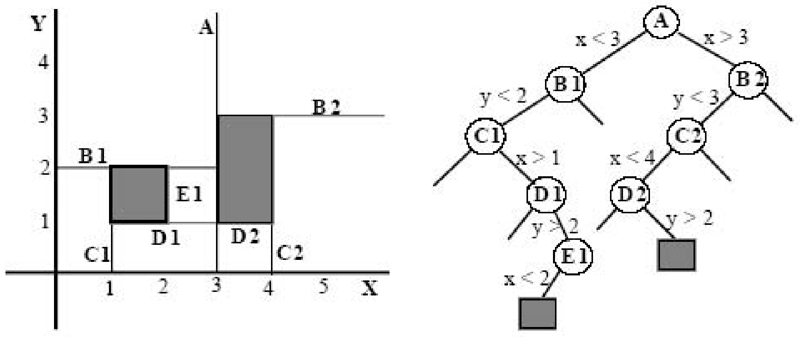


FIGURE 20.6: Extension of binary search trees to 2D as a BSP Tree.

20.3 Visibility Orderings

Visibility orderings are used in image synthesis for visible surface determination (hidden surface removal), shadow computations, ray tracing, beam tracing, and radiosity. For a given center of projection, such as the position of a viewer or of a light source, they provide an ordering of geometric entities, such as objects or faces of objects, consistent with the order in which any ray originating at the center might intersect the entities. Loosely speaking, a visibility ordering assigns a priority to each object or face so that closer objects have priority over objects further away. Any ray emanating from the center or projection that intersects two objects or faces, will always intersect the surface with higher priority first. The simplest use of visibility orderings is with the "Painter's Algorithm" for solving the hidden surface problem. Faces are drawn into a frame-buffer in far-to-near order (low-to-high priority), so that the image of nearer objects/polygons over-writes those of more distant ones.

A visibility ordering can be generated using a single hyperplane; however, each geometric entity or "object" (polyhedron, polygon, line, point) must lie completely on one side of the hyperplane, i.e. no objects are allowed to cross the hyperplane. This requirement can always be induced by partitioning objects by the desired hyperplane into two "halves". The objects on the side containing the viewer are said to have visibility priority over objects on the opposite side; that is, any ray emanating from the viewer that intersects two objects on opposite sides of the hyperplane will always intersect the near side object before it intersects the far side object. See Figures 20.7 and 20.8.

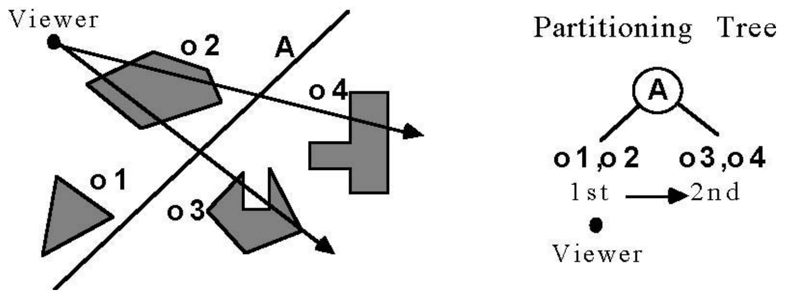


FIGURE 20.7: Left side has priority over right side.

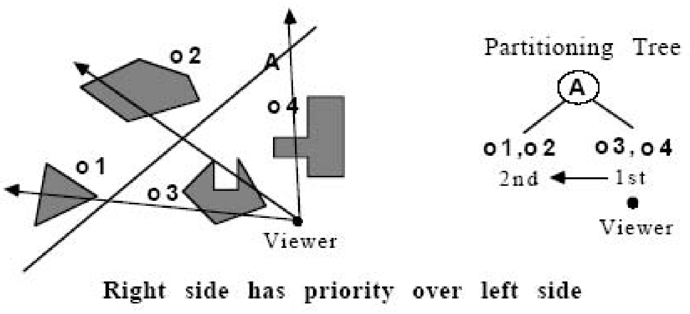


FIGURE 20.8: Right side has priority over left side.

20.3.1 Total Ordering of a Collection of Objects

A single hyperplane cannot order objects lying on the same side, and so cannot provide a total visibility ordering. Consequently, in order to exploit this idea, we must extend it somehow so that a visibility ordering for the entire set of objects can be generated. One way to do this would be to create a unique separating hyperplane for every pair of objects. However, for n objects this would require n^2 hyperplanes, which is too many.

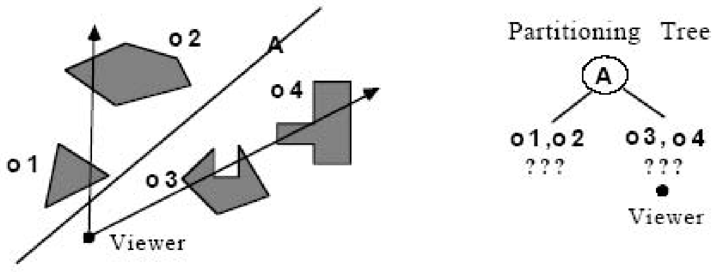


FIGURE 20.9: Separating objects with a hyperplane.

The required number of separating hyperplanes can be reduced to as little as n by using the geometric version of recursive subdivision (divide and conquer). If the subdivision is performed using hyperplanes whose position and orientation is unrestricted, then the result is a BSP Tree. The objects are first separated into two groups by some appropriately chosen hyperplane (Figure 20.9). Then each of the two groups is independently partitioned into two sub-groups (for a total now of 4 sub-groups). The recursive subdivision continues in a similar fashion until each object, or piece of an object, is in a separate cell of the partitioning. This process of partitioning space by hyperplanes is naturally represented as a binary tree (Figure 20.10).

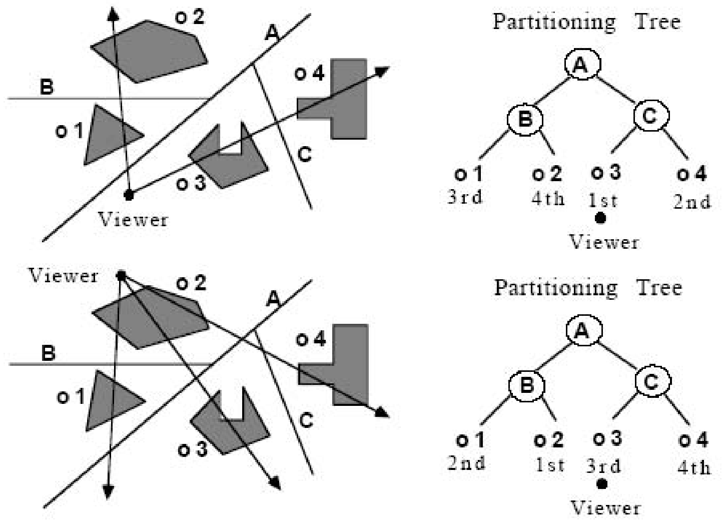


FIGURE 20.10: Binary tree representation of space partitioning.

20.3.2 Visibility Ordering as Tree Traversal

How can this tree be used to generate a visibility ordering on the collection of objects? For any given viewing position, we first determine on which side of the root hyperplane the viewer lies. From this we know that all objects in the near-side subtree have higher priority than all objects in the far-side subtree; and we have made this determination with only a constant amount of computation (in fact, only a dot product). We now need to order the near-side objects, followed by an ordering of the far-side objects. Since we have a recursively defined structure, any subtree has the same form computationally as the whole tree. Therefore, we simply apply this technique for ordering subtrees recursively, going left or right first at each node, depending upon which side of the node's hyperplane the viewer lies. This results in a traversal of the entire tree, in near-to-far order, using only $O(n)$ operations, which is optimal (this analysis is correct only if no objects have been split; otherwise it is $> n$).

20.3.3 Intra-Object Visibility

The schema we have just described is only for inter-object visibility, i.e. between individual objects. And only when the objects are both convex and separable by a hyperplane is the schema a complete method for determining visibility. To address the general unrestricted case, we need to solve intra-object visibility, i.e. correctly ordering the faces of a single object. BSP Trees can solve this problem as well. To accomplish this, we need to change our focus from convex cells containing objects to the idea of hyperplanes containing faces. Let us return to the analysis of visibility with respect to a hyperplane. If instead of ordering objects, we wish to order faces, we can exploit the fact that not only can faces lie on each side of a hyperplane as objects do, but they can also lie on the hyperplane itself. This gives us a 3-way ordering of: near $\rightarrow$ on $\rightarrow$ far (Figure 20.11).

If we choose hyperplanes by which to partition space that always contain a face of an

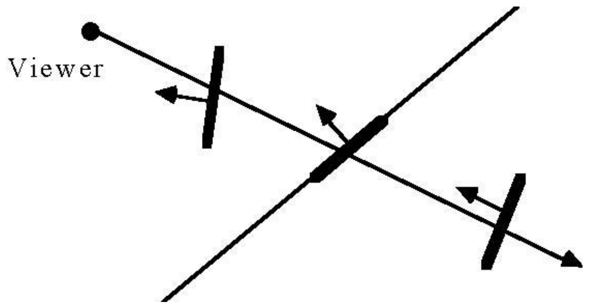


FIGURE 20.11: Ordering of polygons: near → on → far.

object, then we can build a BSP Tree by applying this schema recursively as before, until every face lies in some partitioning hyperplane contained in the tree. To generate a visibility ordering of the faces in this intra-object tree, we use the method above with one extension: faces lying on hyperplanes are included in the ordering, i.e. at each node, we generate the visibility ordering of near-subtree → on-faces → far-subtree. Using visibility orderings provides an alternative to z-buffer based algorithms. They obviate the need for computing and comparing z-values, which is very susceptible to numerical error because of the perspective projection. In addition, they eliminate the need for z-buffer memory itself, which can be substantial (80Mbytes) if used at a sub-pixel resolution of 4x4 to provide anti-aliasing. More importantly, visibility orderings permit unlimited use of transparency (non-refractive) with no additional computational effort, since the visibility ordering gives the correct order for compositing faces using alpha blending. And in addition, if a near-to-far ordering is used, then rendering completely occluded objects/faces can be eliminated, such as when a wall occludes the rest of a building, using a beam-tracing based algorithm.

20.4 BSP Tree as a Hierarchy of Regions

Another way to look at BSP Trees is to focus on the hierarchy of regions created by the recursive partitioning, instead of focusing on the hyperplanes themselves. This view helps us to see more easily how intersections are efficiently computed. The key idea is to think of a BSP Tree region as serving as a bounding volume: each node v corresponds to a convex volume that completely contains all the geometry represented by the subtree rooted at v. Therefore, if some other geometric entity, such as a point, ray, object, etc., is found to not intersect the bounding volume, then no intersection computations need be performed with any geometry within that volume.

Consider as an example a situation in which we are given some test point and we want to find which object if any this point lies in. Initially, we know only that the point lies somewhere space (Figure 20.12).

By comparing the location of the point with respect to the first partitioning hyperplane, we can find in which of the two regions (a.k.a. bounding volumes) the point lies. This eliminates half of the objects (Figure 20.13).

By continuing this process recursively, we are in effect using the regions as a hierarchy of bounding volumes, each bounding volume being a rough approximation of the geometry it bounds, to quickly narrow our search (Figure 20.14).

For a BSP Tree of a single object, this region-based (volumetric) view reveals how BSP Trees can provide a multi-resolution representation. As one descends a path of the tree, the

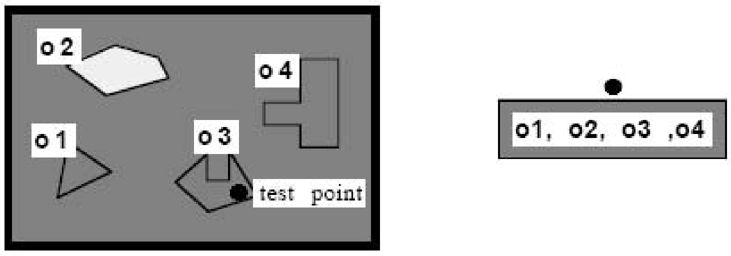


FIGURE 20.12: Point can lie anywhere.

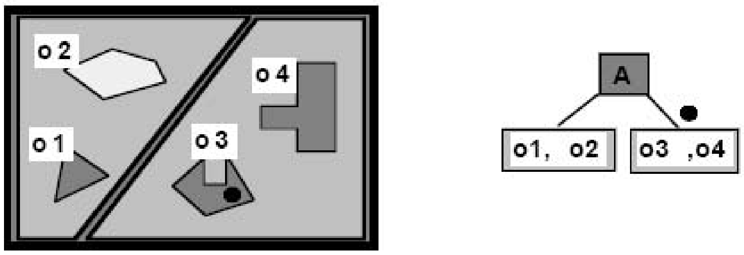


FIGURE 20.13: Point must lie to the right of the hyperplane.

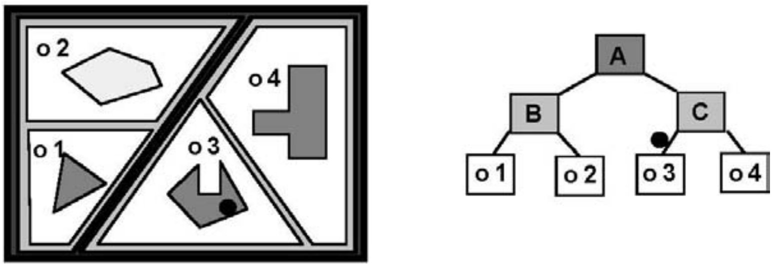


FIGURE 20.14: Point's location is narrowed down to one object.

regions decrease in size monotonically. For curved objects, the regions converge in the limit to the curve/surface. Truncating the tree produces an approximation, ala the Taylor series of approximations for functions (Figure 20.15).

20.4.1 Tree Merging

The spatial relations between two objects, each represented by a separate tree, can be determined efficiently by merging two trees. This is a fundamental operation that can be used to solve a number of geometric problems. These include set operations for CSG modeling as well as collision detection for dynamics. For rendering, merging all object-trees into a single model-tree determines inter-object visibility orderings; and the model-tree can be intersected with the view-volume to efficiently cull away off-screen portions of the scene and provide solid cutaways with the near clipping plane. In the case where objects are

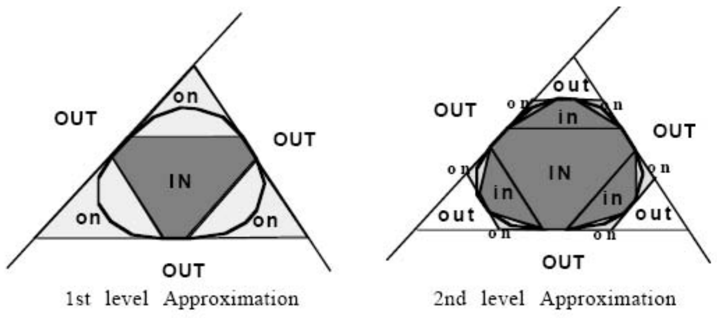


FIGURE 20.15: Multiresolution representation provided by BSP tree.

both transparent and interpenetrate, tree merging acts as a view independent geometric sorting of the object faces; each tree is used in a manner analogous to the way Merge Sort merges previously sorted lists to quickly created a new sorted list (in our case, a new tree). The model-tree can be rendered using ray tracing, radiosity, or polygon drawing using a far-to-near ordering with alpha blending for transparency. An even better alternative is multi-resolution beam-tracing, since entire occluded subtrees can be eliminated without visiting the contents of the subtree, and distance subtrees can be pruned to the desired resolution. Beam tracing can also be used to efficiently compute shadows.

All of this requires as a basic operation an algorithm for merging two trees. Tree merging is a recursive process, which proceeds down the trees in a multi-resolution fashion, going from low-res to high-res. It is easiest to understand in terms of merging a hierarchy of bounding volumes. As the process proceeds, pairs of tree regions, a.k.a. convex bounding volumes, one from each tree, are compared to determine whether they intersect or not. If they do not, the contents of the corresponding subtrees are never compared. This has the effect of "zooming in" on those regions of space where the surfaces of the two objects intersect (Figure 20.16).

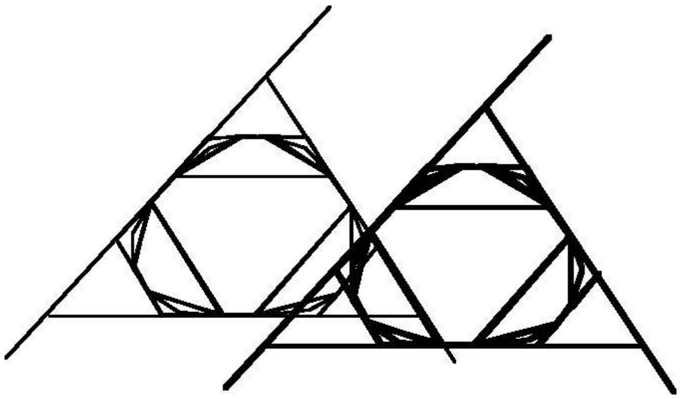


FIGURE 20.16: Merging BSP Trees.

The algorithm for tree merging is quite simple once you have a routine for partitioning a tree by a hyperplane into two trees. The process can be thought of in terms of inserting one tree into the other in a recursive manner. Given trees T1 and T2, at each node of T1 the hyperplane at that node is used to partition T2 into two "halves". Then each half is merged with the subtree of T1 lying on the same side of the hyperplane. (In actuality, the algorithm is symmetric w.r.t. the role of T1 and T2 so that at each recursive call, T1 can split T2 or T2 can split T1.)

```
Merge_Bspts : ( T1, T2 : Bspt ) -> Bspt
   Types
      BinaryPartitioner : { hyperplane, sub-hyperplane}
      PartitionedBspt : ( inNegHs, inPosHs : Bspt )

   Imports
      Merge_Tree_With_Cell : ( T1, T2 : Bspt ) -> Bspt User defined semantics.
      Partition_Bspt : ( Bspt, BinaryPartitioner ) -> PartitionedBspt

   Definition
      IF T1.is_a_cell OR T2.is_a_cell
      THEN
        VAL := Merge_Tree_With_Cell( T1, T2 )
      ELSE
        Partition_Bspt( T2, T1.binary_partitioner ) -> T2_partitioned
        VAL.neg_subtree := Merge_Bspts(T1.neg_subtree, T2_partitioned.inNegHs)
        VAL.pos_subtree:= Merge_Bspts(T1.pos_subtree, T2_partitioned.inPosHs )
      END
   RETURN} VAL
   END Merge_Bspts
```

While tree merging is easiest to understand in terms of comparing bounding volumes, the actual mechanism uses *sub-hyperplanes,* which is more efficient. A sub-hyperplane is created whenever a region is partitioned by a hyperplane, and it is just the subset of the hyperplane lying within that region. In fact, all of the illustrations of trees we have used are drawings of sub-hyperplanes. In 3D, these are convex polygons, and they separate the two child regions of an internal node. Tree merging uses sub-hyperplanes to simultaneously determine the spatial relations of four regions, two from each tree, by comparing the two sub-hyperplanes at the root of each tree. For 3D, this is computed using two applications of convex-polygon clipping to a plane, and there are three possible outcomes: intersecting, non-intersecting and coincident (Figure 20.17). This is the only overtly geometric computation in tree merging; everything else is data structure manipulation.

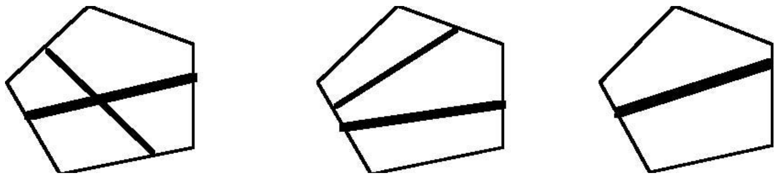


FIGURE 20.17: Three cases (intersecting, non-intersecting, coincident) when comparing sub-hyperplanes during tree merging.

20.4.2 Good BSP Trees

For any given set, there exist an arbitrary number of different BSP Trees that can represent that set. This is analogous to there being many different programs for computing the same function, since a BSP Tree may in fact be interpreted as a computation graph specifying a particular search of space. Similarly, not all programs/algorithms are equally efficient, and not all searches/trees are equally efficient. Thus the question arises as to what constitutes a good BSP Tree. The answer is a tree that represents the set as a sequence of approximations. This provides a multi-resolution representation. By pruning the tree at various depths, different approximations of the set can be created. Each pruned subtree is replaced with a cell containing a low degree polynomial approximation of the set represented by the subtree (Figures 20.18 and 20.19).

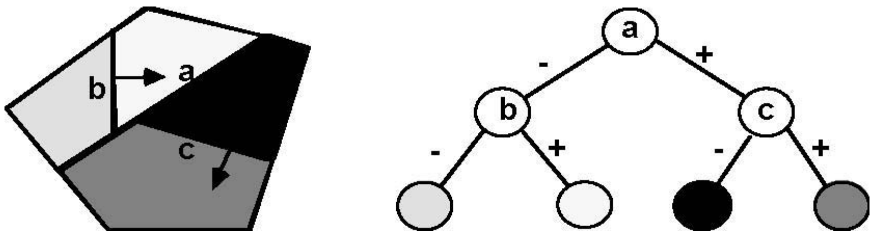


FIGURE 20.18: Before Pruning.

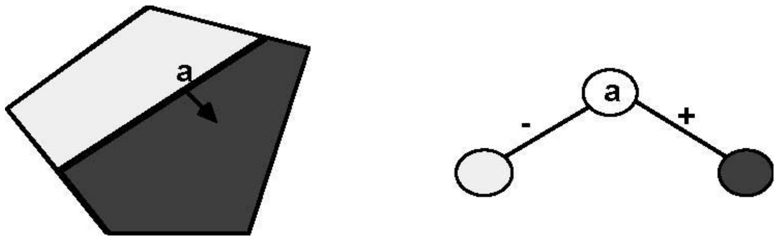


FIGURE 20.19: After Pruning.

In Figure 20.20, we show two quite different ways to represent a convex polygon, only the second of which employs the sequence of approximations idea. The tree on the left subdivides space using lines radiating from the polygonal center, splitting the number of faces in half at each step of the recursive subdivision. The hyperplanes containing the polygonal edges are chosen only when the number of faces equals one, and so are last along any path. If the number of polygonal edges is n, then the tree is of size $O(n)$ and of depth $O(\log n)$. In contrast, the tree on the right uses the idea of a sequence of approximations. The first three partitioning hyperplanes form a first approximation to the exterior while the next three form a first approximation to the interior. This divides the set of edges into three sets. For each of these, we choose the hyperplane of the middle face by which to partition, and by doing so refine our representation of the exterior. Two additional hyperplanes refine the interior and divide the remaining set of edges into two nearly equal sized sets. This process precedes recursively until all edges are in partitioning hyperplanes. Now, this tree is also of size $O(n)$ and depth $O(\log n)$, and thus the worst case, say for point classification, is the same for both trees. Yet they appear to be quite different.

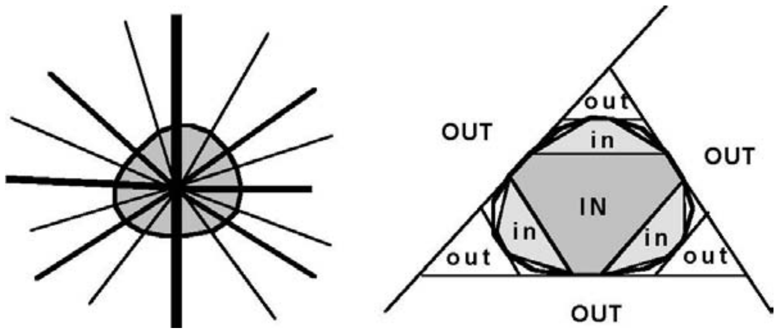


FIGURE 20.20: Illustration of bad vs. good trees.

This apparent qualitative difference can be made quantitative by, for example, considering the expected case for point classification. With the first tree, all cells are at depth $\log n$, so the expected case is the same as the worst case regardless of the sample space from which a point is chosen. However, with the second tree, the top three out-cells would typically constitute most of the sample space, and so a point would often be classified as OUT by, on average, two point-hyperplane tests. Thus the expected case would converge to $O(1)$ as the ratio of polygon-area/sample-area approaches 0. For line classification, the two trees differ not only in the expected case but also in the worst case: $O(n)$ vs. $O(\log n)$. For merging two trees the difference is $O(n^2)$ vs. $O(n \log n)$. This reduces even further to $O(\log n)$ when the objects are only contacting each other, rather overlapping, as is the case for collision detection. However, there are worst case "basket weaving" examples that do require $O(n^2)$ operations. These are geometric versions of the Cartesian product, as for example when a checkerboard is constructed from n horizontal strips and n vertical strips to produce $n \times n$ squares. These examples, however, violate the Principle of Locality: that geometric features are local not global features. For almost all geometric models of physical objects, the geometric features are local features. Spatial partitioning schemes can accelerate computations only when the features are in fact local, otherwise there is no significant subset of space that can be eliminated from consideration. The key to a quantitative evaluation, and also generation, of BSP Trees is to use expected case models, instead of worst-case analysis. Good trees are ones that have low expected cost for the operations and distributions of input of interest. This means, roughly, that high probability regions can be reached with low cost, i.e. they have short paths from the root to the corresponding node, and similarly low probability regions should have longer paths. This is exactly the same idea used in Huffman codes. For geometric computation, the probability of some geometric entity, such as a point, line segment, plane, etc., lying in some arbitrary region is typically correlated positively to the size of the region: the larger the region the greater the probability that a randomly chosen geometric entity will intersect that region. To compute the expected cost of a particular operation for a given tree, we need to know at each branch in the tree the probability of taking the left branch, p, and the probability of taking the right branch p^+. If we assign a unit cost to the partitioning operation, then we can compute the expected cost exactly, given the branch probabilities, using the following recurrence relation:

```
$E_{cost}[T]$=
    IF T is a cell
    THEN 0
    ELSE $1 + p^- * E_{cost}[T^-] + p^+ * E_{cost}[T^+]$
```

This formula does not directly express any dependency upon a particular operation; those characteristics are encoded in the two probabilities $p^-$and p^+. Once a model for these is specified, the expected cost for a particular operation can be computed for any tree.

As an example, consider point classification in which a random point is chosen from a uniform distribution over some initial region R. For a tree region of r with child regions r^+ and r^-, we need the conditional probability of the point lying in r^+ and r^-, given that it lies in r. For a uniform distribution, this is determined by the sizes of the two child-regions relative to their parent:

$p^+ = \mathrm{vol}(r^+)/\mathrm{vol}(r)$
$p^- = \mathrm{vol}(r^-)/\mathrm{vol}(r)$

Similar models have been developed for line, ray and plane classification. Below we describe how to use these to build good trees.

20.4.3 Converting B-reps to Trees

Since humans do not see physical objects in terms of binary trees, it is important to know how such a tree be constructed from something that is more intuitive. The most common method is to convert a boundary representation, which corresponds more closely to how humans see the world, into a tree. In order for a BSP Tree to represent a solid object, each cell of the tree must be classified as being either entirely inside or outside of the object; thus, each leaf node corresponds to either an in-cell or an out-cell. The boundary of the set then lies between in-cells and out-cells; and since the cells are bounded by the partitioning hyperplanes, it is necessary for the entire boundary to lie in the partitioning hyperplanes (Figure 20.21).

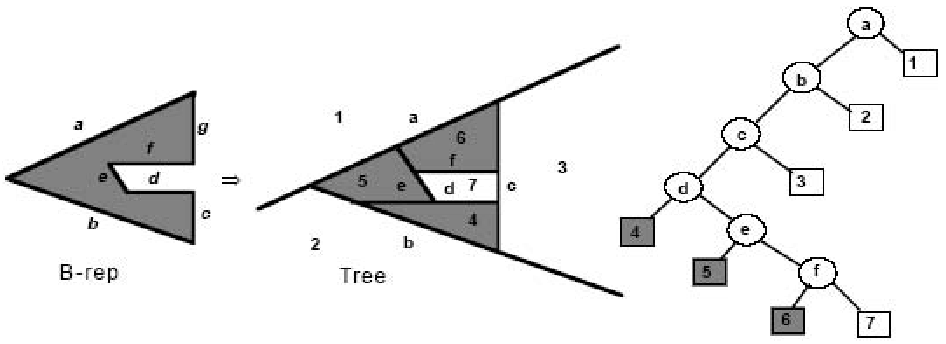


FIGURE 20.21: B-rep and Tree representation of a polygon.

Therefore, we can convert from a b-rep to a tree simply by using all of the face hyperplanes as partitioning hyperplanes. The face hyperplanes can be chosen in any order and the resulting tree will always generate a convex decomposition of the interior and the exterior. If the hyperplane normals of the b-rep faces are consistently oriented to point to the exterior, then all left leaves will be in-cells and all right leaves will be out-cells. The following algorithm summarizes the process.

```
Brep_to_Bspt: Brep b − > Bspt T
IF b == NULL
THEN
  T = if a left-leaf then an in-cell else an out-cell
ELSE
  h = Choose_Hyperplane(b)
  { $b^+, b^-, b^0$ } = Partition_Brep(b,h)
  T.faces = $b^0$
  T.pos_subtree = Brep_to_Bspt($b^+$)
  T.neg_subtree = Brep_to_Bspt($b^-$)
END
```

However, this does not tell us in what order to choose the hyperplanes so as to produce the best trees. Since the only known method for finding the optimal tree is by exhaustive enumeration, and there are at least n! trees given n unique face hyperplanes, we must employ heuristics. In 3D, we use both the face planes as candidate partitioning hyperplanes, as well as planes that go through face vertices and have predetermined directions, such as aligned with the coordinates axes. Given any candidate hyperplane, we can try to predict how effective it will be using expected case models; that is, we can estimate the expected cost of a subtree should we choose this candidate to be at its root. We will then choose the least cost candidate. Given a region r containing boundary b which we are going to partition by a candidate h, we can compute exactly p^+ and $p^-$for a given operation, as well as the size of b^+ and b^-. However, we can only estimate $E_{cost}[T^+]$ and $E_{cost}[T^-]$. The estimators for these values can depend only upon a few simple properties such as number of faces in each halfspace, how many faces would be split by this hyperplane, and how many faces lie on the hyperplane (or area of such faces). Currently, we use $|b^+|^n$ for $E_{cost}[T^+]$, where n typically varies between .8 and .95, and similarly for $E_{cost}[T^-]$. We also include a small penalty for splitting a face by increasing its contribution to b^+ and b^- from 1.0 to somewhere between 1.25 and 1.75, depending upon the object. We also favor candidates containing larger surface area, both in our heuristic evaluation and by first sorting the faces by surface area and considering only the planes of the top k faces as candidates. One interesting consequence of using expected case models is that choosing the candidate that attempts to balance the tree is usually not the best; instead the model prefers candidates that place small amounts of geometry in large regions, since this will result in high probability and low cost subtrees, and similarly large amounts of geometry in small regions. Balanced is optimal only when the geometry is uniformly distributed, which is rarely the case (Figure 20.22). More importantly, minimizing expected costs produces trees that represents the object as a sequence of approximations, and so in a multi-resolution fashion.

20.4.4 Boundary Representations vs. BSP Trees

Boundary Representations and BSP Trees may be viewed as complementary representations expressing difference aspects of geometry, the former being topological, the later expressing hierarchical set membership. B-reps are well suited for interactive specification of geometry, expressing topological deformations, and scan-conversion. BSP Trees are well suited for intersection and visibility calculations. Their relationship is probably more akin to the capacitor vs. inductor, than the tube vs. transistor.

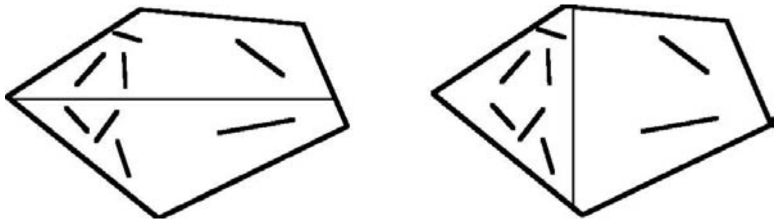


FIGURE 20.22: Balanced is not optimal for non-uniform distributions.

The most often asked question is what is the size of a BSP Tree representation of a polyhedron vs. the size of its boundary representation. This, of course, ignores the fact that expected cost, measured over the suite of operations for which the representation will be used, is the appropriate metric. Also, boundary representations must be supplemented by other devices, such as octrees, bounding volumes hierarchies, and z-buffers, in order to achieve an efficient system; and so the cost of creating and maintaining these structure should be brought into the equation. However, given the intrinsic methodological difficulties in performing a compelling empirical comparison, we will close with a few examples giving the original number of b-rep faces and the resulting tree using our currently implemented tree construction machinery. The first ratio is number-of-tree-faces/number-of-brep-faces. The second ratio is number-of-tree-nodes/number-of-brep-faces, where number-of-tree-nodes is the number of internal nodes. The last column is the expected cost in terms of point, line and plane classification, respectively, in percentage of the total number of internal nodes, and where the sample space was a bounding box 1.1 times the minimum axis-aligned bounding box. These numbers are pessimistic since typical sample spaces would be much larger than an object's bounding box. Also, the heuristics are controlled by 4 parameters, and these numbers were generated, with some exceptions, without a search of the parameter space but rather using default parameters. There are also quite a number of ways to improve the conversion process, so it should be possible to do even better.

Data Set	# brep faces	# tree faces	faces ratio	# tree nodes	faces/nodes ratio	Point E[T]	Line E[T]	Plane E[T]
Hang glider man	189	406	2.14	390	2.06	1.7	3.4	21.4
Space shuttle	575	1,006	1.75	927	1.61	1.2	2.5	13.2
Human head 1	927	1,095	1.21	1,156	1.24	1.4	4.4	25.0
Human head 2	2,566	5,180	2.01	5,104	1.99	0.2	0.8	9.1
Allosauros	4,072	9,725	2.38	9,914	2.43	NA	NA	NA
Lower Manhattan	4,532	5,510	1.22	4,273	0.94	0.3	0.6	10.5
Berkeley CS Bldg.	9,129	9,874	1.08	4,148	0.45	0.4	1.3	14.6
Dungeon	14,061	20,328	1.44	15,732	1.12	0.1	0.1	1.7
Honda Accord	26,033	51,730	1.98	42,965	1.65	NA	NA	NA
West Point Terrain	29,400	9,208	0.31	7,636	0.26	0.1	0.3	4.2
US Destroyer	45,802	91.928	2.00	65,846	1.43	NA	NA	NA

Bibliography

1. Sandra H. Bloomberg, A Representation of Solid Objects for Performing Boolean Operations, U.N.C. Computer Science Technical Report 86-006 (1986).
2. W. Thibault and B. Naylor, Set Operations On Polyhedra Using Binary Space BSP Trees, *Computer Graphics* Vol. 21(4), pp. 153-162, (July 1987).
3. Bruce F. Naylor, John Amanatides and William C. Thibault, Merging BSP Trees Yields Polyhedral Set Operations, *Computer Graphics* Vol. 24(4), pp. 115-124, (August 1990).

4. Bruce F. Naylor, Binary Space BSP Trees as an Alternative Representation of Polytopes, *Computer Aided Design*, Vol. 22(4), (May 1990).
5. Bruce F. Naylor, SCULPT: an Interactive Solid Modeling Tool, *Proceedings of Graphics Interface* (May 1990).
6. Enric Torres, Optimization of the Binary Space Partition Algorithm (BSP) for the Visualization of Dynamic Scenes, *Eurographics '90* (Sept. 1990).
7. Insung Ihm and Bruce Naylor, Piecewise Linear Approximations of Curves with Applications, *Proceedings of Computer Graphics International '91*, Springer-Verlag (June 1991).
8. George Vanecek, Brep-index: a multi-dimensional space partitioning tree, *Proceedings of the first ACM symposium on Solid modeling foundations and CAD/CAM applications*, (May 1991).
9. Bruce F. Naylor, *Interactive Solid Modeling Via BSP Trees*, *Proceeding of Graphics Interface*, pp 11-18, (May 1992).
10. Y. Chrysanthou and M. Slater, Computing Dynamic Changes to BSP Trees, *Eurographics '92*, 11(3), pp. 321-332.
11. Sigal Ar, Gil Montag, Ayellet Tal , Collision Detection and Augmented Reality: Deferred, Self-Organizing BSP Trees", *Computer Graphics Forum,* Volume 21, Issue 3 (September 2002).
12. R. A. Schumacker, R. Brand, M. Gilliland, and W. Sharp, Study for Applying Computer-Generated Images to Visual Simulation, AFHRL-TR-69-14, U.S. Air Force Human Resources Laboratory (1969).
13. Ivan E. Sutherland, Polygon Sorting by Subdivision: a Solution to the Hidden-Surface Problem, unpublished manuscript, (October 1973).
14. H. Fuchs, Z. Kedem, and B. Naylor, On Visible Surface Generation by a Priori Tree Structures, *Computer Graphics* Vol. 14(3), pp. 124-133, (June 1980).
15. Henry Fuchs, Gregory Abrams and Eric Grant, Near Real-Time Shaded Display of Rigid Objects, *Computer Graphics* Vol. 17(3), pp. 65-72, (July 1983).
16. Bruce F. Naylor and William C. Thibault, Application of BSP Trees to Ray-Tracing and CSG Evaluation, Technical Report GIT-ICS 86/03, School of Information and Computer Science, Georgia Institute of Technology, Atlanta, Georgia 30332 (February 1986).
17. Norman Chin and Steve Feiner, Near Real-Time Shadow Generation Using BSP Trees, *Computer Graphics* Vol. 23(3), pp. 99-106, (July 1989).
18. A.T. Campbell and Donald S. Fussell, Adaptive Mesh Generation for Global Diffuse Illumination, *Computer Graphics* Vol. 24(4), pp. 155-164, (August 1990).
19. A.T. Campbell, Modeling Global Diffuse for Image Synthesis, Ph.D. Dissertation, Department of Computer Science, University of Texas at Austin, (1991).
20. Dan Gordon and Shuhong Chen, *Front-to-Back Display of BSP Trees*, *IEEE Computer Graphics & Applications*, pp. 79-85, (September 1991).
21. Norman Chin and Steve Feiner, Fast Object-Precision Shadow Generation for Area Light Sources Using BSP Trees, *Symp. on 3D Interactive Graphics*, (March 1992).
22. Bruce F. Naylor, BSP Tree Image Representation and Generation from 3D Geometric Models, Proceedings of *Graphics Interface* (May 1992).
23. Dani Lischinski, Filippo Tampieri and Donald Greenburg, Discountinuity Mesh-

ing for Accurate Radiosity, *IEEE Computer Graphics & Applications* 12(6), pp. 25-39, (November 1992).

24. Seth Teller and Pat Hanrahan, Global Visibility Algorithms for Illumination Computations, *Computer Graphics* Vol. 27, pp. 239-246, (August 1993).
25. Yiorgos Chrysanthou, Shadow Computation for 3D Interaction and Animation, Ph.D. Thesis, Computer Science, University of London, (1996).
26. Mel Slater and Yiorgos Chrysanthou, View volume culling using a probabilistic caching scheme, *Proceedings of the ACM symposium on virtual reality software and technology*, (Sept. 1997), pp. 71-77.
27. Zhigeng Pan, Zhiliang Tao, Chiyi Cheng, Jiaoying Shi, A new BSP tree framework incorporating dynamic LoD models, *Proceedings of the ACM symposium on Virtual reality software and technology* (2000), pp. 134–141.
28. Hayder Rahda, Riccardo Leonardi, Martin Vetterli and Bruce Naylor, Binary Space BSP Tree Representation of Images, *Visual Communications and Image Representation*, Vol. 2(3), pp. 201-221, (Sept. 1991).
29. K.R. Subramanian and Bruce Naylor, Representing Medical images with BSP Trees, *Proceeding of Visualization '92*, (Oct. 1992).
30. Hayder M. Sadik Radha, Efficient Image Representation Using Binary Space BSP Trees, Ph.D. dissertation, CU/CTR/TR 343-93-23, Columbia University, (1993).
31. K.R. Subramanian and Bruce F. Naylor, Converting Discrete Images to Partitioning Trees, *IEEE Transactions on Visualization & Computer Graphics*, 3(3), 1997.
32. Ajay Rajkumar, Farid Fesulin, Bruce Naylor and Lois Rogers, Prediction RF Coverage in Large Environments using Ray-Beam Tracing and Partitioning Tree Represented Geometry, *ACM WINET*, 2(2), (1996).
33. Thomas Funkhouser, Ingrid Carlbom, Gary Elko, Gopal Pingali, Mohan Sondhi, Jim West, A beam tracing approach to acoustic modeling for interactive virtual environments, *Siggraph '98,* (July 1998).
34. Michael O. Rabin, Proving Simultaneous Positivity of Linear Forms, *Journal of Computer and Systems Science*, v6, pp. 639-650 (1991).
35. E. M. Reingold, On the Optimality of some Set Operations, *Journal of the ACM*, Vol. 19, pp. 649-659 (1972).
36. Bruce F. Naylor, A Priori Based Techniques for Determining Visibility Priority for 3-D Scenes, Ph.D. Thesis, University of Texas at Dallas (May 1981).
37. M. S. Paterson and F. F. Yao, Efficient Binary Space Partitions for Hidden-Surface Removal and Solid Modeling, *Discrete & Computational Geometry*, v5, pp. 485-503, 1990.
38. M. S. Paterson and F. F. Yao, Optimal Binary Space Partitions for Orthogonal Objects, *Journal of Algorithms*, v5, pp. 99-113, 1992.
39. Bruce F. Naylor, Constructing Good BSP Trees, *Graphics Interface '93*, Toronto CA, pp. 181-191, (May 1993).
40. Mark de Berg, Marko M. de Groot and Mark Overmars, Perfect Binary Space Partitions, *Canadian Conference on Computational Geometry*, 1993.
41. Pankaj K. Agarwal, Leonidas J. Guibas, T. M. Murali, Jeffrey Scott Vitter, Cylindrical and kinetic binary space partitions , *Proceedings of the thirteenth annual symposium on Computational Geometry, (*August 1997).

42. Joao Comba, Kinetic Vertical Decomposition Trees , Ph.D. Thesis, Computer Science Department, Stanford University, (1999).
43. Mark de Berg, Joao Comba, Leonidas J. Guibas, A segment-tree based kinetic BSP, *Proceedings of the seventeenth annual symposium on computational geometry*, (June 2001).
44. Sunil Arya, Binary space partitions for axis-parallel line segments: size-height tradeoffs, *Information Processing Letters*, 84[4] (Nov. 2002), pp. 201-206.
45. Csaba D. Tóth, Binary Space Partitions for Line Segments with a Limited Number of Directions, *SIAM J. on Computing*, 32(2), 2003, pp. 307-325.

21 R-trees

Scott Leutenegger
University of Denver

Mario A. Lopez
University of Denver

21.1 Introduction

Spatial database management systems must be able to store and process large amounts of disk-resident spatial data. Multidimensional data support is needed in many fields including geographic information systems (GIS), computer aided design (CAD), and medical, multimedia, and scientific databases. Spatial data operations that need to be supported include spatial joins and various types of queries such as intersection, containment, topological and proximity queries. The challenge, and primary performance objective, for applications dealing with disk-resident data is to minimize the number of disk retrievals, or I/Os, needed to answer a query. Main memory data structures are designed to reduce computation time rather than I/O, and hence are not directly applicable to a disk based environment.

Just as the B-tree [13] (Chapter 15) and its variants were proposed to optimize I/O while processing single dimensional disk resident data, the original R-tree [23] and later variants have been proposed to index disk resident multidimensional data efficiently.

R-trees are very versatile, in the sense that they can be used with little or no modification to tackle many different problems related to spatial data. Asymptotically better data structures exist for specific instances of these problems, but the solution of choice is different for different types of inputs or queries. Thus, one of the main advantages of R-trees is that the same data structure can be used to handle different problems on arbitrary types of multidimensional data.

In this chapter we describe the original R-tree proposal and some of its variants. We analyze the efficiency of various operations and examine various performance models. We also describe R-tree based algorithms for additional operations, such as proximity queries and spatial joins. There are many more applications, variants and issues related to R-trees than we can possibly cover in one chapter. Some of the ones we do not cover include parallel and distributed R-trees [9, 27, 46, 55], variants for high dimensional [5, 30, 67] and for spatio-temporal data [31, 48, 49, 53, 54, 62, 63], and concurrency [33, 41]. See Chapter 47 for more on concurrent data structures. Other data structures with similar functionality, such as range, quad and k-d trees, are covered in other chapters of Part IV of this handbook. Some of these (see Chapter 27) are specifically designed for disk-resident data.

21.2 Basic Concepts

R-trees were first introduced in [23]. An R-tree is a hierarchical data structure derived from the B$^+$-tree and originally designed to perform intersection queries efficiently. The tree stores a collection of d-dimensional points or rectangles which can change in time via insertions and deletions. Other object types, such as polygons or polyhedra, can be handled by storing their minimum bounding rectangles (MBRs) and performing additional tests to eliminate false hits. A false hit happens when the query object intersects the MBR of a data object but does not intersect the object itself. In the sequel we talk about rectangles only, with the understanding that a point is simply a degenerate rectangle. We use the terms MBR and bounding box, interchangeably. In our context, the d-dimensional rectangles are "upright", i.e., each rectangle is the Cartesian product of d one-dimensional intervals: $[l_1, h_1] \times \ldots \times [l_d, h_d]$. Thus, $2d$ values are used to specify a rectangle.

Each node of the tree stores a maximum of B entries. With the exception of the root, each node also stores a minimum of $b \leq B/2$ entries. This constraint guarantees a space utilization of at least b/B. Each entry E consists of a rectangle r and a pointer p_r. As with B$^+$-trees all input values are stored at the leaves. Thus, at the leaf level, r is the bounding box of an actual object pointed to by p_r. At internal nodes, r is the bounding box of all rectangles stored in the subtree pointed to by p_r.

A downward path in the tree corresponds to a sequence of nested rectangles. All leaf nodes occur at the same level (i.e., have the same depth), even after arbitrary sequences of updates. This guarantees that the height of the tree is $O(\log_b n)$, where n is the number of input rectangles. Notice that MBRs at the same level may overlap, even if the input rectangles are disjoint.

Figure 21.1 illustrates an R-tree with 3 levels (the root is at level 0) and a maximum of $B = 4$ rectangles per node. The 64 small dark data rectangles are grouped into 16 leaf level nodes, numbered 1 to 16. The bounding box of the set of rectangles stored at the same node is one of the rectangles stored at the parent of the node. In our example, the MBRs of leaf level nodes 1 through 4 are placed in node 17, in level 1. The root node contains the MBRs of the four level 1 nodes: 17, 18, 19, and 20.

Intersection queries

To perform an intersection query Q, all rectangles that intersect the query region must be retrieved and examined (regardless of whether they are stored in an internal node or a leaf node). This retrieval is accomplished by using a simple recursive procedure that starts at the root node and which may follow *multiple* paths down the tree. In the worst case, all nodes may need to be retrieved, even though some of the input data need not be reported. A node is processed by first identifying all the rectangles stored at that node which intersect Q. If the node is an internal node, the subtrees corresponding to the identified rectangles are searched recursively. Otherwise, the node is a leaf node and the retrieved rectangles (or the data objects themselves) are simply reported.

For illustration, consider the query Q in the example of Figure 21.1. After examining the root node, we determine that nodes 19 and 20 of level 1 must be searched. The search then proceeds with each of these nodes. Since the query region does not intersect any of the MBRs stored in node 19, this sub-query is terminated. While processing the other sub-query, it is determined that Q intersects the MBR corresponding to node 13 and this node

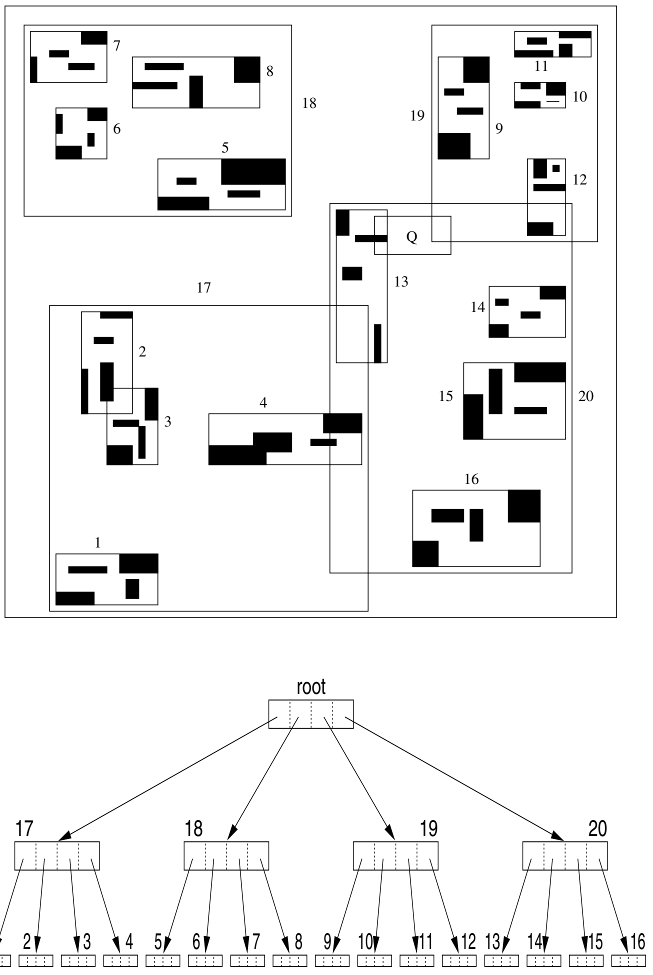


FIGURE 21.1: A sample R-tree using $B = 4$. Input rectangles are shown solid.

is retrieved. Upon checking the rectangles in node 13, the one data rectangle intersected by Q is returned.

Other type of queries, such as arbitrarily shaped queries (e.g., point or polygonal queries) or retrieving all rectangles contained or containing Q, can be handled using a straightforward modification of the above procedure.

Updating the tree

Many applications require support for update operations, such as insertion and deletion of rectangles. A tree that can change over time via such operations is said to be *dynamic*.

New rectangles can be inserted using a procedure similar to that used to insert a new key in a B$^+$-tree. In other words, the new rectangle r is first added to a leaf node v and, if the node overflows, a split is performed that requires updating one rectangle and inserting another one in the parent of v. This procedure continues until either a node with fewer than B entries is found or the root is split, in which case a new node is created and the height of the tree grows by one. Independent of whether a split is performed or not, the bounding rectangles of all ancestors of r may need to be updated.

One important difference between R-trees and B$^+$-trees is that, in our case, there is no incorrect leaf node to which the new rectangle can be added. Choosing a leaf node impacts performance, not correctness, and performance depends on our ability to cluster rectangles so as to minimize the expected number of rectangles intersected by a query of a given size. In practice this means that clustering should attempt to minimize the areas and perimeters of the resulting MBRs. Models for predicting the expected number of nodes that need to be visited while performing an intersection query are discussed in Section 21.5.

The problem of partitioning a set of rectangles into buckets with capacity $B > 2$ such that the expected number of rectangles intersected by a random query is minimized is NP-hard [43]. Hence, it is unlikely we will ever know how to build optimal R-trees efficiently. As a result, many heuristic algorithms for building the tree have been proposed. It is worth noting that, in practice, some of the proposed heuristics result in well tuned R-trees and near optimal I/O for 2-dimensional data. We start with Guttman's original heuristics, which are of two types: leaf selection and node splitting.

To identify a leaf for insertion, Guttman proposes proceeding down the tree, always choosing the rectangle in the current node of the path whose area would increase by the smallest amount were we to insert the new rectangle in the subtree that corresponds to that rectangle. The reasoning behind this approach is that rectangles with small areas are less likely to be chosen for further exploration during a search procedure.

When a node overflows, a split is performed. Ideally, when this happens, one would like to partition a set S of $B + 1$ rectangles into two sets S_1 and S_2 such that the sum of their areas is minimized and each set contains at least b entries. Guttman proposes three different strategies, only the first of which is guaranteed to yield an optimal partition. The first strategy is a brute force algorithm that chooses the best split by checking all candidate partitions of the overflowed set S. This strategy is not practical, as the number of candidates is exponential in the node capacity, which can easily exceed 50 or so rectangles. The second strategy, quadratic split, starts by selecting the two rectangles $r_1, r_2 \in S$ which maximize the quantity area$(r') - $area$(r_1) - $area$(r_2)$, where r' is the MBR of $r_1 \cup r_2$. These two rectangles act as seeds which are assigned to different sides of the partition, i.e., one is assigned to S_1 and the other to S_2. The remaining entries of S are then assigned to the set (S_1 or S_2) whose MBR area increases the least when including the new rectangle in that set. The entries are not considered in arbitrary order. Rather, the next entry to be allocated is the one with the strongest preference for a group, i.e., the entry r that maximizes $|A_1 - A_2|$, where $A_i = \text{area}(\text{MBR}(S_i \cup \{r\})) - \text{area}(\text{MBR}(S_i))$. This heuristic runs in quadratic time and attempts to assign a priority to the unallocated entries according to the performance penalty that the wrong assignment could cause. If at any time during the allocation procedure the size of the smaller set plus the number of unallocated entries is equal to b, then all remaining entries are allocated to that set.

The third and final strategy, linear split, also assigns entries to the group whose MBR area increases the least, but differs from quadratic split in the method used to pick the seeds and in the order in which the remaining entries are allocated. The seeds are the two rectangles r_1 and r_2 whose separation is largest along (at least) one of the dimensions. We elaborate on this. Let $l_j(r)$ and $h_j(r)$ denote the low and high endpoints, respectively, of the j-th interval of r. The width of S along dimension j is simply $w_j = \max_r\{h_j(r)\} - \min_r\{l_j(r)\}$. The normalized separation of S along dimension j is $s_j = (\max_r\{l_j(r)\} - \min_r\{h_j(r)\})/w_j$. The seeds r_1 and r_2 are the two rectangles that yield the largest normalized separation considering all dimensions j. Once the seeds are chosen, the remaining entries are allocated to one set or the other in random order. This last heuristic runs in linear time.

A different linear split algorithm is described in [1]. Other efforts [2, 19] include polynomial time algorithms to partition a set of rectangles so as to minimize the sum of areas of the two resulting bounding rectangles.

In order to delete a rectangle r we first find the node v containing r and remove the entry, adjusting the bounding rectangles in all ancestors, as necessary. If the node occupancy goes below b the tree needs to be readjusted so as to keep the height in $O(\log_b n)$. There are different ways in which one can readjust the tree. One possibility is to redistribute the remaining entries of v among the siblings of v, in a manner similar to how underflowed nodes are treated in some B-tree algorithms. Instead, Guttman suggests reinserting the remaining entries of v, as this helps the global structure of the tree by considering non-sibling nodes during reinsertion. Of course, this procedure needs to be applied recursively, as internal nodes may underflow as well. Finally, if after deletion the root has exactly one child, the tree height shrinks by one and the only child becomes the new root.

21.3 Improving Performance

Since R-trees were first proposed in [23], many variants and methods to improve the structure and performance of the tree have been proposed. We discuss a few of the more common ones: R*-trees [3], Hilbert R-trees [29] and several bulk loading algorithms [28, 36, 51]. Other proposals for improving performance include [15, 20, 56, 57].

21.3.1 R* Tree

Given that the known R-tree insertion algorithms are based on heuristic optimization, it is reasonable to assess their merit experimentally. Beckmann et al [3] conducted an extensive experimental study to explore the impact of alternative approaches for leaf selection and node splitting. Based on their experiments, they proposed the R* tree which has become the most commonly implemented R-tree variant.

The R* tree differs from the original Guttman R-tree in three ways.

First, the leaf where a new object is inserted is chosen differently. The path selection algorithm makes use of the concept of *overlap* of entry E_i in node v_j, defined as $\text{overlap}(E_i) = \sum_{j=1, j \neq i}^{m} \text{area}(r_i \cap r_j)$, where m is the number of entries in node v_j and r_i is the rectangle associated with E_i. When descending from the root, if the next node to be selected is a leaf, the algorithm chooses the node that requires the least increase in overlap, and resolves ties as least area enlargement. If the next node is not a leaf, the entry with the least area enlargement is chosen.

The second difference is the use of *forced reinserts*. The authors discovered that the initial order of inserts significantly impacts tree quality. They also observed that query performance of an existing R-tree can be improved by removing half of the entries and then re-inserting them. Of course, the authors do not recommend performing a restructuring of this magnitude frequently. Rather, they used this insight to modify the policy for dealing with overflowed nodes. If an insertion causes an overflow, calculate the distance from the center of each of the $B + 1$ entries to the center of the MBR enclosing all $B + 1$ entries. Then sort the entries in decreasing order of this distance. Remove the p furthest entries, where p is set to 30% of B, and re-insert the p removed entries into the tree. Some subset of the p re-inserts may be inserted into nodes other than the initial node that overflowed. For each of the p re-inserts, if they do not cause an overflow, do nothing; otherwise, split the node using the algorithm below.

The third difference is in the node splitting algorithm. When a split is needed, the node entries are first sorted twice along each of the d dimensions. The two sorts are based on the low and on the high MBR endpoint values, respectively. Remember that nodes must have a minimum of b and a maximum of B entries. Thus, using one of the sorted lists, the $B+1$ entries can be partitioned into two groups, S_1 and S_2, by splitting anyplace after the i-th entry, $b \leq i \leq B-b+1$, of the sorted list. S_1 and S_2 consist of the entries before and after the split position, respectively. In order to choose the best split, the following three objective functions were considered (for 2-d data) and tested using different combinations:

1. area-value =area(MBR(S_1)) + area(MBR(S_2))
2. perimeter-value = perimeter(MBR(S_1)) + perimeter(MBR(S_2))
3. overlap-value = area(MBR(S_1) $\cap$ MBR(S_2))

Notice that for a fixed area, the MBR with smallest perimeter is the square.

Based on experiments, the following split policy is adopted. The R* tree computes the perimeter-values for each possible grouping (S_1, S_2) over both sorted lists of all dimensions and chooses the dimension that yields the minimum perimeter-value. Once the dimension has been chosen, the algorithm then chooses the grouping for that dimension that minimizes the overlap-value.

These three changes were shown to substantially improve the I/O performance for all data sets studied.

21.3.2 Hilbert Tree

The Hilbert R-tree [29] further improves performance by imposing a linear order on the input rectangles that results in MBRs of small area and perimeter. The tree is actually an R-tree augmented with order information. Intersection queries are performed as before, using the standard R-tree algorithm; but as a consequence of the ordering constraints, insertion and deletion can proceed as in B$^+$-trees and there is no longer a need to consider various leaf selection heuristics. Additionally, the linear order allows for effective use of deferred splitting, a technique which improves node utilization and performance as trees require fewer nodes for a given input set.

To define an ordering of the input values, Kamel and Faloutsos [29] propose the use of a space-filling curve, such as the Hilbert curve. The power of these curve lies in its ability to linearly order multidimensional points such that nearby points in this order are also close in multidimensional space. Hilbert curves are not the only reasonable choice. Other curves, such as the Peano or Z-order curve, may also be used. See [52] for a discussion on space-filling curves.

A d-dimensional Hilbert curve of order k is a curve H_k^d that visits every vertex of a finite d dimensional grid of size $2^k \times \ldots \times 2^k = 2^{kd}$. Its construction can best be viewed as a sequence of stages. At each stage, an instance of the curve of the previous stage is rotated and placed in each of 2^d equal-sized sub-quadrants. Endpoints of the 2^d sub-curves are then connected to produce the curve at the next stage. The first three stages of the Hilbert curve for two and three dimensions are illustrated in Figures 21.2 and 21.3, respectively.

Each grid vertex is assigned a Hilbert value, which is an integer that corresponds to its position along the curve. For instance, in H_2^2, $(0,0)$ and $(1,2)$ have Hilbert values 0 and 7, respectively. This assignment is easily extended to rectangles, in which case the Hilbert value of the grid point closest to the rectangle center is assigned. Algorithms for computing the position of a point along a space filling curve are given in [6, 10, 58].

The structure of the R-tree is modified as follows. Leaf nodes remain the same. Each entry

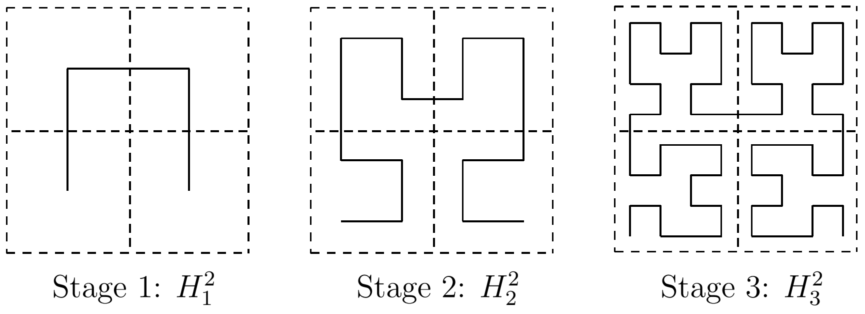


FIGURE 21.2: The first three stages of a 2-dimensional Hilbert curve.

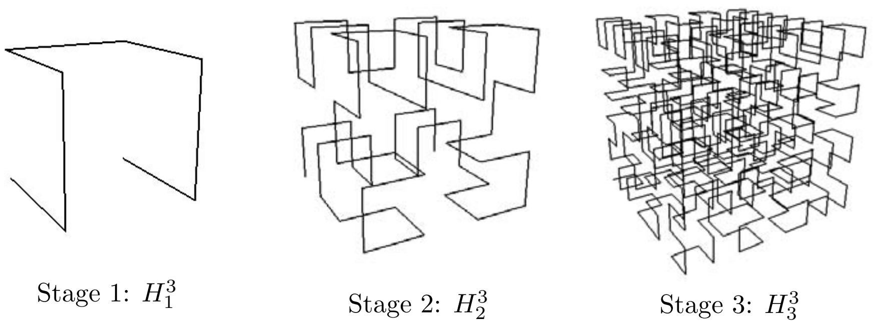


FIGURE 21.3: The first three stages of a 3-dimensional Hilbert curve.

of an internal node now has the form (r, p, v), where r and p have the same interpretation as before, and v is the largest Hilbert value of all data items stored in the subtree with root p. This assignment results in update algorithms that are similar to those used for B$^+$-trees. In particular, it is straightforward to implement an effective policy for *deferred splitting* which reduces the number of splits needed while performing insertions. The authors propose the following policy, which they call s-to-$(s+1)$ splitting. When a node overflows, an attempt is first made to shift entries laterally to $s-1$ sibling nodes at the same level. An actual split occurs only if the additional entry cannot be accommodated by shifting, because the $s-1$ siblings are already full. When this happens a new node is created and the $sB+1$ entries are distributed among the $s+1$ nodes. Because entries at the leaves are sorted by Hilbert value, bounding boxes of leaves tend to have small area and perimeter, and node utilization is high. Notice that there is a clear trade-off between node utilization and insertion complexity (which increases as s increases). The case $s = 1$ corresponds to the regular split policy used in the original R-tree, i.e., split whenever a node overflows.

A sample tree with 5 data rectangles is shown in Figure 21.4. There are two leaf nodes and one internal node (the root). The pointer entries of the internal node are represented by arrows. Each input rectangle has been annotated with its Hilbert value. In reality, the corners of the rectangles would fall on grid vertices. They have been made smaller in order to make the figure more readable. Inserting a new rectangle whose center has Hilbert value 17 would cause an overflow in r. With deferred splitting, a split is not necessary. Instead, the new rectangle would be accommodated in r after shifting rectangle c (with Hilbert value 28) to its sibling s.

The authors report improvements of up to 28% in performance over R*-trees and recommend using 2-to-3 splitting which results in an average node utilization of 82.2%.

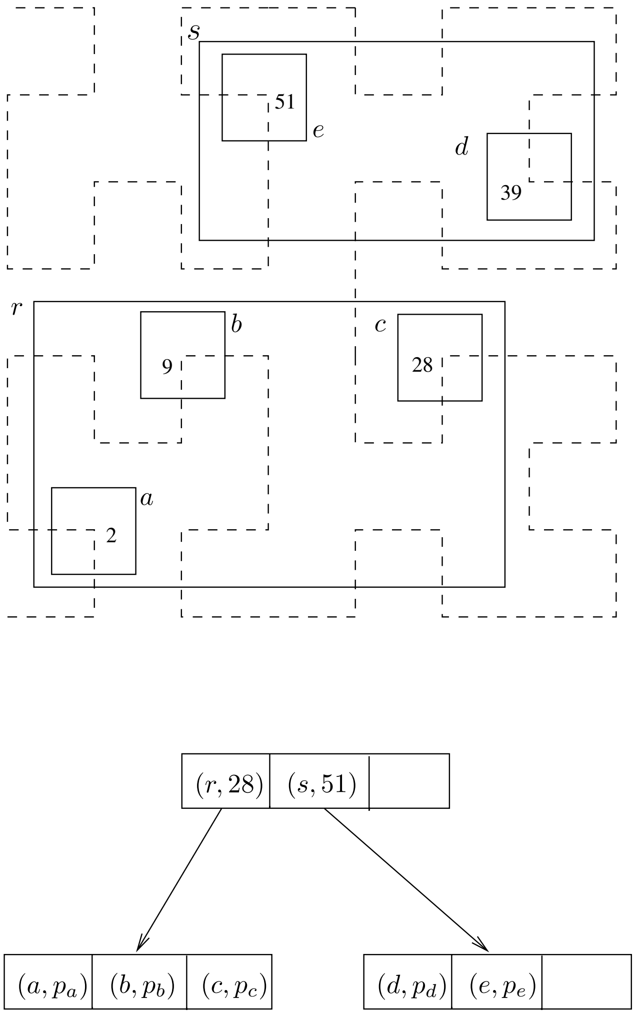


FIGURE 21.4: A Hilbert R-tree with $B = 3$.

In practice, one does not store or compute all bit values on the hypothetical grid. Let β be the number of bits required to describe one coordinate. Storing the Hilbert value of a d-dimensional point requires $d\beta$ bits of storage, which may be larger than the size of native machine integers. It is possible to compare the Hilbert values of two points without storing the values explicitly. Conceptually, the process computes bit positions, one at a time, until discrimination is possible. Consider the case of 2-d and notice that the first bit of the x- and y-coordinates of a point determine which quadrant contains it. Successive bits determine which successively smaller sub-quadrants contain the point. When two center points (x_1, y_1) and (x_2, y_2) need to be compared, the bits of each coordinate are examined until it can be determined that one of the points lies in a different sub-quadrant than the other (one can use the sense and rotation tables described in [28] to accomplish this task). The information gathered is used to decide which point is closer to the origin (along the Hilbert Curve).

21.3.3 Bulk Loading

There are applications where the data is static, or does not change very frequently. Even if the data is dynamic, it may happen that an index needs to be constructed for a large data set which is available a priori. In these circumstances, building an R-tree by inserting one object at a time has several disadvantages: (a) high load time, (b) sub-optimal space utilization, and, most important, (c) poor R-tree structure requiring the retrieval of a large number of nodes in order to satisfy a query. As discussed in the previous section, other dynamic algorithms [3, 57] improve the quality of the R-tree, but still are not competitive with regard to query time when compared to loading algorithms that are allowed to pre-process the data to be stored. When done properly, preprocessing results in R-trees with nearly 100% space utilization and improved query times (due to the fact that fewer nodes need to be accessed while performing a query). Such *packing algorithms* were first proposed by Roussopoulos [51] and later by Kamel and Faloutsos [28], and Leutenegger et al [36]. An approach that is intermediary between inserting a tuple at a time and constructing the entire tree by bulk loading is followed by [12], where an entire batch of input values is processed by partitioning the input into clusters and then inserting R-trees for the clusters into the existing R-tree.

The general approach to bulk loading an R-tree is similar to building a B-tree from a collection of keys by creating the leaf level first and then creating each successively higher level until the root node is created. The general approach is outlined below.

General Algorithm:

1. Sort the n rectangles and partition them into $\lceil n/B \rceil$ consecutive groups of B rectangles. Each group of B rectangles is eventually placed in the same leaf level node. Note that the last group may contain fewer than B rectangles.
2. Load the $\lceil n/B \rceil$ groups of rectangles into nodes and output the (MBR, address) for each leaf level node into a temporary file. The addresses are used as the child pointer fields for the nodes of the next higher level.
3. Recursively pack these MBRs into nodes at the next level, proceeding upwards, until the root node is created.

The three algorithms differ only in how the rectangles are sorted at each level. These differences are described below.

Nearest-X (NX):

This algorithm was proposed in [51]. The rectangles are sorted by the x-coordinate of a designated point such as the center. Once sorted, the rectangles are packed into nodes, in groups of size B, using this ordering. While our description is in terms of x, a different coordinate can clearly be used.

Hilbert Sort (HS):

The algorithm of [28] orders the rectangles using the Hilbert space filling curve. The center points of the rectangles are sorted based on their distance from the origin, measured along the curve. This process determines the order in which the rectangles are placed into the nodes of the R-Tree.

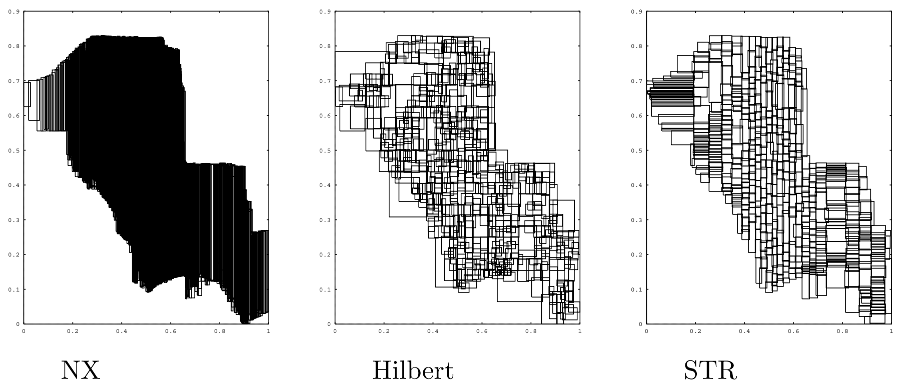


FIGURE 21.5: Leaf level nodes for three packing algorithms.

Sort-Tile-Recursive (STR):

STR [36] is best described recursively with $d = 2$ providing the base case. (The case $d = 1$ is already handled well by regular B-trees.) Accordingly, we first consider a set of rectangles in the plane. The basic idea is to "tile" the data space using $\sqrt{n/B}$ vertical slices so that each slice contains enough rectangles to pack roughly $\sqrt{n/B}$ nodes. Once again we assume coordinates are for the center points of the rectangles. Determine the number of leaf level pages $P = \lceil n/B \rceil$ and let $S = \lceil \sqrt{P} \rceil$. Sort the rectangles by x-coordinate and partition them into S vertical slices. A slice consists of a run of $S \cdot B$ consecutive rectangles from the sorted list. Note that the last slice may contain fewer than $S \cdot B$ rectangles. Now sort the rectangles of each slice by y-coordinate and pack them into nodes by grouping them into runs of length B (the first B rectangles into the first node, the next n into the second node, and so on).

The case $d > 2$ is is a simple generalization of the approach described above. First, sort the hyper-rectangles according to the first coordinate of their center. Then divide the input set into $S = \lceil P^{\frac{1}{d}} \rceil$ slabs, where a slab consists of a run of $B \cdot \lceil P^{\frac{d-1}{d}} \rceil$ consecutive hyper-rectangles from the sorted list. Each slab is now processed recursively using the remaining $d - 1$ coordinates (i.e., treated as a $(d - 1)$-dimensional data set).

Figure 21.5 illustrates the results from packing a set of segments from a Tiger file corresponding to the city of Long Beach. The figure shows the resultant leaf level MBRs for the same data set for each of the three algorithms using a value of $B = 100$ to bulk load the trees.

As reported in [36], both Hilbert and STR significantly outperform NX packing on all types of data except point data, where STR and NX perform similarly. For tests conducted with both synthetic and actual data sets, STR outperformed Hilbert on all but one set, by factors of up to 40%. In one instance (VLSI data), Hilbert packing performed up to 10% faster. As expected, these differences decrease rapidly as the query size increases.

21.4 Advanced Operations

Even though R-trees were originally designed to perform intersections queries, it did not take long before R-trees were being used for other types of operations, such as nearest neighbor, both simple [11, 25, 50] and constrained to a range [18], reverse nearest neighbor [32, 61, 68], regular and bichromatic closest pairs [14, 24, 59], incremental nearest neighbors

[25], topological queries [44], spatial joins [7, 8, 21, 22, 37–40, 47] and distance joins[24, 60]. Some of the proposals work on an ordinary R-tree while others require that the tree be modified or augmented in various ways. Because of space limitations we concentrate our attention on two problems: nearest neighbors and spatial joins.

21.4.1 Nearest Neighbor Queries

We discuss the problem of finding the input object closest to an arbitrary query point Q. We assume the standard Euclidean metric, i.e., if $a = (a_1, \ldots, a_d)$ and $b = (b_1, \ldots, b_d)$ are arbitrary points then $\mathrm{dist}(a, b) = (\sum_{i=1}^{d}(a_i - b_i)^2)^{1/2}$. For a general object O, such as a rectangle or polygon, we define $\mathrm{dist}(Q, O) = \min_{p \in O} \mathrm{dist}(Q, p)$.

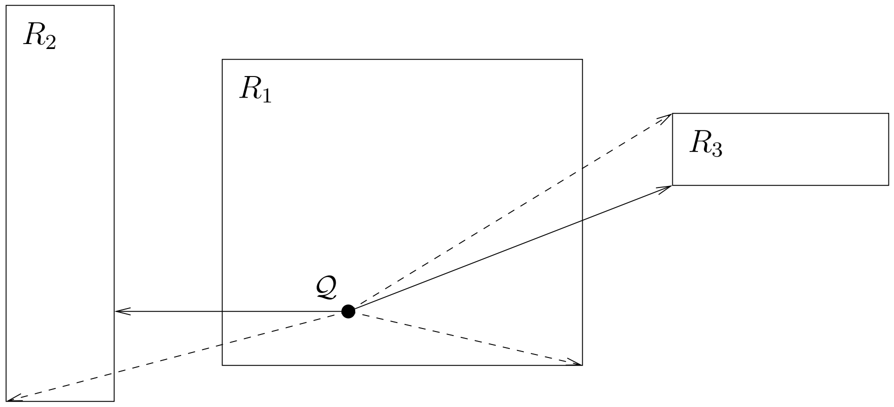


FIGURE 21.6: Illustration of mindist (solid) and minmaxdist (dashed) for three rectangles. Notice that $\mathrm{mindist}(Q, R_1) = 0$ because $Q \in R_1$.

The first proposal to solve this problem using R-trees is from [50]. Roussopoulos et al define two bounding functions of Q and an arbitrary rectangle r:

$\mathrm{mindist}(Q, r) = \mathrm{dist}(Q, r)$, the distance from Q to the closest point in r.

$\mathrm{minmaxdist}(Q, r) = \min_f \max_{p \in f}(\mathrm{dist}(Q, p)$, where f ranges over all $(d-1)$-dimensional facets of r.

Notice that $\mathrm{mindist}(Q, r) = 0$ if Q is inside r and that for any object or rectangle s that is a descendant of r, $\mathrm{mindist}(Q, r) \le \mathrm{dist}(Q, s) \le \mathrm{minmaxdist}(Q, r)$. This last fact follows from the fact that each of the facets of r must share a point with at least one input object, but this object can be as far as possible within an incident face. Thus, the bounding functions serve as optimistic and pessimistic estimates of the distance from Q to the nearest object inside r.

The following properties of the bounding functions readily follow:

P1 For any object or MBR r and MBR s, if $\mathrm{mindist}(Q, r) > \mathrm{minmaxdist}(Q, s)$ then r cannot be or contain the nearest neighbor of Q.

P2 For any MBR r and object s, if $\mathrm{mindist}(Q, r) > \mathrm{dist}(Q, s)$ then r cannot contain the nearest neighbor of Q.

The authors describe a branch-and-bound algorithm that performs a depth-first traversal of the tree and keeps track of the best distance so far. The two properties above are used to identify and prune branches that are guaranteed not to contain the answer. For each call the algorithm keeps a list of active nodes, i.e., nodes that tentatively need to be explored in search of a better estimate. No node of the list is explored until the subtrees corresponding to nodes appearing earlier in the active list have been processed or pruned. Thus, a sorting policy to determine the order in which rectangles stored in a node are processed is also required. In practice one would like to examine first those nodes that lower the best distance estimate as quickly as possible, but this order is difficult to determine a priori. Two criteria considered include sorting by mindist and by minmaxdist. Experimental results suggest that sorting by mindist results in slightly better performance. The algorithm is summarized in Figure 21.7. Global variable bestDistance stores the estimate of the distance to the nearest neighbor. The initial call uses the query and root of the tree as arguments.

```
findNearestNeighbor($Q$,$v$)
    if $v$ is a leaf then
        foreach rectangle $r$ in $v$ do
            if $dist(Q, r)$ < bestDistance then
                bestDistance ← $dist(Q, r)$
    else
        produce a list $L$ of all entries (of the form $(r, w)$) in $v$
        sort $L$ according to sorting criterion
        prune $L$ using property P1
        while $L$ is not empty do
            retrieve and remove next entry $(r, w)$ from $L$
            findNearestNeighbor($Q$,$w$)
            prune $L$ using property P2
end
```

FIGURE 21.7: Nearest neighbor algorithm of [50].

A simple modification to the above algorithm allows [50] to report the $k > 1$ nearest neighbors of Q. All is needed is to keep track of the best k distances encountered so far and to perform the pruning with respect to the k-th best.

Cheung and Fu [11] show that pruning based on P1 is not necessary and do away with computing minmaxdist altogether. They simplify the algorithm by pruning with P2 exclusively, and by rearranging the code so that the pruning step occurs before, and not after, each recursive call.

Hjaltason and Samet [24] also describe an algorithm that avoids using P1. Furthermore, unlike [50] which keeps a local list of active entries for each recursive call, their algorithm uses a global priority queue of active entries sorted by the optimistic distance from Q to that entry. This modification minimizes the number of R-tree nodes retrieved and results in an incremental algorithm, i.e., one that reports answers in increasing order of distance, a desirable characteristic when the value of k is not known a priori.

21.4.2 Spatial Joins

We consider the problem of calculating the intersection of two sets $R = \{r_1, r_2, \ldots, r_n\}$ and $S = \{s_1, s_2, \ldots, s_m\}$ of spatial objects. The spatial join, $R \bowtie S$, is the set of all pairs (r_i, s_j) such that $r_i \cap s_j \neq \emptyset$. The join of two spatial data sets is a useful and common operation. Consider, for example, the rivers and roads of a region, both represented using line segments. The spatial join of the river and road data sets yields the set of likely bridge locations. If the subset of river segments whose level is above a given threshold has been previously selected, the same spatial join now computes likely locations for road flooding.

Methods to compute spatial joins without R-trees exist but are not covered in this chapter. We consider the case where R-trees have been constructed for one or both data sets and hence can be used to facilitate the join computation. Unless otherwise stated, we will assume that the r_i's and s_j's refer to the MBRs enclosing the actual data objects.

In [8], Brinkhoff et al proposed the "canonical" spatial join algorithm based on R-trees. A similar join algorithm was proposed at the same time by Gunther [21]. Gunther's algorithm is applicable for general trees and includes R-trees as a special case. In Figure 21.8 we paraphrase the algorithm of [8]. Let v_1 and v_2 be nodes of R-trees T_1 and T_2, respectively. Let E_{ij}, be an entry of v_i of the form (r_{ij}, p_{ij}), where r_{ij} and p_{ij} denote the MBR and child pointer, respectively. To join data sets R and S, indexed by R-trees T_1 and T_2, invoke the SpatialJoin algorithm in Figure 21.8, passing as arguments the roots of T_1 and T_2.

```
 SpatialJoin($v_1$,$v_2$)
     **foreach** entry $E_{1j} \in v_1$ **do**
         **foreach** entry $E_{2j} \in v_2$ such that $R_{1j} \cap R_{2j} \neq \emptyset$ **do**
             **if** $v_1$ and $v_2$ are leaf nodes **then**
                 output entries $E_{1j}$ and $E_{2j}$
             **else**
                 Read the nodes pointed to by $p_{1j}$ and $p_{2j}$
                 SpatialJoin($p_{1j}$, $p_{2j}$)
**end**
```

FIGURE 21.8: Spatial join algorithm of [8] for two R-trees.

In [8] the authors generalize the algorithm for the case where the two R-trees have different heights. They also improve upon the basic algorithm by reducing CPU computation and disk accesses. The CPU time is improved by reducing the number of rectangle intersection checks. One method to accomplish this is to first calculate the subset of entries within nodes v_1 and v_2 that intersect MBR(v_1) $\cap$ MBR(v_2), and then do full intersection tests among this subset. A second approach is to sort the entries and then use a sweep-line algorithm. In addition, the paper considers reduction of page I/O by ordering the accesses to pages so as to improve the buffer hit ratio. This is done by accessing the data in sweep-line order or Z-order.

In [7] Brinkhoff et al suggest the following 3-step approach for joining complex polygonal data sets: (1) use the R-trees and MBRs of the data sets to reduce the set to a list of potential hits; (2) use a lightweight approximation algorithm to further reduce the set, leaving in some "false positives"; (3) conduct the exact intersection tests on the remaining polygons using techniques from computational geometry (see Chapter 2 of [16], for example).

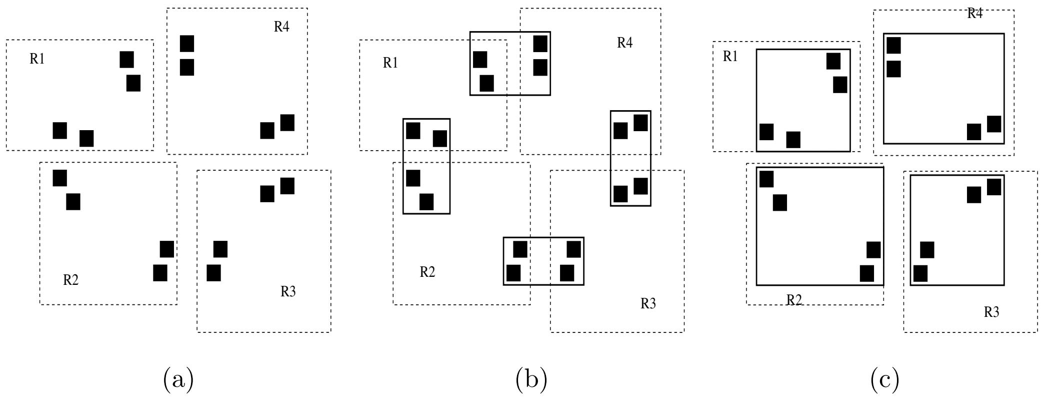


FIGURE 21.9: Seeded Tree Example. (a) Existing R-tree and non-indexed data (dark squares); (b) Normal R-tree structure for non-indexed data; (c) Seeded structure.

Often only one of the data sets to be joined will be indexed by an R-tree. This is the case when the DBA decides that maintaining an index is too expensive, when the data set being joined is the result of a series of non-spatial attribute selections, or for multi-step joins where intermediate results are not indexed.

To join two data sets when only one is indexed by an R-tree, Lo and Ravishankar [37] propose the idea of *seeded tree join*. The main idea is to use the existing R-tree to "seed" the creation of a new index, called the seeded tree, for the non-indexed data set, and then perform a join using the method of [8].

Consider the example shown in Figure 21.9 (similar to the example in [37]). Assume that the existing tree has four entries in the root node. The four rectangles, R1 to R4, in the left hand side represent the bounding rectangles of these entries. The dark filled squares do not belong to the existing R-tree but, rather, correspond to some of the data items in the non-indexed data set. Assuming a node capacity of four (i.e., $B = 4$) and using the normal insertion or loading algorithms which minimize area and perimeter, the dark rectangles would likely be grouped as shown by the dark MBRs in Figure 21.9b. A spatial join would then require that each node of each tree be joined with two nodes from the other tree, for a total of eight pairs. On the other hand, if the second R-tree was structured as show in Figure 21.9c, then each node would be joined with only one node from the other tree, for a total of four pairs. Hence, when performing a spatial join, it might be better to structure the top levels of the tree in a fashion that is sub-optimal for general window queries.

The general algorithm for seeded tree creation is to copy the top few levels of the existing tree and use them as the top levels for the new tree. These top levels are called the seed levels. The entries at the lowest seed level are called "slots". Non-indexed data items are then inserted into the seeded tree by inserting them into an R-tree that is built under the appropriate slot. In [37] the authors experimentally compare three join algorithms: (1) R-tree join, where an R-tree is fist built on the non-indexed data set and then joined; (2) brute force, where the non-indexed data set is read sequentially and a region query is run against the existing R-tree for each data item in the non-indexed data set; (3) seeded tree join, where a seeded tree is built and then joined. The authors consider several variants of the seeded tree creation algorithm and compare the performance. Experimental studies show that the seeded tree method significantly reduces I/O. Note that if the entire seeded tree does not fit in memory significant I/O can occur during the building process. The authors propose to minimize this I/O by buffering runs for the slots and then building the tree for each slot during a second pass.

Another approach [22] for the one-index case is to sort the non-indexed data using the MBR lower endpoints for one of the dimensions, sort the leaf level MBRs from the existing R-tree on the same dimension, and finally join the two sorted data sets using a sweep-line algorithm. The authors analytically demonstrate that as long as the buffer size is sufficient to merge-sort efficiently, their algorithm results in less I/O than creating any type of index followed by a join. Experimental results also show a significant I/O reduction.

In [47] a relational hash-based join approach is used. Although mostly hash-based, the method does need to default to R-trees in some cases. A sampled subset of R is partitioned into N buckets, $R_1 \dots R_N$, for some N. Each non-sampled object from R is then added to the bucket that requires the least enlargement. Set S is then partitioned in N corresponding buckets, $S_1 \dots S_N$ by testing each object o_S of S and placing a copy of it into S_i, for each bucket R_i such that $o_S \cap R_i \neq \emptyset$. If an object in S does not intersect any of the R_i buckets the object is discarded. The bucket pairs (R_i, S_i) are then read into memory and pairwise joined. If a bucket pair is too large to fit in memory, an R-tree index is built for one of the two buckets and an index-loop join is used.

In [38, 40] a method that combines the seeded tree and the hash-join approach is proposed. The proposed method, *slot index spatial join*, chooses a level from the existing tree and uses the MBRs of the entries as the buckets for the hashing. The chosen level is determined by the number of entries at that level and the number of buffer pages available for the bucket joining algorithm. Since R-trees have a wide fan out, the optimal number of buckets to use usually falls between level sizes. The paper considers several heuristics for combining MBRs from the next level down to tune the number of buckets. Overall performance is shown to be better than all previously existing methods in all but a few cases.

A method to join multiple spatial relations together is discussed in [38, 39]. The authors propose a multi-way join algorithm called *synchronous traversal* and develop cost models to use for query optimization. They consider using only pairwise joins, only the synchronous traversal method, and combinations of the two approaches. They show that, in general, a combination of the two results in the best performance.

The concept of spatial join has been further generalized to *distance joins*, where a distance based ordering of a subset of the Cartesian product of the two sets is returned. In [24, 60], distance join algorithms using R-trees are proposed.

21.5 Analytical Models

Analytical models of R-trees provide performance estimates based on the assumptions of the model. Such models can be useful for gaining insight, comparing algorithms, and for query optimization.

Early analytical models for range queries were proposed by Kamel and Faloutsos [28], and Pagel et al [42]. The models are similar and use the MBRs of all nodes in the tree as inputs to the model. They derive the probability that Q intersects a given MBR, and use this estimate to compute the expected number of MBRs intersected by Q. In [4, 34] the basic model was modified to correct for an error that may arise for MBRs near the boundary of the query sample space. In order to do this, [34] assumes that all queries fall completely within the data space. The changes necessary to handle different sample spaces are straightforward.

The models provide good insight into the problem, especially by establishing a quantitative relationship between performance and the total area and perimeter of MBRs of the tree nodes. We describe the model as presented in [34].

Consider a 2-dimensional data set consisting of rectangles to be stored in an R-tree T with $h+1$ levels, labeled 0 through h. Assume all input rectangles have been normalized to fit within the unit square $U = [0,1] \times [0,1]$. Queries are rectangles Q of size $q_x \times q_y$. (A point query corresponds to the case $q_x = q_y = 0$.) Initially assume that queries are uniformly distributed over the unit square. Although this description concentrates on 2-d, generalizations to higher dimensions are straightforward.

Assume the following notation:

m_i	$=$	number of nodes at the ith level of T
m	$=$	Total number nodes in T, i.e., $\sum_{i=0}^{h} m_i$
R_{ij}	$=$	jth rectangle at the ith level of T
X_{ij}	$=$	x-extent (width) of R_{ij}
Y_{ij}	$=$	y-extent (height) of R_{ij}
A_{ij}	$=$	area of R_{ij}, i.e., $A_{ij} = X_{ij} \cdot Y_{ij}$
A_{ij}^{Q}	$=$	probability that R_{ij} is accessed by query Q
B_{ij}	$=$	number of accesses to R_{ij}
$\mathcal{A}$	$=$	Sum of the areas of all MBRs in T
L_x	$=$	Sum of the x-extents of all MBRs in T
L_y	$=$	Sum of the y-extents of all MBRs in T
N	$=$	number of queries performed so far
N^*	$=$	expected number of queries required to fill the buffer
β	$=$	buffer size
$D(N)$	$=$	number of distinct nodes (at all levels) accessed in N queries
$E_T^P(q_x, q_y)$	$=$	expected number of nodes (buffer resident or not) of T accessed while performing a query of size $q_x \times q_y$
$E_T^D(q_x, q_y)$	$=$	expected number of disk accesses while performing a query of size $q_x \times q_y$

The authors of [28, 42] assume that performance is measured by the number of nodes accessed (independent of buffering). They observe that for uniform point queries the probability of accessing R_{ij} is just the area of R_{ij}, namely, A_{ij}. They point out that the level of T in which R_{ij} resides is immaterial as all rectangles containing Q (and only those) need to be retrieved. Accordingly, for a point query, the expected number of nodes retrieved as derived in [28] is the sum of node areas*:

*We have modified the notation of [28] to make it consistent with the notation used here.

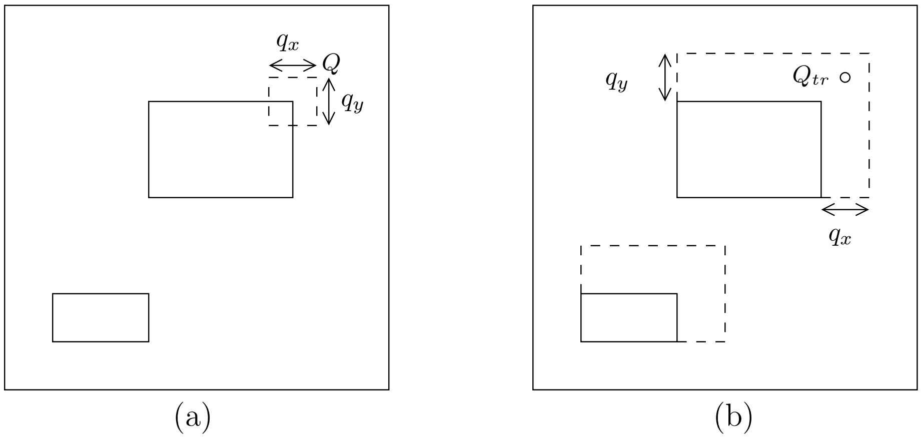


FIGURE 21.10: (a) Two data rectangles and region query Q (b) Corresponding extended rectangles and equivalent point query Q_{tr}.

$$E_T^P(0,0) = \sum_{i=0}^{h}\sum_{j=1}^{m_i} A_{ij} = \mathcal{A} \tag{21.1}$$

which is the sum of the areas of all rectangles (both leaf level MBRs as well as MBRs of internal nodes).

We now turn our attention to region queries. Let $\langle (a,b),(c,d)\rangle$ denote an axis-parallel rectangle with bottom left and top right corners (a,b) and (c,d), respectively. Consider a rectangular query $Q = \langle Q_{bl}, Q_{tr}\rangle$ of size $q_x \times q_y$. Q intersects $R = \langle (a,b),(c,d)\rangle$ if and only if Q_{tr} (the top right corner of Q) is inside the *extended rectangle* $R' = \langle (a,b),(c+q_x,d+q_y)\rangle$, as illustrated in Figure 21.10.

Kamel and Faloutsos infer that the probability of accessing R while performing Q is the area of R', as the region query Q is equivalent to a point query Q_{tr} where *all* rectangles in T have been extended as outlined above. Thus, the expected number of nodes retrieved (as derived in [28]) is:

$$\begin{aligned} E_T^P(q_x,q_y) &= \sum_{i=0}^{h}\sum_{j=1}^{m_i}(X_{ij}+q_x)(Y_{ij}+q_y) \\ &= \sum_{i=0}^{h}\sum_{j=1}^{m_i}(X_{ij}Y_{ij} + q_x\sum_{i=0}^{h}\sum_{j=1}^{m_i}Y_{ij} + q_y\sum_{i=0}^{h}\sum_{j=1}^{m_i}X_{ij} + mq_xq_y \\ &= \mathcal{A} + q_xL_y + q_yL_x + mq_xq_y \end{aligned} \tag{21.2}$$

Equation 21.2 illustrates the fact that a good insertion/loading algorithm should cluster rectangles so as to minimize both the total area and total perimeter of the MBRs of all nodes. For point queries, on the other hand, $q_x = q_y = 0$, and minimizing the total area is enough.

In [34] the model of [28] was modified to handle query windows that fall partially outside the data space as well as data rectangles close to the boundary of the data space, as suggested by Six et al [42]. Specifically:

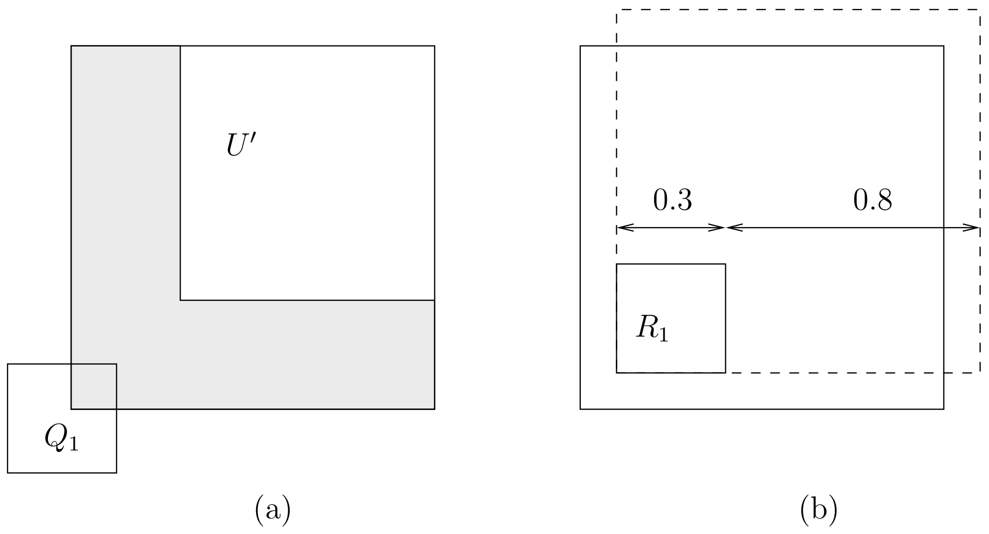


FIGURE 21.11: The domain of Q_{tr} for a query of size 0.3×0.3 is U' (area not shaded).

1. For uniformly distributed rectangular queries of size $q_x \times q_y$ the top right corner of the query region Q, cannot be an arbitrary point inside the unit square if the entire query region is to fit within the unit square. For example, if $q_x = q_y = 0.3$, a query such as Q_1 in Figure 21.11a should not be allowed. Rather, Q_{tr} must be inside the box $U' = [q_x, 1] \times [q_y, 1]$.
2. The probability of accessing a rectangle $R = \langle (a, b), (c, d) \rangle$ is *not always* the area of $R' = \langle (a, b), (c + q_x, d + q_y) \rangle$ as this value can be bigger than one. For example, in Figure 21.11b, the probability that a query of size 0.8×0.8 accesses rectangle R_1 should not be $1.1 \cdot 1.1 = 1.21$, which is the area of the extended rectangle R'_1, obtained by applying the original formula. Rather, the access probability is the percentage of U' covered by the rectangle $R' \cap U'$.

Thus, we change the probability of accessing rectangle i of level j to:

$$\begin{aligned} A^Q_{i,j} &= \frac{\text{area}(R' \cap U')}{\text{area}(U')} \\ &= \frac{C \cdot D}{(1 - q_x)(1 - q_y)} \end{aligned} \tag{21.3}$$

where $C = [\min(1, c + q_x) - \max(a, q_x)]$ and $D = [\min(1, d + q_y) - \max(b, q_y)]$.

In [35] the R-tree model was expanded to take into account the distribution of the input data. Specifically, rather than being uniformly distributed, the query regions were assumed to be distributed according to the input data distribution.

The above models do not consider the impact of the buffer. In [35] a buffer model is integrated with the query model. Specifically, under uniformly distributed point queries, the probability of accessing rectangle R_{ij} while performing a query is A_{ij}. Accordingly, the probability that R_{ij} is not accessed during the next N queries is $P[B_{ij} = 0|N] = (1 - A_{ij})^N$. Thus, $P[B_{ij} \geq 1|N] = 1 - (1 - A_{ij})^N$ and the expected number of *distinct* nodes accessed in N queries is,

$$D(N) = \sum_{i=0}^{h} \sum_{j=1}^{m_i} P[B_{ij} \geq 1|N] = m - \sum_{i=0}^{h} \sum_{j=1}^{m_i} (1 - A_{ij})^N \tag{21.4}$$

Note that $D(0) = 0 < \beta$ and $D(1) = \mathcal{A}$ (which may or may not be bigger than β). The buffer, which is initially empty, first becomes full after performing N^* queries, where N^* is the smallest integer that satisfies $D(N^*) \geq \beta$. The value of N^* can be determined by a simple binary search.

While the buffer is not full the probability that R_{ij} is in the buffer is equal to $P[B_{ij} \geq 1]$. The probability that a random query requires a disk access for R_{ij} is $A_{ij} \cdot P[B_{ij} = 0]$. Since the steady state buffer hit probability is approximately the same as the buffer hit probability after N^* queries, the expected number of disk accesses for a point query at steady state is

$$\sum_{i=0}^{h} \sum_{j=1}^{m_i} A_{ij} \cdot P[B_{ij} = 0|N^*] = \sum_{i=0}^{h} \sum_{j=1}^{m_i} A_{ij} \cdot (1 - A_{ij})^{N^*} \tag{21.5}$$

The above derivation also holds for region queries provided that A_{ij}^Q is used instead of A_{ij}, i.e.:

$$E_T^D(q_x, q_y) = \sum_{i=0}^{h} \sum_{j=1}^{m_i} A_{ij}^Q \cdot (1 - A_{ij}^Q)^{N^*}$$

In [34] the authors compare the model to simulation and explore the I/O impact of pinning the top levels of an R-tree into the buffer.

Other analytical models include the following. Theodoridis and Sellis [65] provide a fully analytical model that does not require the R-tree MBRs as input. In [17], a technique is developed for analyzing R-tree performance with skewed data distributions. The technique uses the concept of fractal dimension to characterize the data set and resultant R-tree performance. Analysis has also been used for estimating the performance of R-tree based spatial joins [26, 66] and nearest neighbor queries [45, 64].

Acknowledgment

This work was supported in part by the National Science Foundation under Grants IRI-9610240 and DMS-0107628.

References

[1] C.H. Ang and T.C. Tan. New linear node splitting algorithm for R-trees. In *Advances in Spatial Databases, 5th International Symposium (SSD)*, volume 1262 of *Lecture Notes in Computer Science*, pages 339–349. Springer, 1997.

[2] B. Becker, P. G. Franciosa, S. Gschwind, S. Leonardi, T. Ohler, and P. Widmayer. Enclosing a set of objects by two minimum area rectangles. *Journal of Algorithms*, 21(3):520–541, 1996.

[3] N. Beckmann, H.P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: an efficient and robust access method for points and rectangles. In *Proc. ACM SIGMOD*, pages 323–331, may 1990.

[4] S. Berchtold, C. Bohm, and H.-P. Kriegel. Improving the query performance of high-dimensional index structures by bulk-load operations. In *Proc. 6th International Conference on Extending Database Technology*, pages 216–230, 1998.

[5] S. Berchtold, D. Keim, and H.-P. Kriegel. The X-tree: An index structure for high-dimensional data. In *Proc. 22nd International Conference on Very Large Databases (VLDB)*, pages 28–39, 1996.

[6] T. Bially. Space-filling curves: their generation and their application to bandwidth reduction. *IEEE Transactions on Information Theory*, IT-15(6):658–664, 1969.

[7] T. Brinkhoff, H.-P. Kriegel, R. Schneider, and B. Seeger. Multi-step processing of spatial joins. In *Proc. ACM SIGMOD*, pages 197–208, 1994.

[8] T. Brinkhoff, H.-P. Kriegel, and B. Seeger. Efficient processing of spatial joins using R-trees. In *Proc. ACM SIGMOD*, pages 237–246, 1993.

[9] T. Brinkhoff, H.-P. Kriegel, and B. Seeger. Parallel processing of spatial joins using R-trees. In *Proc. 12th International Conference on Data Engineering (ICDE)*, pages 258–265, 1996.

[10] A.R. Butz. Alternative algorithm for Hilbert's space-filling curve. *IEEE Transactions on Computers*, C-20:424–426, 1971.

[11] K. L. Cheung and A. W.-C. Fu. Enhanced nearest neighbour search on the R-tree. *SIGMOD Record*, 27(3):16–21, 1998.

[12] R. Choubey, L. Chen, and E.A. Rundensteiner. GBI: A generalized R-tree bulk-insertion strategy. In *Symposium on Large Spatial Databases*, pages 91–108, 1999.

[13] D. Comer. The ubiquitous B-tree. *ACM Computing Surveys*, 11(2):121–137, 1979.

[14] A. Corral, Y. Manolopoulos, Y. Theodoridis, and M. Vassilakopoulos. Closest pair queries in spatial databases. In *Proc. ACM SIGMOD*, pages 189–200, 2000.

[15] M. de Berg, J. Gudmundsson, M. Hammar, and M. Overmars. On R-trees with low query complexity. *Computational geometry: Theory of Applications*, 24(3):179–195, 2003.

[16] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, 2nd edition, 2000.

[17] C. Faloutsos and I. Kamel. Beyond uniformity and independence: Analysis of R-trees using the concept of fractal dimension. In *Proc. ACM Symposium on Principles of Database Systems (PODS)*, pages 4–13, 1994.

[18] H. Ferhatosmanoglu, I. Stanoi, D. Agrawal, and A. El Abbadi. Constrained nearest neighbor queries. *Lecture Notes in Computer Science*, 2121:257–278, 2001.

[19] Y. Garcia, M.A. Lopez, and S.T. Leutenegger. On optimal node splitting for R-trees. In *Proc. International Conference on Very Large Databases (VLDB)*, pages 334–344, August 1998.

[20] Y. Garcia, M.A. Lopez, and S.T. Leutenegger. Post-optimization and incremental refinement of R-trees. In *Proc. 7th International Symposium on Advances in Geographic Information Systems (ACM GIS)*, pages 91–96. ACM, 1999.

[21] O. Gunther. Efficient computation of spatial joins. In *Proc. International Conference on Data Engineering (ICDE)*, pages 50–59, 1993.

[22] C. Gurret and P. Rigaux. The sort/sweep algorithm: A new method for R-tree based spatial joins. In *Proc. SSDBM*, pages 153–165, 2002.

[23] A. Guttman. R-trees: a dynamic index structure for spatial searching. In *Proc. ACM SIGMOD*, pages 47–57, June 1984.

[24] G.R. Hjaltason and H. Samet. Incremental distance join algorithms for spatial databases. In *Proc. ACM SIGMOD*, pages 237–248, 1998.

[25] G.R. Hjaltason and H. Samet. Distance browsing in spatial databases. *ACM Transactions on Database Systems*, 24(2):265–318, 1999.

[26] Y.-W. Huang, N. Jing, and E. A. Rundensteiner. A cost model for estimating the performance of spatial joins using R-trees. In *Statistical and Scientific Database Management*, pages 30–38, 1997.

[27] I. Kamel and C. Faloutsos. Parallel R-trees. In *Proc. ACM SIGMOD*, pages 195–204, 1992.

[28] I. Kamel and C. Faloutsos. On packing R-trees. In *Proc. 2nd International Conference on Information and Knowledge Management (CIKM)*, November 1993.

[29] I. Kamel and C. Faloutsos. Hilbert R-tree: an improved R-tree using fractals. In *Proc. International Conference on Very Large Databases (VLDB)*, pages 500–509, September 1994.

[30] N. Katayama and S. Satoh. The SR-tree: An index structure for high-dimensional nearest neighbor queries. In *Proc. ACM SIGMOD*, pages 369–380, 1997.

[31] G. Kollios, V.J. Tsotras, D. Gunopulos, A. Delis, and M. Hadjieleftheriou. Indexing animated objects using spatiotemporal access methods. *IEEE Transactions on Knowledge and Data Engineering*, 13(5):758–777, 2001.

[32] F. Korn and S. Muthukrishnan. Influence sets based on reverse nearest neighbor queries. In *Proc. ACM SIGMOD*, pages 201–212, 2000.

[33] M. Kornacker and D. Banks. High-concurrency locking in R-trees. In *Proc. International Conference on Very Large DataBases (VLDB)*, pages 134–145, 1995.

[34] S.T. Leutenegger and M.A. Lopez. The effect of buffering on the performance of R-trees. In *Proc. 15th International Conference on Data Engineering (ICDE)*, pages 164–171, 1998.

[35] S.T. Leutenegger and M.A. Lopez. The effect of buffering on the performance of R-trees. *IEEE Transactions on Knowledge and Data Engineering*, 12(1):33–44, 2000.

[36] S.T. Leutenegger, M.A. Lopez, and J.M. Edgington. STR: A simple and efficient algorithm for R-tree packing. In *Proc. 14th International Conference on Data Engineering (ICDE)*, pages 497–506, 1997.

[37] M-L. Lo and C.V. Ravishankar. Spatial joins using seeded trees. In *Proc. ACM SIGMOD*, pages 209–220, 1994.

[38] N. Mamoulis and D. Papadias. Integration of spatial join algorithms for processing multiple inputs. In *Proc. ACM SIGMOD*, 1999.

[39] N. Mamoulis and D. Papadias. Multiway spatial joins. *ACM Transactions on Database systems*, 26(4):424–475, 2001.

[40] N. Mamoulis and D. Papadias. Slot index spatial join. *IEEE Transactions on Knowledge and Data Engineering*, 15(1):1–21, 2003.

[41] V. Ng and T. Kameda. Concurrent access to R-trees. In *Proc. SSD*, pages 142–161, 1993.

[42] B-U. Pagel, H-W. Six, H. Toben, and P. Widmayer. Towards an analysis of range query performance in spatial data structures. In *Proc. ACM Symposium on Principles of Database Systems (PODS)*, pages 214–221, May 1993.

[43] B.-U. Pagel, H.-W. Six, and M. Winter. Window query-optimal clustering of spatial objects. In *Proc. 14th ACM Symposium on Principles of Database Systems (PODS)*, pages 86–94, 1995.

[44] D. Papadias, T. Sellis, Y. Theodoridis, and M. Egenhofer. Topological relations in the world of minimum bounding rectangles: a study with R-trees. In *Proc. ACM SIGMOD*, pages 92–103, 1995.

[45] A. Papadopoulos and Y. Manolopoulos. Performance of nearest neighbor queries in R-trees. In *Proc. 6th International Conference on Database Theory*, pages 394–408, 1997.

[46] A. Papadopoulos and Y. Manolopoulos. Similarity query processing using disk arrays. In

Proc. ACM SIGMOD, pages 225–236, 1998. parallel NN using R-trees.

[47] J.M. Patel and D.J. DeWitt. Partition based spatial-merge join. In *Proc. ACM SIGMOD*, 1996.

[48] D. Pfoser, C.S. Jensen, and Y. Theodoridis. Novel approaches in query processing for moving object trajectories. In *Proc. 26th International Conference on Very Large Databases (VLDB)*, pages 395–406, 2000.

[49] C.M. Procopiuc, P.K. Agarwal, and S. Har-Peled. Star-tree: An efficient self-adjusting index for moving objects. In *Proc. 4th Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 178–193, 2002.

[50] N. Roussopoulos, S. Kelley, and F. Vincent. Nearest neighbor queries. In *Proc. ACM SIGMOD*, pages 71–79, May 1995.

[51] N. Roussopoulos and D. Leifker. Direct spatial search on pictorial databases using packed R-trees. In *Proc. ACM SIGMOD*, pages 17–31, Austin, Texas, May 1985.

[52] H. Sagan. *Space-filling curves*. Springer-Verlag, 1994.

[53] S. Saltenis and C. Jensen. Indexing of now-relative spatio-bitemporal data. *The VLDB Journal*, 11(1):1–16, 2002.

[54] S. Saltenis, C. Jensen, S. Leutenegger, and M.A. Lopez. Indexing the positions of continuously moving points. In *Proc. ACM SIGMOD*, pages 331–342, May 2000.

[55] B. Schnitzer and S.T. Leutenegger. Master-client R-trees: A new parallel R-tree architecture. In *Proc. Conference of Scientific and Statistical Database Systems (SSDBM)*, pages 68–77, 1999.

[56] T. Schreck and Z. Chen. Branch grafting method for R-tree implementation. *Journal of Systems and Software*, 53:83–93, 2000.

[57] T. Sellis, N. Roussopoulos, and C. Faloutsos. The R^+ tree: A dynamic index for multi-dimensional objects. In *Proc. 13th International Conference on Very Large Databases (VLDB)*, pages 507–518, September 1987.

[58] K. Sevcik and N. Koudas. Filter trees for managing spatial data over a range of size granularities. Technical Report CSRI-TR-333, Computer Systems Research Institute, University of Toronto, October 1995.

[59] J. Shan, D. Zhang, and B. Salzberg. On spatia-range closest-pair queries. In *Proc. 8th International Symposium on Spatial and Temporal Databases (SSTD)*, pages 252–269, 2003.

[60] H. Shin, B. Moon, and S. Lee. Adaptive multi-stage distance join processing. In *Proc. ACM SIGMOD*, pages 343–354, 2000.

[61] I. Stanoi, D. Agrawal, and A. El Abbadi. Reverse nearest neighbor queries for dynamic databases. In *ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery*, pages 44–53, 2000.

[62] Y. Tao and D. Papadias. MV3R-tree: A spatio-temporal access method for timestamp and interval queries. In *Proc. 27th International Conference on Very Large Databases (VLDB)*, pages 431–440, 2001.

[63] Y. Tao, D. Papadias, and J. Sun. The TPR*-tree: An optimized spatio-temporal access method for predictive queries. In *Proc. International Conference on Very Large Databases (VLDB)*, pages 790–801, 2003.

[64] Y. Tao, J. Zhang, D. Papadias, and N. Mamoulis. An efficient cost model for optimization of nearest neighbor search in low and medium dimensional spaces. *IEEE Transactions on Knowledge and Data Engineering*, to appear.

[65] Y. Theodoridis and T. Sellis. A model for the prediction of R-tree performance. In *Proc. 8th ACM Symposium on Principles of Database Systems (PODS)*, May 1996.

[66] Y. Theodoridis, E. Stefanakis, and T. K. Sellis. Efficient cost models for spatial queries using R-trees. *IEEE Transactions Knowledge and Data Engineering*, 12(1):19–32, 2000.

[67] D. White and R. Jain. Similarity indexing: algorithms and performance. In *Proc. SPIE Storage and Retrieval for Still Image and Video Databases IV*, volume 2670, pages 62–73, 1996.

[68] C. Yang and K.-I. Lin. An index structure for efficient reverse nearest neighbor queries. In *Proc. International Conference on Data Engineering (ICDE)*, pages 485–492, 2001.

22
Managing Spatio-Temporal Data

Sumeet Dua
Louisiana Tech University

S. S. Iyengar
Louisiana State University

22.1 Introduction and Background

Space and time are indispensable for many objects in the real world. Spatial databases represent, store and manipulate spatial data such as points, lines, areas, surfaces and hyper-volumes in multidimensional space. Most of these databases suffer from, what is commonly called, the "Curse of Dimensionality" [1]. Curse of dimensionality refers to a performance degradation of similarity queries with increasing dimensionality of these databases. One way to reduce this curse is to develop data structures for indexing such databases to answer similarity queries efficiently. Specialized data structures such as R-trees and its variants (Chapter 21), have been proposed for this purpose which have demonstrated multi-fold performance gains in access time on this data over sequential search.

Temporal databases, on the other hand, store time-variant data. Traditional spatial data structures can store only one 'copy' of the data, latest at the 'present time', while we frequently encounter applications where we need to store more than one copy of the data. While spatial data structures typically handle objects having spatial components and temporal data structures handle time-variant objects, research in Spatio-temporal data structures and data models have concentrated around storing moving points and moving objects, or objects with geometries changing over time.

Some examples of domains generating spatio-temporal data include but are not limited to the following: geographical information systems, urban planning systems, communication systems, multimedia systems and traffic planning systems. While the assemblage of spatio-temporal data is growing, the development of efficient data structures for storage and retrieval of these data types hasn't kept pace with this increase. Research in spatio-temporal data abstraction is strewn but there are some important research results in this area that have laid elemental foundation for development of novel structures for these data types. In this chapter we will present an assortment of key modeling strategies for Spatio-temporal data types.

In Spatio-temporal databases (STB), two concepts of times are usually considered: *transaction* and *valid* time. According to [2], transaction time is the time during which a piece of data is recorded in a relation and may be retrieved. Valid time is a time during which a fact is true in the modeled reality. The valid time can be in the future or in the past.

There are two major directions [3] in the development of spatio-temporal data structures. The first direction is space-driven structures, which have indexing based upon the partitioning of the embedding multi-dimensional space into cells, independent of the distribution of the data in this space. An example of such a space-driven data structure is multiversion linear quadtree for storing spatio-temporal data. The other direction for storing spatio-temporal data is data-driven structures. These structures partition the set of the data objects, rather than the embedding space. Examples of data-driven structures include those based upon R-trees and its variants.

In this chapter, we will discuss both space-driven and data-driven data structures for storing spatio-temporal data. We will initiate our discussion with multiversion linear quad tree space driven data structure.

22.2 Overlapping Linear Quadtree

Tzouramanis, Vassilakopoulos and Manolopoulos in [4] proposed Multiversion linear quadtrees (MVLQ), also called overlapping linear quad trees, which are analogous to Multiversion B-trees (MVBT) [5], but with significant differences. Instead of storing transaction time for each individual object in MVBT, an MVLQ consolidates object descriptors that share a same transaction time. As it will be evident from the following discussion, these object descriptors are code words derived from a linear representation of multiversion quadtrees.

The idea of storing temporal information about the objects is based upon including a parameter for transaction time for these objects. Each object is given a unique time-stamp T_i (transaction time), where $i \in [1..n]$ and n is the number of objects in the database. This time stamp implicitly associates a time value to each record representing the object. Initially, when the object is created in the database, this time-interval is set equal to $[T_i, *)$. Here, '*' refers to *as-of-now*, a usage indicating that the current object is valid until a unspecified time in the future, when it would be terminated (at that time, * will be replaced by the T_j, $j \in [1..n]$ and $j > i$, the time-stamp at the time when the object is terminated).

Before we proceed further, let us gather some brief background in regional Quadtrees, commonly called quadtrees, a spatial data structure. In the following discussion it is assumed that a Spatio-temporal data structure (STDS) is required to be developed for a sequence of evolving regional images, which are stored in an image database. Each image is assumed to be represented as a $2^N \times 2^N$ matrix of pixel values, where N is a positive integer.

Quadtree is a modification of T-pyramid. Every node of the tree, except the leaves, has four children (NW: north-western, NE: north-eastern, SW: south-western, SE: south-eastern). The image is divided into four equal quadrants at each hierarchical level, but it is not necessary to store nodes at all levels. If a parent node has four children of the same homogeneous (for example, intensity) value, it is not necessary to record them. Figure 22.1 represents an image and its corresponding quadtree representation. Quadtree is a memory-resident data structure, with each node being stored as a record pointing to its children. However, when the represented object becomes too large it may not be possible to store

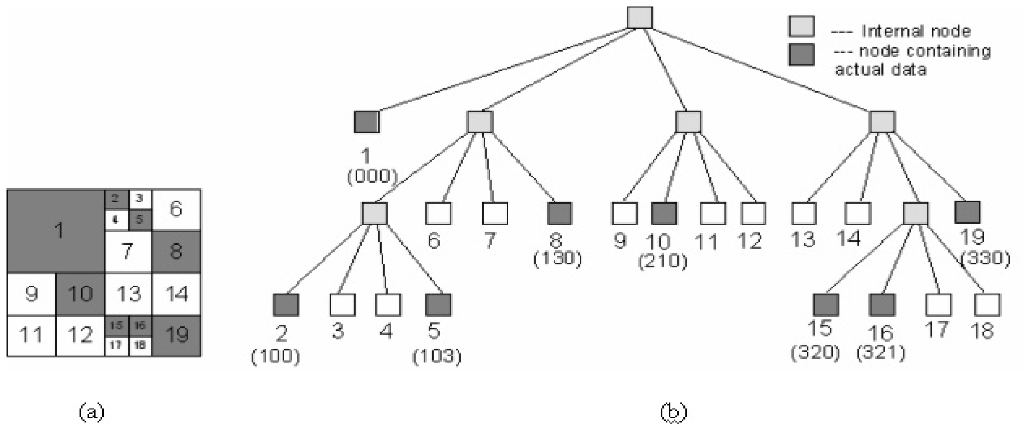


FIGURE 22.1: (a) A binary $2^3 \times 2^3$ image (b) Its corresponding region quadtree. The location codes for the homogeneous codes are indicated in brackets.

the entire quadtree in the main memory. The strategy that is followed at this point is that the homogeneous components of the quadtrees are stored in a B+ tree, a secondary memory data structure, eliminating the need of pointers in quadtree representation. In this case, addresses representing the location of the homogeneous node and its size constitute a record. Such a version of Quadtree is called linear region quadtree [6].

Several linear representations of regional quadtrees have been proposed, but fixed length (FL), fixed length-depth (FD) and variable length (VL) versions have attracted most attention. We refer the interested reader to [6] for details on these representations, while we concentrate on the FD representation. In this representation, each homogeneous node is represented by a pair of two codes, *location code* and *level number*. Location node C denotes the correct path to this node when traversing the Quadtree from its root till the appropriate leaf is reached and a level number L refers to the level at which the node is located. This makes up the fixed length-depth linear implementation of Quadtree.The quadrants NW, NE, SW and SE are represented as 0, 1, 2 and 3 respectively. The location codes for the homogeneous nodes of a quadtree are presented in Figure 22.1.

A persistent data structure (Chapter 31) [7] is one in which a change to the structure can be made without eliminating the old version, so that all versions of the structure persist and can at least be accessed (the structure is said to be partially persistent) or even modified (the structure is said to be fully persistent). Multi-Version Linear Quad-tree (MVLQ) is an example of a persistent data structure, in contrast to Linear Quadtree, a transient data structure. In MVLQ each object in the database is labeled with a time for maintaining the past states. MVLQ couples time intervals with spatial objects in each node.

The leaf of the MVLQ contains the data records of the format:

$$< \quad (C, L),\ T,\ EndTime >$$

where,

- C is the location code of the homogeneous node of the region Quadtree,
- L is the level of the region quad tree at which the homogeneous node is present,
- T represents the time interval when the homogeneous node appears in the image sequence,

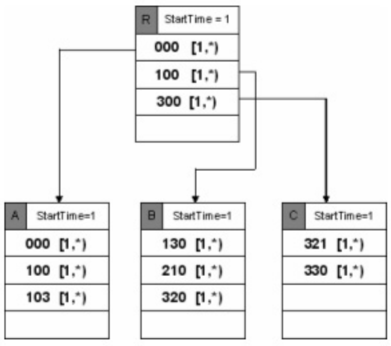


FIGURE 22.2: MVLQ structure after the insertion of image given in Figure 22.1.a.

- *EndTime* is Transaction time when the homogeneous leaf node becomes historical.

The non-leaf nodes contain entries of the form [4]:

$$< \quad C', P', ptr, StartTime>$$

where,

- C' is the smallest C recorded in that descendant node,
- P' is the time interval that expresses the lifespan of the latter node,
- *Ptr* is a pointer to a descendant node,
- *StartTime* is the time instant when the node was created.

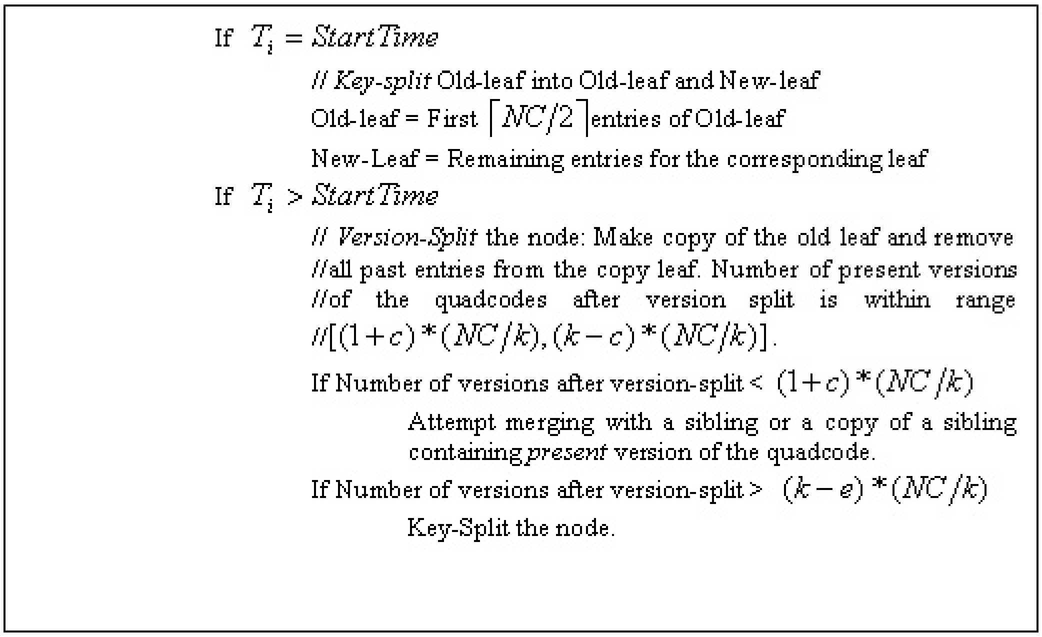
The MLVQ structure after the insertion of image given in Figure 22.1.a is given in Figure 22.2.

In addition to the main memory structure of the MVLQ described above, it also contains two additional main memory sub-structures: *Root*table* and Depth First expression (*DF-expression*) of the last inserted object.

*Root*table:* MVLQ has one tree for each version of the data. Consequently, each version can be reached through its root which can be identified by the time-interval of the version that it represents and a pointer to its location. If T'' is the time interval of the root, given by T''=$[T_i, T_j)$, where $i, j \in [1..n]$, i<j, and Ptr' is a pointer to the physical address of the root, then each record in the *Root*table* identifying a root is represented in the following form:

$$< T'', ptr' >$$

DF-expression of the last inserted object [4,8]: The purpose of Depth first expression (DF-expression) is to contain the pre-fetched location of the homogeneous nodes of last insert data object. The advantage of this storage is that when a new data object is being inserted, the stored information can be used to calculate the cost of deletions, insertions and/or updates. If the DF-expression is not stored, then there is an input/output cost associated with locating the last insert object's homogeneous nodes locations. The DF-expression is an array representation of preorder traversal of the last inserted object's Quadtree.



```
If $T_i$ = $StartTime$
        // Key-split Old-leaf into Old-leaf and New-leaf
        Old-leaf = First $\lceil NC/2 \rceil$ entries of Old-leaf
        New-Leaf = Remaining entries for the corresponding leaf
If $T_i$ > $StartTime$
        // Version-Split the node: Make copy of the old leaf and remove
        //all past entries from the copy leaf. Number of present versions
        //of the quadcodes after version split is within range
        //$[(1+c)*(NC/k), (k-c)*(NC/k)]$.
        If Number of versions after version-split < $(1+c)*(NC/k)$
                Attempt merging with a sibling or a copy of a sibling
                containing present version of the quadcode.
        If Number of versions after version-split > $(k-e)*(NC/k)$
                Key-Split the node.
```

FIGURE 22.3: Algorithm for an insertion of an object in MVLQ.

22.2.1 Insertion of an Object in MVLQ

The first step in insertion of a quadcode of a new object involves identifying the corresponding leaf node. If the corresponding leaf node is full, then a node overflow [4] occurs. Two possibilities may arise at this stage, and depending on the *StartTime* field of the leaf, a split may be initiated. If *NC* is the node capacity, k and c are integer constants (greater than zero), then the insertion is performed based on the algorithm presented in Figure 22.3.

22.2.2 Deletion of an Object in MVLQ

The algorithm for the deletion of an object from MVLQ is straightforward. If $T_i = StartTime$, then *physical deletion* occurs and the appropriate entry of the object is deleted from the leaf. If number of entries in the leaf $< \lceil NC/k \rceil$ (threshold), then a *node-underflow* is handled as in B+ tree with an additional step of checking for a sibling's *StartTime* for key redistribution. If$T_i > StartTime$, then *logical deletion* occurs and the temporal information of an entry between range $([T_i, *), [T_i, T_j))$ is updated. If an entry is logically deleted in a leaf with exactly $\lceil NC/k \rceil$ present quadcode versions, then a *version underflow* [5] occurs that causes a version split of the node, copying the present versions of its quadcodes into a new node. After version split, the number of present versions of quadcodes is below (1+e) $\lceil NC/k \rceil$ and a merge is then attempted with a sibling or a copy of that sibling.

22.2.3 Updating an Object in MVLQ

Updating a leaf entry refers to update of the field L(level) of the object's code. This is implemented in a two-step fashion. First, a logical deletion of the entry is performed. Second, the new version is inserted in place of that entry through the steps outlined above.

22.3 3D R-tree

In the previous section a space driven, multi-version linear Quadtree based spatio-temporal data structure was presented. In this section we discuss a data driven data structure that partitions the set of objects for efficient spatio-temporal representation.

Theidoridis, Vazirgiannis and Sellis in [9] have proposed a data structure termed 3D R-tree for indexing spatio-temporal information for large multimedia applications. The data structure is a derivation of R-trees (Chapter 21) whose variants have been demonstrated to be an efficient indexing schema for high-dimensional spatial objects.

In [9] Theidoridis *et al.* have adopted a set of operators defined in [10] to represent possible topological-directional spatial relationships between two 2-dimensional objects. An anthology of 169 relationships $R_{i\_j}(i \in [1..13], j \in [1..13])$ can represent a complete set of spatial operators. Figure 22.4 represents these relations. An interested reader can find a complete illustration of these topographical relations relationships in [10]. To represent the temporal relationships, a set of temporal operators defined in [11] are employed. Any spatio-temporal relationships among objects can be found using these operators. For Example, object Q to appear 7 seconds after the object P, 14cm to the right and 2cm down the right bottom vertex of object P can be represented as the following composition tuple:

$$R_t = P[(r_{13\_13}, v3, v2, 14, 2), (-7- >)]Q$$

where $r_{13_13}$is the corresponding spatial relationship, $(-7- >)$ is the temporal relationship between the objects, $v3$ and $v4$ are the named vertices of the objects while (14, 2) are their spatial distances on the two axes.

Theidoridis *et al.* have employed the following typical spatio-temporal relationships to illustrate their indexing schema [9]. These relationships can be defined as *spatio-temporal operators*.

- *overlap_during*(a,*b*): returns the catalog of objects *a* that spatially overlap object *b* during its execution.
- *overlap_before*(*a*,*b*): returns the catalogue of objects *a* that spatially overlap object *b* and their execution terminates before the start of execution of *b*.
- *above_during*(*a*,*b*): returns the catalogue of objects *a* that spatially lie above object *b* during the course of its execution.
- *above_before*(*a*,*b*): returns the catalogue of objects *a* that spatially lie above object *b* and their execution terminates before the start of execution of *b*.

Spatial and Temporal features of objects are typically identified by a six dimensions (each spatio-temporal object can be perceived as a point in a 6-dimensional space):
$(x_1, x_2, y_1, y_2, t_1, t_2)$, where
(x_1, x_2) : Projection of the object on the horizontal plane.
(y_1, y_2) : Projection of the object on the vertical plane.
(t_1, t_2) : Projection of the object on the time plane.

In a naïve approach, these object descriptors coupled by the object *id* (unique for the object that they represent) can be stored sequentially in a database. An illustration of such an organization is demonstrated in Figure 22.5.

Such a sequential schema has obvious demerits. Answering of spatial temporal queries, such as one described above, would require a full scan of the data organization, at least once. As indicated before, most spatio-temporal databases suffer from the curse of dimensionality

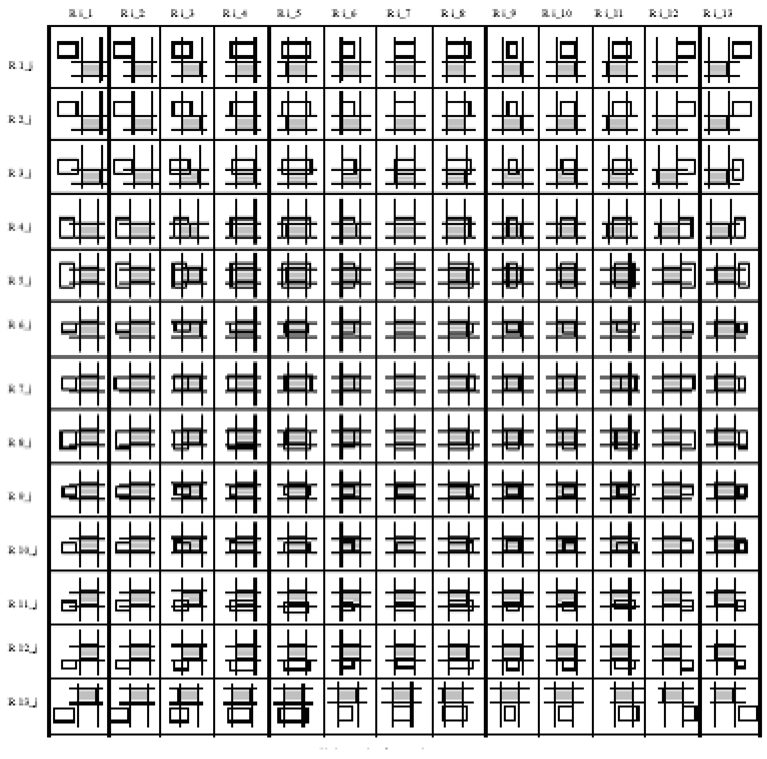


FIGURE 22.4: Spatial relationships between two objects covering directional-topological information [4].

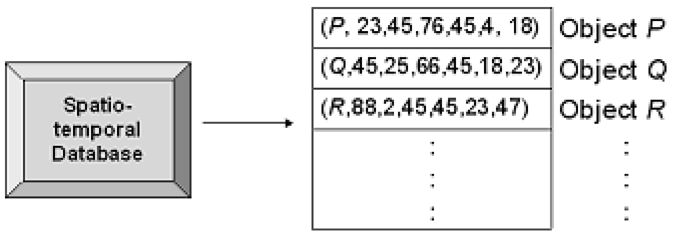


FIGURE 22.5: Schema for sequential organization of spatio-temporal data.

and sequential organization of data exhibits this curse through depreciated performance, rather than reducing it.

In another schema, two indices can be maintained to store spatial and temporal components separately. Specifically, they can be organized as follows.

1. Spatial index: An index to store the size and coordinates of the objects in two dimensions
2. Temporal index: A one-dimensional index for storing the duration and start/stop time for objects in one dimension.

R-trees and their variants have been demonstrated to be efficient for storing n-dimensional data. Generally speaking, they could be used to store the spatial space components in a 2D R-tree and temporal components in a 1D R-tree. This schema is better than sequential search, since a tree structure provides a hierarchical organization of data leading to a logarithmic time performance. More details on R-trees and its variants can be found in Chapter 21.

Although this schema is better than sequential searching, it still suffers from a limitation [9]. Consider the query *overlap_during*, which would require that both the indices (spatial 2D R-tree and temporal 1D R-tree) are searched individually (in the first phase) and then the intersection of the recovered answer sets from each of the indices is reported as the index's response to the query. Access to both the indices individually and then post-intersection can cumulatively be a computationally expensive procedure, especially when each of these indices are dense. Spatial-joins [9,12] have been proposed to handle queries on two indexes, provided these indexing schemas adopt the same spatial data structure. It is not straightforward to extend these to handle joins in two varieties of spatial data structures. Additionally, there might be possible inherent relationships between the spatial descriptors and the temporal descriptors resident in two different indexes which can be learned and exploited for enhanced performance. Arranging and searching these descriptors separately may not be able to exploit these relationships. Hence, a unified framework is needed to present spatio-temporal components preferably in a same indexing schema.

Before we proceed further, let us briefly discuss the similarity search procedure in R-trees. In R-trees Minimum bounding boxes (MBB); or minimum bounding rectangles in two-dimensions) are used to assign geometric descriptors to objects for similarity applications, especially in data mining [14]. The idea behind usage of MBB in R-trees is the following. If two objects are disjoint, then their MBBs should be disjoint and if two objects overlap, then their MBB should definitely overlap. Typically, a spatial query on a MBB based index involves the following steps.

1. Searching the index: This step is used to select the answer and some possible false alarms from a given data set, ignoring those records that cannot possibly satisfy the query criterion.

2. Dismissals of false alarms: The input to this step is the resultant of the index searching step. In this step the false alarms are eliminated to identify correct answers, which are reported as the query response.

Designing an indexing schema for a multimedia application requires design of a spatio-temporal indexing structure to support spatio-temporal queries. Consider the following scenario.

Example 22.1

An application starts with a video clip A located at point (1,7) relative to the application origin Θ. After 2 minutes an image B appears inside A with 1 unit above its lower horizontal edge and 3 units after its left vertical edge. B disappears after 2 minutes of its presence, while A continues. After 2 minutes of B's disappearance, an text window C appears 3 units below the lower edge of A and 4 units to the right of left edge of it. A then disappears after 1 minute of C's appearance. The moment A disappears, a small image D appears 2 units to the right of right edge of C and 3 units above the top edge of C. C disappears after 2 minutes of D's appearance. As soon as C disappears, a text box E appears 2 units below the lower edge of D and left aligned with it. E lasts for 4 minutes after which it disappears. D disappears 1 minute after E's disappearance. The application ends with D's disappearance. The spatial layout of the above scenario is presented in Figure 22.6a and the temporal layout is presented in Figure 22.6b.

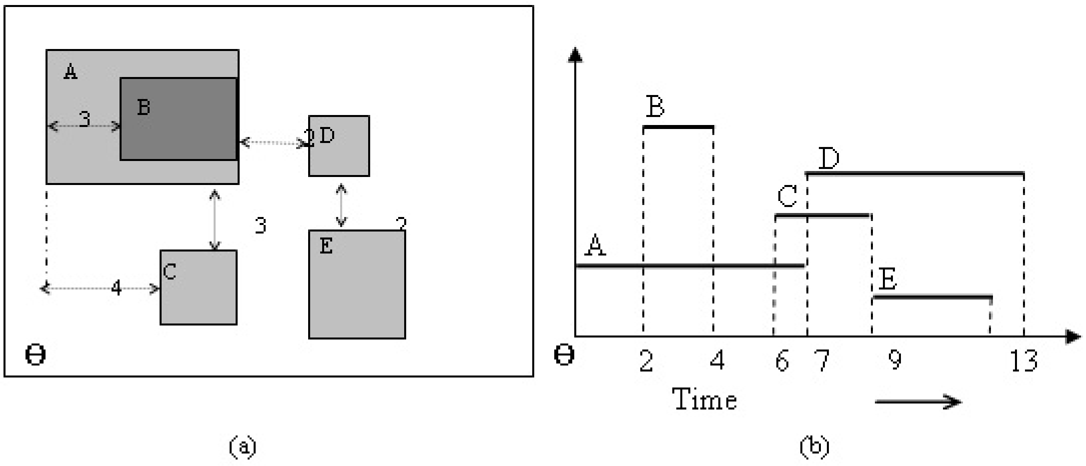


FIGURE 22.6: (a) Spatial layout of the multimedia database (b) Temporal layout of the multimedia database.

A typical query that can be posed on such a database of objects described above is "Which objects overlap the object A during its presentation?". In [9], the authors have proposed a unified schema to handle queries on a spatio-temporal database, such as the one stated above. The schema amalgamates the spatial and temporal components in a single data structure such as R-trees to exhibit advantages and performance gains over the other schemas described above. This schema amalgamates need of spatial joins on two spatial data structures besides aggregating both attributes under a unified framework. The idea for representation of spatio-temporal attributes is as follows. If an object which initially lies at point (x_a, y_a)during time $[t_a, t_b)$and at (x_b, y_b) during $[t_b, t_c)$, it can be modeled by two lines $[(x_a, y_a, t_a), (x_a, y_a, t_b))$ and $[(x_b, y_b, t_b), (x_b, y_b, t_c))$. These lines can be presented in a hierarchical R-tree index in three dimensions. Figure 22.7 presents the unified spatial-temporal schema for the spatial and temporal layouts of the example presented above.

22.3.1 Answering Spatio-Temporal Queries Using the Unified Schema

Answering queries on the above presented unified schema is similar to handling similarity queries using R-trees. Consider the following queries [9]:

Query-1: "Find all the objects on the screen at time T_2" (Spatial layout query). This query can be answered by considering a rectangle $Q1$ (Figure 22.7) intersecting the time-axis at exactly one point, T_2.

Query-2: "Find all the objects and their corresponding temporal duration between the time interval (T_0,T_1)" (temporal layout query). This query can be answered by considering a box $Q2$(Figure 22.7) intersecting the time-axis between the intervals (T_0, T_1).

After we have obtained the objects enclosed by the rectangle $Q1$ and box $Q2$, these objects are filtered (in main memory) to obtain the answer set.

Query-3: "Find all the objects and their corresponding spatial layout at time T=3 minutes". This query can be answered by looking at the screenshot of spatial layout of objects that were present at the given instant of time. The resultant would be the list of objects and their corresponding spatial descriptors. The response to this query is given in Figure 22.8a.

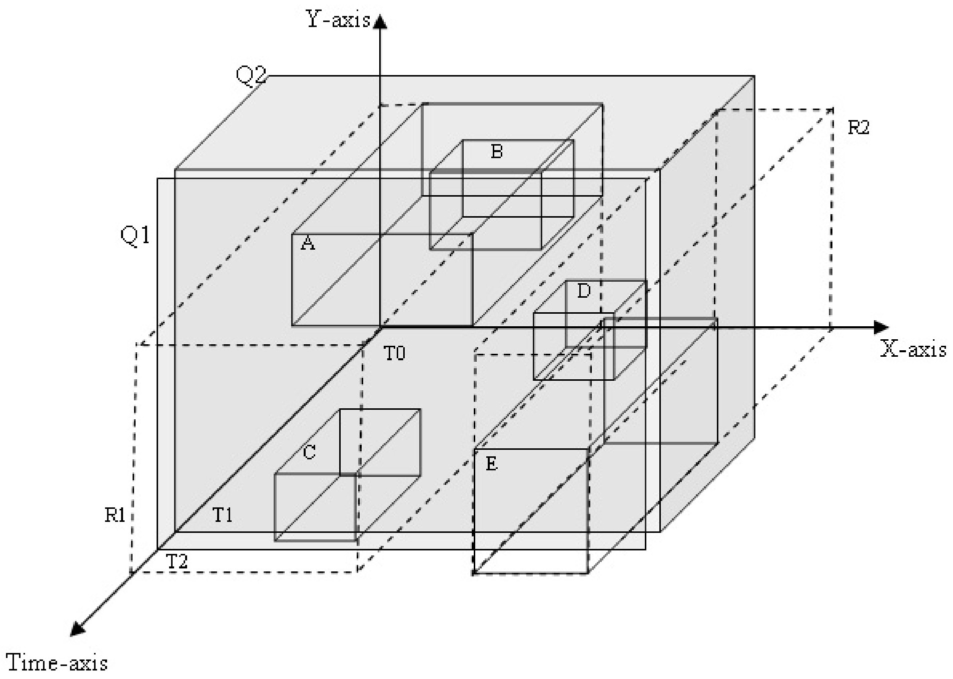


FIGURE 22.7: A unified R-tree based schema for storing spatial and temporal components.

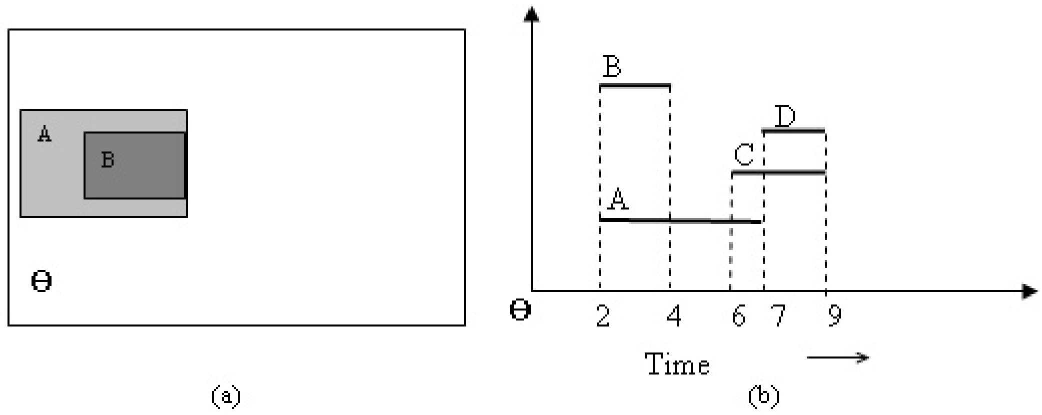


FIGURE 22.8: (a) Response to Query-3 (b) Response to Query-4.

Query-4: Find the temporal layout of all the objects between the time interval ($T_1 = 2, T_2 = 9$). This query can be answered by drawing a rectangle on the unified index with dimensions $(X_{max} - 0) \times (Y_{max} - 0) \times (T_2 - T_1)$. The response of the query is shown in Figure 22.8b.

22.3.2 Performance Analysis of 3D R-trees

Theodoridis *et al.* in [9] analyzed the performance of the proposed R-trees using the expected retrieval cost metric that they presented in [15]. An interested reader is referred to [15,9] for details on this metric. Based on the analytical model of this metric, it is asserted that one can estimate the retrieval cost of an *overlap query* based on the information attainable from the query window and data set only. In the performance analysis it was demonstrated that since the expected retrieval cost metric expresses the expected performance of R-trees on overlapping queries, the retrieval of spatio-temporal operators using R-trees is cost-equivalent to the cost of retrieval of an overlap query using an appropriate query window Q. Rigorous analysis on 10,000 objects asserted the following conclusions in [9]:

1. For the operators with high selectivity (*overlap*, *during*, *overlap_during*), the proposed 3D R-trees outperformed sequential search at a level of one to two orders of magnitude.
2. For operators with low selectivity (*above*, *before*, *above_before*), the proposed 3D R-trees outperformed sequential search by factors ranging between 0.25-0.50 fraction of the sequential cost.

22.3.3 Handling Queries with Open Transaction-Times

In the previous section although 3D R-trees were demonstrated to be very efficient compared to sequential search, it suffers from a limitation. The transaction times presented as creation and termination time of an object are expected to be known *a priori* before they can be stored and queries from the index. However, in most practical circumstances the duration of existence of an object is not known. In other words, when the object is created, all we know is that it would remain valid *until changed.* The concept of *until changes* is a well-discussed issue (see [16,17]). Data structures like R-trees and also its modified form of 3D R-tree are typically not capable of handling such queries having open transaction times. In the next and the following section, we discuss two spatio-temporal data structures capable of handling queries with open transaction times.

22.4 2+3 R-tree

Nascimento *et al.* in [17] have proposed a solution to the problem of handling objects with open transaction times. The idea is to split the indexing of these objects into two parts: one for two-dimensional points and other for three-dimensional lines. Two-dimensional points store the spatial information about the objects and their corresponding start time. Three-dimensional lines store the historical information about the data. Once the 'started' point stored in 2D R-tree becomes closed, a corresponding line is constructed to be stored in the 3D R-tree and the point entry is deleted from the 2D R-tree. It should be understood that both trees are expected to be searched depending on the time stamp at which the query is posed. If the end times for each of the spatial points is known *a priori*, then the need of a 2-D can be completely eliminated reducing the structure to a 3D R-tree.

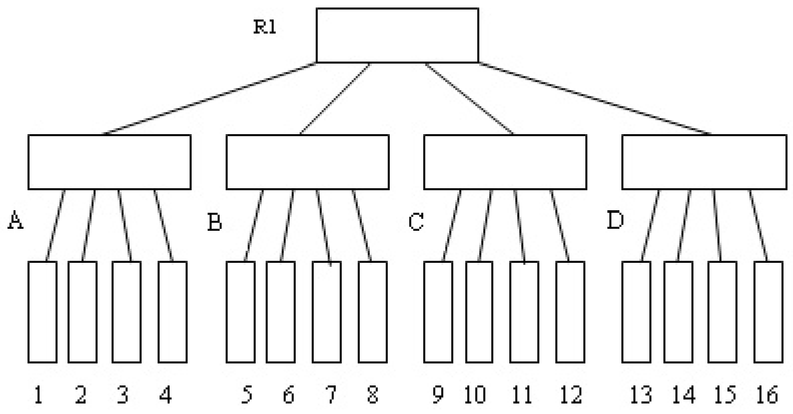


FIGURE 22.9: A R-tree at time T0.

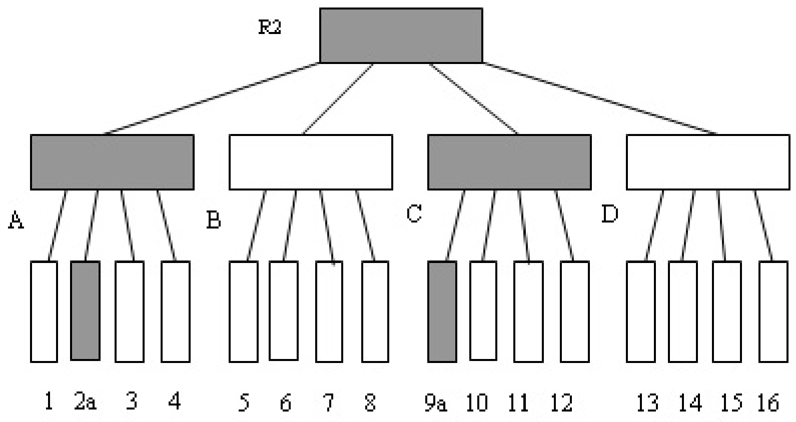


FIGURE 22.10: A R-tree at time T1. The modified nodes are in dark boxes.

22.5 HR-tree

One possible way to index spatio-temporal data is to build one R-tree for each timestamp, which is certainly space-inefficient. The demerit of 2+3 R-tree is that it requires two stores to be searched to answer a query. The HR-tree [18] or Historical R-tree "compresses" multiple R-trees, so that R-tree nodes which are common in two timestamps are shared by the corresponding R-trees. The main idea is based on the expectation that a vast majority of indexed points do not change their positions at every time stamp. In other words, it is reasonable to expect that a vast majority of spatial points will 'share' common nodes in respective R-trees. The HR-tree exploits this property by keeping a 'logical' linkage to a previously present spatial point if it is referenced again in the new R-tree. Consider two R-trees in Figure 22.9 and Figure 22.10 at two different time stamps (T0, T1). An HR-tree for these R-trees is given in Figure 22.11.

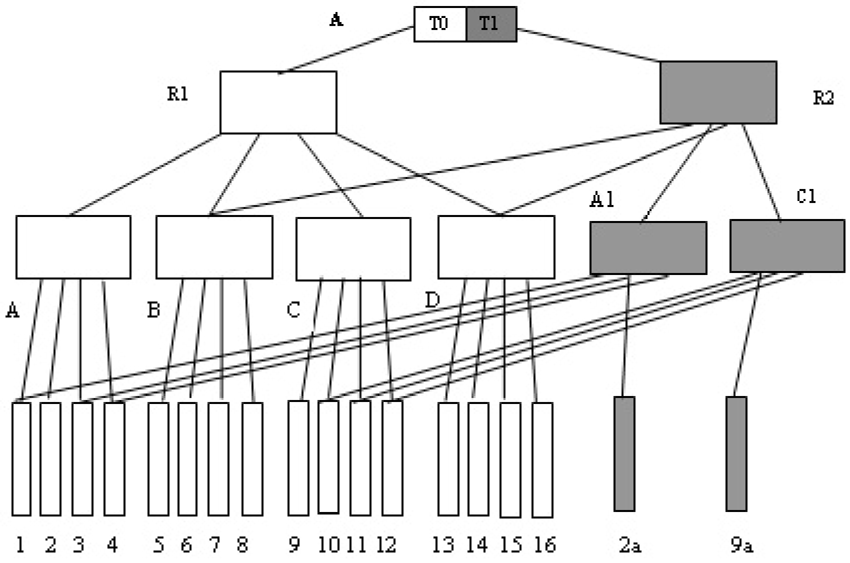


FIGURE 22.11: The HR-tree.

22.6 MV3R-tree

The MV3R-tree was proposed by Yufei and Papadias in [19] and is demonstrated to process time-stamp and interval queries efficiently. Timestamp queries discover all objects that intersect a window query at a particular timestamp. On the other hand, interval queries include multiple timestamps in the discovered response. While the spatial data structures proposed before have been demonstrated [9,19] to be suitable for either of these queries (or a subset of it), none of them have been established to process both timestamp and interval queries efficiently. MV3R-tree has addressed limitations in terms of efficiency of the above trees in handling both of these queries. MV3R-tree consists of a multi-version R-tree (MVR-tree) and an auxiliary 3D R-tree built on the leaves of the MVR-tree. Although the primary motivation behind the development of MV3R-tree are Multiversion B-trees [20] (which are extended from B+ trees), they are significantly different from these versions. Before we proceed further, let us understand the working of a multiversion B-tree.

A typical entry of a multiversion B-tree takes the following form $< id, time_{start}, time_{end}, P >$. For non-leaf nodes, $time_{start}$ and $time_{end}$ are the minimum and maximum values respectively in this node and P is a pointer to the next level node. For a leaf node, the time-stamps $time_{start}$ and $time_{end}$ indicate when the object was insert and deleted from the index and pointer P points to the actual record with a corresponding id value. At time $time_{current}$, the entry is said to be *alive* if $time_{start} < time_{current}$, otherwise *dead* [19]. There can be multiple roots in a Multiversion B-tree, where each root has a distinguishing time-range it represents. A search on the tree begins at identifying the root within which the time-stamp of the query belongs. The search is continued based on the id, $time_{start}$ and $time_{end}$. A *weak version condition* specifies that for each node, except the root, at least $K.T$ entries are alive at time t, where K is the node capacity and T is the tree parameter. This condition ensures that entries alive at the identical time-stamps are in a majority of the cases assembled together to allow easy time stamp queries.

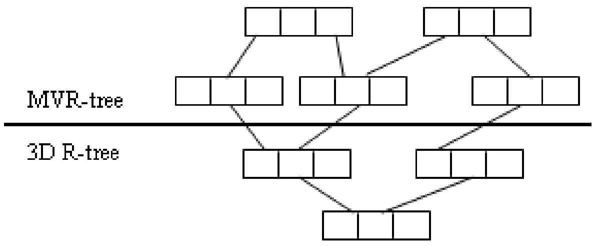


FIGURE 22.12: A MV3R-tree.

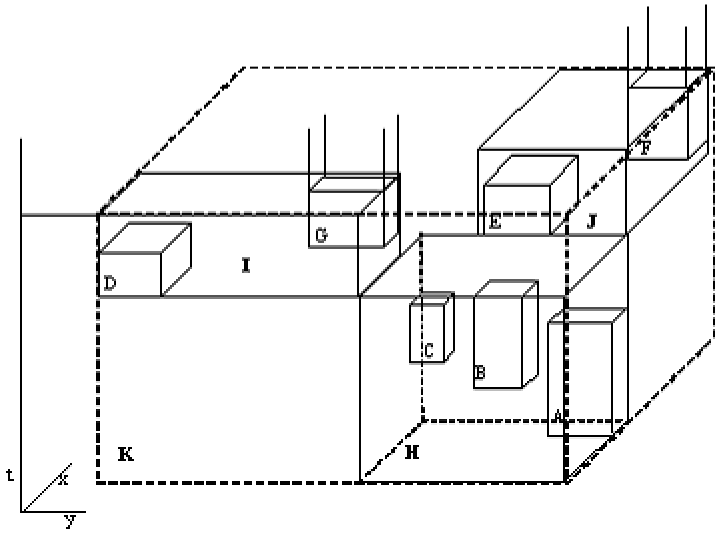


FIGURE 22.13: A 3-D visualization of MVR-tree.

3D R-tree are very space-efficient and can handle long interval queries efficiently. However, timestamp and short-interval queries using 3D R-trees are expensive. In addition to this, 3D R-trees do not include a methodology such as the weak version condition to ensure that each node has a minimum number of live entries at a given timestamp. HR-trees [18], on the other hand, maintain an R-tree (or its derivative) for each timestamp and the timestamp query is directed to the corresponding R-tree to be searched within it. In other words, the query disintegrates into an ordinary window query and is handled very efficiently. However, in case of an interval query, several timestamps should search the corresponding trees of all the timestamps constituting the interval. The original work in HR-tree did not present a schema for handling interval queries; however, authors in [19] have proposed a solution to this problem by the use of negative and positive pointers. The performance of this schema is then compared with MV3R-tree in [19]. It is demonstrated that MV3R-trees outperform HR-trees and 3D R-trees even in extreme cases of only timestamp and interval-based queries.

Multiversion 3D R-trees (MV3R-trees) combine a multiversion R-tree (MVR-tree) and a small auxiliary 3D R-tree built on the leaf nodes of the MVR-tree as shown in Figure 22.12. MVR-trees maintain multiple R-trees and has entries of the form $< MBR$,

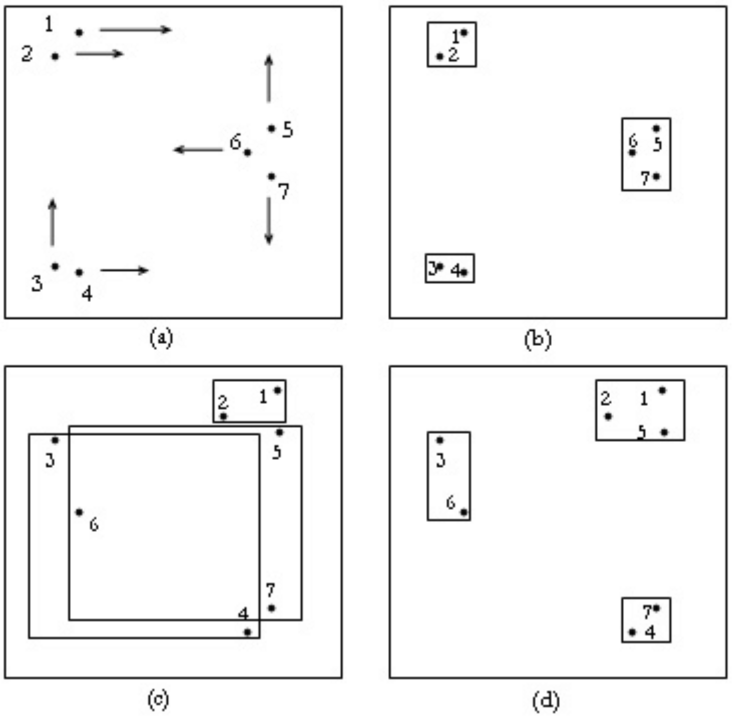


FIGURE 22.14: Moving data points and their leaf-level MBBs at subsequent points.

$time_{start}$, $time_{end}$, $P >$, where MBR refers to Minimum bounding rectangle and other entries are similar to B+ trees. An example of a 3-D visualization of a MVR-tree of height 2 is shown in Figure 22.13. The tree consists of Object cubes (A-G), leaf nodes (H-J) and root of the tree K.Detailed algorithms for insertion and deletion in these trees are provided in [19]. Time-stamp queries can be answered efficiently using MVR-trees. An auxiliary 3D R-tree is built on the leaves of the MVR-tree in order to process interval queries. It is suggested that for a moderate node capacity, the number of leaf nodes in an MVR-tree is much lower than the actual number of objects, hence this tree is expected to be small compared to a complete 3D R-tree.

Performance analysis has shown that the MV3R-tree offers better tradeoff between query performance and structure size than the HR-tree and 3D R-tree. For typical situations where workloads contain both timestamp and interval queries, MV3R-trees outperform HR-trees and 3D R-trees significantly. The incorporation of the auxiliary 3D R-tree not only accelerates interval queries, but also provides flexibilities towards other query processing, such as, spatio-temporal joins.

22.7 Indexing Structures for Continuously Moving Objects

Continuously moving objects pose new challenges to indexing technology for large databases. Sources for such data include GPS systems, wireless networks, air-traffic controls etc. In the previous sections we have outlined some indexing schemas that could efficiently index spatio-temporal data types but some other schemas have been developed specifically for answering predictive queries in a database of continuously moving objects. In this section we discuss an assortment of such indexing schemas.

Databases of continuously moving objects have two kinds of indexing issues: Storing the historical movements in time of objects and predicting the movement of objects based in previous positional and directional information. Such predictions can be made more reliably for a future time T_f, not far from the current timestamp T_f. As T_f increases, the predictions become less and less reliable since the change of trajectory by a moving object results in inaccurate prediction. Traditional indexing schemas such as R*-trees are successful in storing multidimensional data points but are not directly useful for storing moving objects. An example[21] of such kind of system is shown in Figure 22.14. For simplicity a two dimensional space is illustrated, but practical systems can have larger dimensions. First part of figure shows multiple objects moving in different directions. If traditional R*-trees is used for indexing these data, the minimum bounding boxes for the leaf level of the tree are demonstrated in Figure 22.14b. But these objects might be following different trajectories (as shown by arrows in Figure 22.14a and at subsequent time stamps, the leaf level MBBs might change in size and position, as demonstrated in Figure 22.14c and Figure 22.14d.

Since traditional indexing methods are not designed for such kind of applications, some novel indexing schemas are described in [21-24] to handle such data types and queries imposed on them.

22.7.1 TPR-tree

Saltenis et al in [21] proposed TPR-tree, an acronym for Time-Parameterized R*-tree, based on the underlying principles of R-tree. TPR-tree indexes the current and future anticipated positions of moving objects in one, two and three dimensions. The basic algorithms of R*-trees are employed for TPR-tree with a modification that the leaf and non-leaf minimum bounding rectangles are now augmented with velocity vectors for these rectangles. The velocity vector for an edge of the rectangle is chosen so that the object remains inside the moving rectangle. TPR-tree can typically handle the following three types of queries,

- Timeslice Query: A query Q specified by hyper-rectangle R located at time point t.
- Window Query: A query Q specified by hyper-rectangle R covering an interval from $[T_a, T_b]$.
- Moving Query: A query Q specified by hyper-rectangles R_a and R_b at different times T_a and T_b, forming a trapezoid.

Figure 22.15 shows objects o1, o2, o3 and o4 moving in time. The trajectories of these objects are shifting, as shown in figure. The three types of queries as described above are illustrated in this figure. $Q0$ and $Q1$ are timeslice queries, $Q2$ and $Q3$ are window queries and $Q4$ is a moving query.

The structure of TPR-tree is very similar to R*-tree with leaves consisting of position and pointer of the moving object. The nodes of the tree consist of pointers to subtree and bounding rectangles for the entries in subtree. TPR-trees store the moving objects as linear function of time with time-parameterized bounding rectangles. The index does not consist of points and rectangles for time stamp older than current time. TPR-tree differs from the R*-trees in how its insertion algorithms group points into nodes. While in R*-trees the heuristics of the minimized area, overlap, and margin of bounding rectangles are used to assign points to the nodes of the tree, in case of TPR-trees these heuristics are replaced

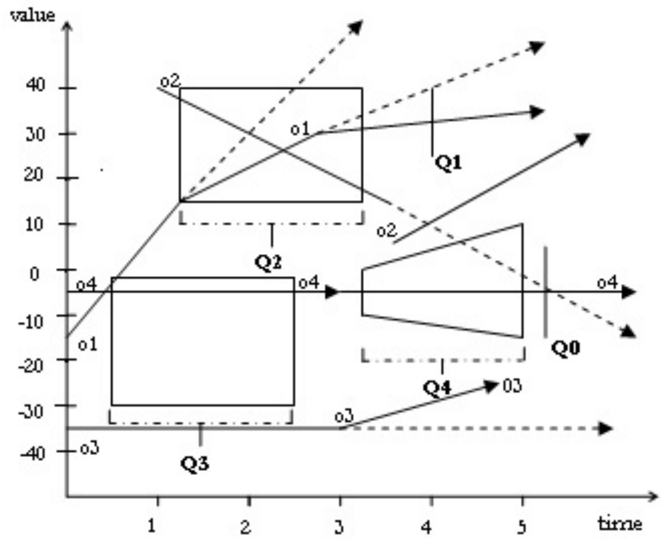


FIGURE 22.15: Types of queries on one-dimensional data.

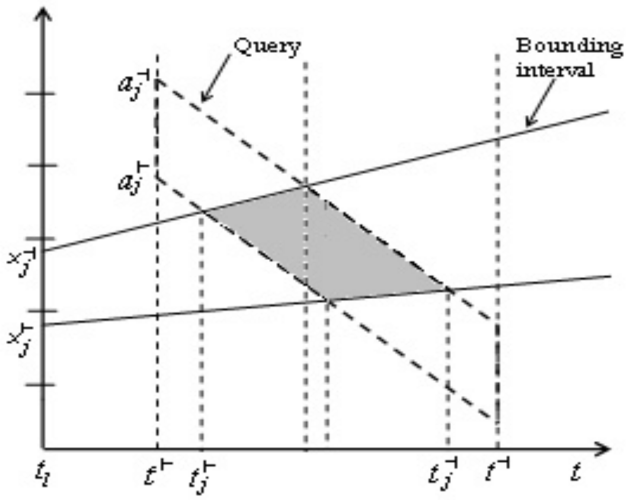


FIGURE 22.16: A bounding interval and a query imposed on the TPR-tree.

by their respective integrals, which are representative their temporal component. Given an objective function $F(t)$, the following integral is expected to be minimized [21].

$\int_{t_c}^{t_c+H} F(t)dt$, where t_c is the current time and H is the time horizon.

The objective function can be area or perimeters of the bounding rectangles, or could represent the overlap between these rectangles. Figure 22.16 represents a bounding interval and a query in the TPR-tree. The area of the shaded region in Figure 22.16 represents the time integral of the length of the bounding interval.

Saltenis *et al.* in [21] compared the performance of TPR-trees with load-time bounding rectangles, TPR-tree with update-time bounding rectangles and R-tree with a set of experiments with varying workloads. The results demonstrated that TPR-tree outperforms other approaches by considerable improvement. It was also demonstrated that tree does not degrade severely in performance with increasing time and it can be tuned to take advantage of a specific update rate.

22.7.2 R^{EXP}-tree

Saltenis and Jensen in [22] proposed R^{EXP}-tree a balanced, multi-way tree with a structure of R*-tree. R^{EXP}-tree is an improvement over TPR-tree, assuming that the some objects used in indexing expires after a certain period. These trees can handle realistic scenario where certain objects are no longer required, that is when they expire. By removing the expired entries and re-computing bounding rectangles, the index organizes itself to handle subsequent queries efficiently. This tree structure finds its application where the objects not reporting their position for a certain period, possibly implying that they are no more interested in the service.

The index structure of R^{EXP}-tree differs from TPR-tree in insertion and deletion algorithms for disposing the expired nodes. R^{EXP}-tree uses a 'lazy strategy' for deleting the expired entries. Another possible strategy is scheduled deletion of entries in TPR-trees. During the search, insertion and deletion operations, only the live entries are searched and expired entries are physically removed when the content of the node are modified and is written to the disk. Whenever an entry in internal node is deleted, the entire subtree is reallocated. The performance results demonstrated in [22] show that choosing the right bounding rectangles and corresponding algorithms for grouping entries is not straightforward and depends on the characteristics of the workloads.

22.7.3 STAR-tree

Procopiuc, Agarwal and Har-Peled in [23] propose a Spatio-Temporal Self-Adjusting R-tree or STAR-tree. STAR-tree indexing schema is similar to TPR trees with few differences. Specifically, STAR-tree groups points according to their current locations and may result in points moving with different velocities being included in the same rectangle. Scheduled events are used to regroups points to control the growth of such bounding rectangles. It improves the structure of TPR-tree by self-adjusting the index, whenever index performance degrades. Intervention of user is not needed for adjustment of the index and the query time is kept low even without continuously updating the index by positions of the objects. STAR-tree doesn't need periodic rebuilding of indexing and estimation of time horizon. It provides tradeoffs between storage and query performance and between time spent in updating the index and in answering queries. STAR-tree can handle not only the timeslice and range queries as those handled by TPR-trees, but also nearest neighbor queries for continuously moving objects.

22.7.4 TPR*-tree

TPR*-tree proposed by Tao, Papadias and Sun in [24] is an optimized spatio-temporal indexing method for predictive queries. TPR-tree, described in the previous section, does not propose an analytical model for cost estimation and query optimization and quantification of its performance. TPR*-tree assumes a probabilistic model that accurately estimates the number of disk accesses in answering a window query in a spatio-temporal index. The authors in [24] investigate the optimal performance of any data-partition index using the proposed model.

The TPR*-tree improves the performance of TPR-tree by employing a new set of insertion and deletion algorithms that minimize the average number of node accesses for answering a window query, whose MBB uniformly distributes in the data space. The static point interval query with the following constraints has been is optimized [24] using the TPR*-tree:

- MBB has a length $|Q^R| = 0$ on each axis.
- Velocity bounding rectangle is $\{0,0,0,0\}$.
- Query interval Q_I= $[0, H]$, where H is the horizon parameter.

It is demonstrated that the above choice of parameters leads to nearly-optimal performance independently of the query parameters. The experiments have also shown that TPR*-trees significantly outperforms the conventional TPR-tree under all conditions.

References

[1] B.-U. Pagel, F. Korn, C. Faloutsos, "Deflating the Dimensionality Curse using Multiple Fractal Dimensions," in *16th Intl. Conf. on Data Engineering (ICDE)*, San Diego, CA - USA, 2000.

[2] C. S. Jensen, J. Clifford, R. Elmarsi et al., "A Consensus Glossary of Temporal Database Concepts," *SIGMOD Record*, 23(1), pp. 52-64, March 1994.

[3] Mahdi Abdelguerfi, Julie Givaudan, Kevin Shaw, Roy Ladner, "The 2-3TR-tree, a trajectory-oriented index structure for fully evolving valid-time spatio-temporal datasets," *ACM-GIS 2002*: 29-34.

[4] T. Tzouramanis, M. Vassilakopoulos, and Y. Manolopoulos, "Multiversion linear quadtree for spatiotemporal data," in *Proc. ADBIS-DASFAA*.

[5] B. Becker, S. Gschwind, T. Ohler, B. Seeger and P. Widmayer, "An Asymptotically Optimal Multiversion B-tree," *The VLDB Journal*, Vol.5, No.4, pp.264-275, 1996.

[6] H. Samet, *The Design and Analysis of Spatial Data Structures*, Addison-Wesley,Reading MA, 1990.

[7] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan, "Making Data Structures Persistent," *Journal of Computer and System Sciences*, Vol.38, pp.86-124, 1989.

[8] E. Kawaguchi and T. Endo, "On a Method of Binary Picture Representation and its Application to Data Compression," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol.2, No.1, pp.27-35, 1980.

[9] Y. Theodoridis, M. Vazirgiannis, T. Sellis, "Spatio-Temporal Indexing for Large Multimedia Applications," *In Proceedings of the 3rd IEEE Conference on Multimedia Computing and Systems (ICMCS)*, 1996.

[10] D. Papadias, Y. Theodoridis, "Spatial Relations, Minimum Bounding Rectangles, and Spatial Data Structures," *International Journal of Geographic Information Systems*, 1996.

[11] M. Vazirgiannis, Y. Theodoridis, T. Sellis, "Spatio-Temporal Composition in Multime-

dia Applications," Technical Report KDBSLAB-TR-95-09, *Knowledge and Database Systems Laboratory*, National Technical University of Athens, 1995.

[12] M. G. Martynov, "Spatial Joins and R-trees," *ADBIS*, 1995: 295-304.

[13] J. Han and M. Kamber., *Data Mining: Concepts and Techniques.*, Morgan Kaufmann, 2000.

[14] Y. Theodoridis, D. Papadias, "Range Queries Involving Spatial Relations: A Performance Analysis," *Proceedings of the 2nd International Conference on Spatial Information Theory (COSIT)*, 1995.

[15] Y. Theodoridis, T. Sellis, "On the Performance Analysis of Multidimensional R-tree-based Data Structures," Technical Report KDBSLABTR-95-03, Knowledge and Database Systems Laboratory, National Technical University of Athens, Greece, 1995.

[16] J. Clifford, C. E. Dyreson, T. Isakowitz, C. S. Jensen, R. T. Snodgrass, "On the Semantics of "Now" in Databases," *ACM Transactions on Database Systems*, 22(2):171-214, 1997.

[17] M. Nascimento, R. Silva, and Y. Theodoridis, "Evaluation of access structures for discretely moving points," *In Proceedings of the International Workshop on Spatio-Temporal Database Management*, pp.171-188, 1999.

[18] M. Nascimento,and J. Silvia, "Towards Historical Rtrees," *In Proceedings of ACM Symposium on Applied Computing (ACM-SAC)*, (1998): pp. 235–240.

[19] Y. Tao, D. Papadias, "The MV3R-tree: A Spatio-Temporal Access Method for Timestamp and Interval Queries," *VLDB*, 2001.

[20] B. Becker, S. Gschwind, T. Ohler, B. Seeger, P. Widmayer, "An Asymptotically Optimal Multiversion B-Tree," *VLDB Journal* (1996), 5(4): 264-275.

[21] S. Saltenis, C. Jensen, S. Leutenegger and M. Lopez, "Indexing the Positions of Continuously Moving Objects," *In Proc. of the 19th ACM-SIGMOD Int. Conf. on Management of Data, Dallas, Texas* (2000).

[22] S. Saltenis and C. S. Jensen, "Indexing of Moving Objects for Location-Based Services," *TimeCenter* (2001), TR-63, 24 pages.

[23] C.M. Procopiuc, P. K. Agarwal, S. Har-Peled. STAR-Tree: An Efficient Self-Adjusting Index for Moving Objects. *4th Workshop on Algorithms Engineering* (2002).

[24] Tao, Y., Papadias, D., Sun, J. The TPR*-Tree: An Optimized Spatio-Temporal Access Method for Predictive Queries. *VLDB* (2003).

23

Kinetic Data Structures

Leonidas Guibas
Stanford University

23.1 Introduction

Motion is ubiquitous in the physical world, yet its study is much less developed than that of another common physical modality, namely shape. While we have several standardized mathematical shape descriptions, and even entire disciplines devoted to that area—such as *Computer-Aided Geometric Design* (CAGD)—the state of formal motion descriptions is still in flux. This in part because motion descriptions span many levels of detail; they also tend to be intimately coupled to an underlying physical process generating the motion (dynamics). Thus, until recently, proper abstractions were lacking and there was only limited work on algorithmic descriptions of motion and their associated complexity measures. This chapter aims to show how an algorithmic study of motion is intimately tied via appropriate data structures to more classical theoretical disciplines, such as discrete and computational geometry. After a quick survey of earlier work (Sections 23.2 and 23.3), we devote the bulk of this chapter to discussing the framework of *Kinetic Data Structures* (Section 23.4) [16, 32] and its many applications (Section 23.5). We also briefly discuss methods for querying moving objects (Section 23.6).

*A version of this chapter has also appeared in the *CRC Handbook of Discrete and Computational Geometry* (Chapter 50: Modeling Motion), 2nd Edition

23.2 Motion in Computational Geometry

Motion has not been studied extensively within the context of theoretical computer science. Until recently, there were only sporadic investigations of moving objects in the computational geometry literature. *Dynamic computational geometry* refers to the study of combinatorial changes in a geometric structure, as its defining objects undergo prescribed motions. For example, we may have n points moving linearly with constant velocities in $\mathcal{R}^2$, and may want to know the time intervals during which a particular point appears on their convex hull, the steady-state form of the hull (after all changes have occurred), or get an upper bound on how many times the convex hull changes during this motion. Such problems were introduced and studied in [13].

A number of other authors have dealt with geometric problems arising from motion, such as collision detection or minimum separation determination [35, 41, 42]. See also Chapter 56. For instance, [42] shows how to check in subquadratic time whether two collections of simple geometric objects (spheres, triangles) collide with each other under specified polynomial motions.

23.3 Motion Models

An issue with the above research is that object motion(s) are assumed to be known in advance, sometimes in explicit form (e.g., points moving as polynomial functions of time). Indeed, the proposed methods reduce questions about moving objects to other questions about derived static objects.

While most evolving physical systems follow known physical laws, it is also frequently the case that discrete events occur (such as collisions) that alter the motion law of one or more of the objects. Thus motion may be predictable in the short term, but becomes less so further into the future. Because of such discrete events, algorithms for modeling motion must be able to adapt in a dynamic way to motion model modifications. Furthermore, the occurrence of these events must be either predicted or detected, incurring further computational costs. Nevertheless, any truly useful model of motion must accommodate this *on-line* aspect of the temporal dimension, differentiating it from spatial dimensions, where all information is typically given at once.

In real-world settings, the motion of objects may be imperfectly known and better information may only be obtainable at considerable expense. The model of *data in motion* of [37] assumes that upper bounds on the rates of change are known, and focuses on being selective in using sensing to obtain additional information about the objects, in order to answer a series of queries.

23.4 Kinetic Data Structures

Suppose we are interested in tracking high-level attributes of a geometric system of objects in motion such as, for example, the convex hull of a set on n points moving in $\mathcal{R}^2$. Note that as the points move continuously, their convex hull will be a continuously evolving convex polygon. At certain discrete moments, however, the combinatorial structure of the convex hull will change (that is, the circular sequence of a subset of the points that appear on the hull will change). In between such moments, tracking the hull is straightforward: its geometry is determined by the positions of the sequence of points forming the hull. How can we know when the combinatorial structure of the hull changes? The idea is that we can focus on certain elementary geometric relations among the n points, a set of *cached* *assertions*, which altogether certify the correctness of the current combinatorial structure of the hull. If we have short-term information about the motion of the points, then we can predict failures of these assertions in the near future. Furthermore, we can hope to choose these certifying relations in such a way so that when one of them fails because of point motion, both the hull and its set of certifying relations can be updated locally and incrementally, so that the whole process can continue.

- **Kinetic data structure:** A kinetic data structure (KDS) for a geometric attribute is a collection of simple geometric relations that certifies the combinatorial structure of the attribute, as well as a set of rules for repairing the attribute and its certifying relations when one relation fails.
- **Certificate:** A certificate is one of the elementary geometric relations used in a KDS.
- **Motion plan:** An explicit description of the motion of an object in the near future.
- **Event:** An event is the failure of a KDS certificate during motion. If motion plans are available for all objects in a certificate, then the future time of failure for this certificate can be predicted. Events are classified as *external* when the combinatorial structure of the attribute changes, and *internal*, when the structure of the attribute remains the same, but its certification needs to change.
- **Event queue:** In a KDS, all certificates are placed in an event queue, according to their earliest failure time.

The inner loop of a KDS consists of repeated certificate failures and certification repairs, as depicted in Figure 23.1.

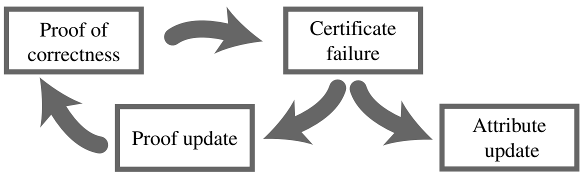
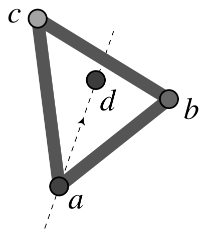


FIGURE 23.1: The inner loop of a kinetic data structure.

We remark that in the KDS framework, objects are allowed to change their motions at will, with appropriate notification to the data structure. When this happens all certificates involving the object whose motion has changed must re-evaluate their failure times.

23.4.1 Convex Hull Example

Suppose we have four points a, b, c, and d in $\mathcal{R}^2$, and wish to track their convex hull. For the convex hull problem, the most important geometric relation is the CCW predicate: $\text{CCW}(a, b, c)$ asserts that the triangle abc is oriented counterclockwise. Figure 23.2 shows a configuration of four points and four CCW relations that hold among them. It turns out that these four relations are sufficient to prove that the convex hull of the four points is the triangle abc. Indeed the points can move and form different configurations, but as long as the four certificates shown remain valid, the convex hull must be abc.



Proof of correctness:

- $\text{CCW}(a, b, c)$
- $\text{CCW}(d, b, c)$
- $\text{CCW}(d, c, a)$
- $\text{CCW}(d, a, b)$

FIGURE 23.2: Determining the convex hull of the points.

Now suppose that points a, b, and c are stationary and only point d is moving with a known plan, as shown in Figure 23.3. At some time t_1 the certificate $\text{CCW}(d, b, c)$ will fail, and at a later time t_2 $\text{CCW}(d, a, b)$ will also fail. Note that the certificate $\text{CCW}(d, c, a)$ will never fail in the configuration shown even though d is moving. So the certificates $\text{CCW}(d, b, c)$ and $\text{CCW}(d, a, b)$ schedule events that go into the event queue. At time t_1, $\text{CCW}(d, b, c)$ ceases to be true and its negation, $\text{CCW}(c, b, d)$, becomes true. In this simple case the three old certificates, plus the new certificate $\text{CCW}(c, b, d)$ allow us to conclude that convex hull has now changed to *abdc*.

Old proof	New proof
$\text{CCW}(a, b, c)$	$\text{CCW}(a, b, c)$
$\text{CCW}(d, b, c)$	$\text{CCW}(c, b, d)$
$\text{CCW}(d, c, a)$	$\text{CCW}(d, c, a)$
$\text{CCW}(d, a, b)$	$\text{CCW}(d, a, b)$

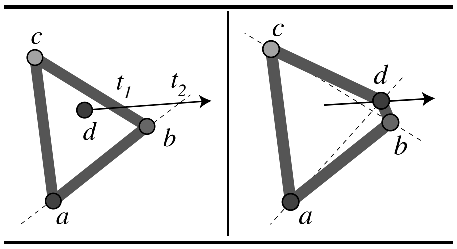


FIGURE 23.3: Updating the convex hull of the points.

If the certificate set is chosen wisely, the KDS repair can be a local, incremental process—a small number of certificates may leave the cache, a small number may be added, and the new attribute certification will be closely related to the old one. A good KDS exploits the continuity or coherence of motion and change in the world to maintain certifications that themselves change only incrementally and locally as assertions in the cache fail.

23.4.2 Performance Measures for KDS

Because a KDS is not intended to facilitate a terminating computation but rather an ongoing process, we need to use somewhat different measures to assess its complexity. In classical data structures there is usually a tradeoff between operations that interrogate a set of data and operations that update the data. We commonly seek a compromise by building indices that make queries fast, but such that updates to the set of indexed data are not that costly as well. Similarly in the KDS setting, we must at the same time have access to information that facilitates or trivializes the computation of the attribute of interest, yet we want information that is relatively stable and not so costly to maintain. Thus, in the same way that classical data structures need to balance the efficiency of access to the data with the ease of its update, kinetic data structures must tread a delicate path between "knowing too little" and "knowing too much" about the world. A good KDS will select a certificate set that is at once economical and stable, but also allows a quick repair of itself and the attribute computation when one of its certificates fails.

- **responsiveness:** A KDS is *responsive* if the cost, when a certificate fails, of repairing the certificate set and updating the attribute computation is small. By "small" we mean polylogarithmic in the problem size—in general we consider small quantities that are polylogarithmic or $O(n^\epsilon)$ in the problem size.
- **efficiency:** A KDS is *efficient* if the number of certificate failures (total number of events) it needs to process is comparable to the number of required changes in the combinatorial attribute description (external events), over some class of allowed motions. Technically, we require that the ratio of total events to external events is small. The class of allowed motions is usually specified as the class of *pseudo-algebraic* motions, in which each KDS certificate can flip between true and false at most a bounded number of times.
- **compactness:** A KDS is *compact* if the size of the certificate set it needs is close to linear in the degrees of freedom of the moving system.
- **locality:** A KDS is *local* if no object participates in too many certificates; this condition makes it easier to re-estimate certificate failure times when an object changes its motion law. (The existence of local KDSs is an intriguing theoretical question for several geometric attribute functions.)

23.4.3 The Convex Hull, Revisited

We now briefly describe a KDS for maintaining the convex hull of n points moving around in the plane [16].

The key goal in designing a KDS is to produce a *repairable certification* of the geometric object we want to track. In the convex hull case it turns out that it is a bit more intuitive to look at the dual problem, that of maintaining the upper (and lower) envelope of a set of moving lines in the plane, instead of the convex hull of the primal points. Such dualities represent a powerful toolkit in computational geometry; readers are referred to any standard computational geometry textbook for details, for example [21]. For simplicity we focus only on the upper envelope of the moving lines from now on; the lower envelope case is entirely symmetric. Using a standard divide-and-conquer approach, we partition our lines into two groups of size roughly $n/2$ each, and assume that recursive invocations of the algorithm maintain the upper envelopes of these groups. For convenience call the groups red and blue.

In order to produce the upper envelope of all the lines, we have to merge the upper envelopes of the red and blue groups and also certify this merge, so we can detect when it ceases to be valid as the lines move; see Figure 23.4.

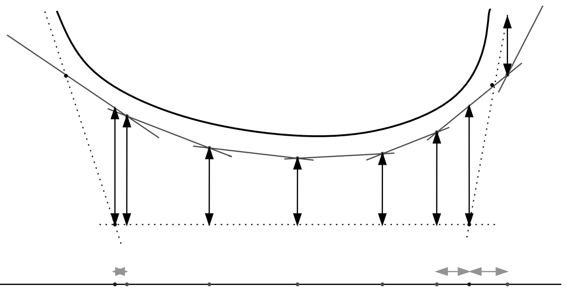


FIGURE 23.4: Merging the red and blue upper envelopes. In this example, the red envelope (solid line) is above the blue (dotted line), except at the extreme left and right areas. Vertical double-ended arrows represent y-certificates and horizontal double-ended arrows represent x-certificates, as described below.

Conceptually, we can approach this problem by sweeping the envelopes with a vertical line from left to right. We advance to the next red (blue) vertex and determine if it is above or below the corresponding blue (red) edge, and so on. In this process we determine when red is above blue or vice versa, as well as when the two envelopes cross. By stitching together all the upper pieces, whether red or blue, we get a representation of the upper envelope of all the lines.

The certificates used in certifying the above merge are of three flavors:

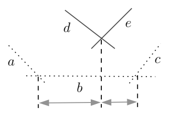
- x-certificates ($<_x$) are used to certify to x-ordering among the red and blue vertices; these involve four original lines.
- y-certificates ($<_y$) are used to certify that a vertex is above or below an edge of the opposite color; these involve three original lines and are exactly the duals of the CCW certificates discussed earlier.
- s-certificates ($<_s$) are slope comparisons between pairs of original lines; though these did not arise in our sweep description above, they are needed to make the KDS local [16].

Figure 23.5 shows examples of how these types of certificates can be used to specify x-ordering constraints and to establish intersection or non-intersection of the envelopes.

A total of $O(n)$ such certificates suffices to verify the correctness of the upper envelope merge.

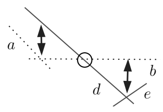
Whenever the motion of the lines causes one of these certificates to fail, a local, constant-time process suffices to update the envelope and repair the certification. Figure 23.6 shows an example where an y-certificate fails, allowing the blue envelope to poke up above the red.

It is straightforward to prove that this kinetic upper envelope algorithm is responsive, local, and compact, using the logarithmic depth of the hierarchical structure of the certification. In order to bound the number of events processed, however, we must assume that the line motions are polynomial or at least pseudo-algebraic. A proof of efficiency can be developed by extruding the moving lines into space-time surfaces. Using certain well-known theorems about the complexity of upper envelopes of surfaces [43] and the overlays of such



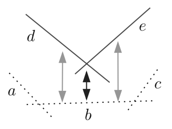
$ab <_x de$
$de <_x bc$

x-ordering certificates



$ab <_y d$
$de <_y b$

intersection certificates



$d <_s b$
$b <_s e$
$b <_y de$

non-intersection certificates

FIGURE 23.5: Using the different types of certificates to certify the red-blue envelope merge.

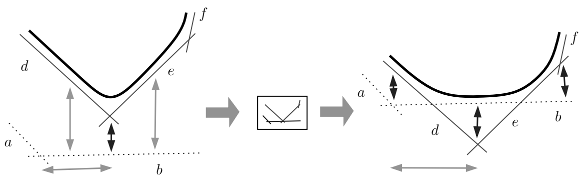


FIGURE 23.6: Envelope repair after a certificate failure. In the event shown lines b, d, and e become concurrent, producing a red-blue envelope intersection

envelopes [3] it can be shown that in the worst case the number of events processed by this algorithm is near quadratic ($O(n^{2+\epsilon})$). Since the convex hull of even linearly moving points can change $\Omega(n^2)$ times [8], the efficiency result follows.

No comparable structure is known for the convex hull of points in dimensions $d \geq 3$.

23.5 A KDS Application Survey

Even though we have presented kinetic data structures in a geometric setting, there is nothing intrinsically geometric about KDS. The idea of cached assertions that help track an attribute of interest can be applied to many other settings where there is continuous evolution over time punctuated by discrete events, beyond motion in the physical world. For example, consider a graph whose edge weights or capacities are functions of time, as might arise in an evolving communications network. Then the problem of tracking various substructures of interest, such as the minimum spanning tree (MST) of the graph, or a shortest path tree form a source node, can be formulated and studied within the KDS framework.

We present below a quick summary of some of the areas to which kinetic data structures have been applied so far. The are mostly geometric in nature, but several non-geometric examples appear as well.

23.5.1 Extent Problems

A number of the original problems for which kinetic data structures were developed are aimed at different measures of how “spread out” a moving set of points in $\mathcal{R}^2$ is—one example is the convex hull, whose maintenance was discussed in the previous subsection. Other measures of interest include the diameter, width, and smallest area or perimeter bounding rectangle for a moving set S of n points. All these problems can be solved using the kinetic convex hull algorithm above; the efficiency of the algorithms is $O(n^{2+\epsilon})$, for any $\epsilon > 0$. There are also corresponding $\Omega(n^2)$ lower bounds for the number of combinatorial changes in these measures. Surprisingly, the best known upper bound for maintaining the smallest enclosing disk containing S is still near-cubic. Extensions of these results to dimensions higher than two are also lacking.

These costs can be dramatically reduced if we consider approximate extent measures. If we are content with $(1 + \epsilon)$ approximations to the measures, then an approximate smallest orthogonal rectangle, diameter, and smallest enclosing disk can be maintained with a number of events that is a function ϵ only and not of n [9]. For example, the bound of the number of approximate diameter updates in $\mathcal{R}^2$ under linear motion of the points is $O(1/\epsilon)$.

23.5.2 Proximity Problems

The fundamental proximity structures in computational geometry are the Voronoi Diagram and the Delaunay triangulation (Chapters 62 and 63). The edges of the Delaunay triangulation contain the closest pair of points, the closest neighbor to each point, as well as a wealth of other proximity information among the points. From the kinetic point of view, these are nice structures, because they admit completely local certifications. Delaunay's 1934 theorem [22] states that if a local empty sphere condition is valid for each $(d-1)$-simplex in a triangulation of points in $\mathcal{R}^d$, then that triangulation must be Delaunay. This makes it simple to maintain a Delaunay triangulation under point motion: an update is necessary only when one of these empty sphere conditions fails. Furthermore, whenever that happens, a local retiling of space (of which the classic “edge-flip” in R^2 is a special case) easily restores Delaunayhood. Thus the KDS for Delaunay (and Voronoi) that follows from this theorem is both responsive and efficient—in fact, each KDS event is an external event in which the structure changes. Though no redundant events happen, an exact upper bound for the total number of such events in the worst-case is still elusive even in R^2, where the best known upper bound is nearly cubic, while the best lower bound only quadratic [12].

This principle of a set of easily checked local conditions that implies a global property has been used in kinetizing other proximity structures as well. For instance, in the *power diagram* [14] of a set of disjoint balls, the two closest balls must be neighbors [31]—and this diagram can be kinetized by a similar approach. Voronoi diagrams of more general objects, such as convex polytopes, have also been investigated. For example, in R^2 [29] shows how to maintain a compact Voronoi-like diagram among moving disjoint convex polygons; again, a set of local conditions is derived which implies the global correctness of this diagram. As the polygons move, the structure of this diagram allows one to know the nearest pair of polygons at all times.

In many applications the exact L_2-distance between objects is not needed and more relaxed notions of proximity suffice. Polyhedral metrics (such as L_1 or L_∞) are widely used, and the normal unit ball in L_2 can be approximated arbitrarily closely by polyhedral approximants. It is more surprising, however, that if we partition the space around each point into a set of polyhedral cones and maintain a number of directional nearest neighbors to each point in each cone, then we can still capture the globally closest pair of points in the L_2 metric. By directional neighbors here we mean that we measure distance only along a given direction in that cone. This geometric fact follows from a packing argument and is exploited in [17] to give a different method for maintaining the closest pair of points in $\mathcal{R}^d$. The advantage of this method is that the kinetic events are changes of the sorted order of the points along a set of directions fixed *a priori*, and therefore the total number of events is provably quadratic.

23.5.3 Triangulations and Tilings

Many areas in scientific computation and physical modeling require the maintenance of a triangulation (or more generally a simplicial complex) that approximates a manifold undergoing deformation. The problem of maintaining the Delaunay triangulation of moving points in the plane mentioned above is a special case. More generally, local re-triangulations are necessitated by collapsing triangles, and sometimes required in order to avoid undesirably "thin" triangles. In certain cases the number of nodes (points) may also have to change in order to stay sufficiently faithful to the underlying physical process; see, for example, [18]. Because in general a triangulation meeting certain criteria is not unique or canonical, it becomes more difficult to assess the efficiency of kinetic algorithms for solving such problems. The lower-bound results in [4] indicate that one cannot hope for a subquadratic bound on the number of events in the worst case in the maintenance an *any* triangulation, even if a linear number of additional Steiner points is allowed.

There is large gap between the desired quadratic upper bound and the current state of art. Even for maintaining an arbitrary triangulation of a set of n points moving linearly in the plane, the best-known algorithm processes $O(n^{7/3})$ events [5] in the worst case. The algorithm actually maintains a pseudotriangulation of the convex hull of the point set and then a triangulation of each pseudotriangle. Although there are only $O(n^2)$ events in the pseudotriangulation, some of the events change too many triangles because of high-degree vertices. Unless additional Steiner points are allowed, there are point configurations for which high-degree vertices are inevitable and therefore some of the events will be expensive. A more clever, global argument is needed to prove a near-quadratic upper bound on the total number of events in the above algorithm. Methods that choose to add additional points, on the other hand, have the burden of defining appropriate trajectories for these Steiner points as well. Finally, today no triangulation that guarantees certain quality on the shapes of triangles as well as a subcubic bound on the number of retiling events is known.

23.5.4 Collision Detection

Kinetic methods are naturally applicable to the problem of collision detection between moving geometric objects. Typically collisions occur at irregular intervals, so that fixed-time stepping methods have difficulty selecting an appropriate sampling rate to fit both the numerical requirements of the integrator as well as those of collision detection. A kinetic method based on the discrete events that are the failures of relevant geometric conditions can avoid the pitfalls of both oversampling and undersampling the system. For two moving convex polygons in the plane, a kinetic algorithm where the number of events is a function of the relative separation of the two polygons is given in [23]. The algorithm is based on constructing certain outer hierarchies on the two polygons. Analogous methods for 3D polytopes were presented in [30], together with implementation data.

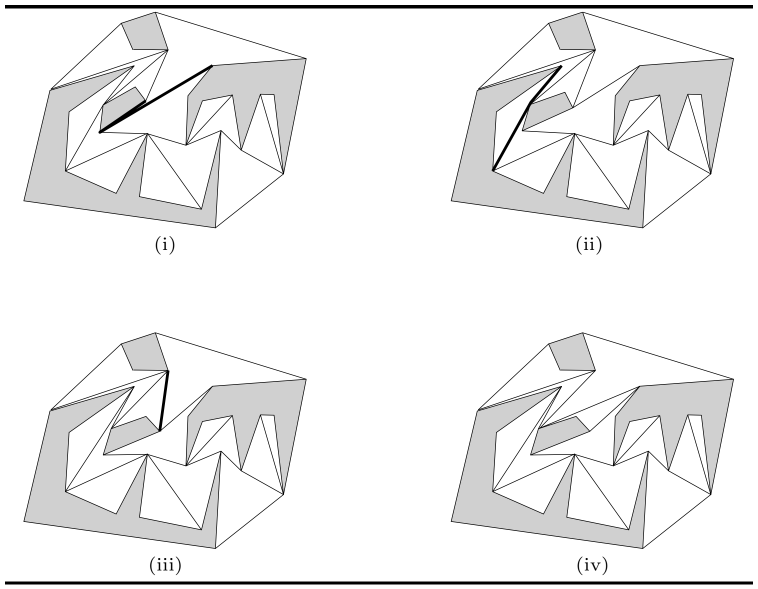


FIGURE 23.7: Snapshots of the mixed pseudotriangulation of [5]. As the center trapezoid-like polygon moves to the right, the edges corresponding to the next about-to-fail certificate are highlighted.

A tiling of the free space around objects can serve as a proof of non-intersection of the objects. If such a tiling can be efficiently maintained under object motion, then it can be the basis of a kinetic algorithm for collision detection. Several papers have developed techniques along these lines, including the case of two moving simple polygons in the plane [15], or multiple moving polygons [5, 38]. These developments all exploit deformable pseudotriangulations of the free space—tilings which undergo fewer combinatorial changes than, for example, triangulations. An example from [5] is shown in Figure 23.7. The figure shows how the pseudotriangulation adjusts by local retiling to the motion of the inner quadrilateral. The approach of [5] maintains a canonical pseudotriangulation, while others are based on letting a pseudotriangulation evolve according to the history of the motion. It is unclear at this point which is best. An advantage of all these methods is that the number of certificates needed is close to size of the min-link separating subdivision of the objects, and thus sensitive to how intertwined the objects are.

Deformable objects are more challenging to handle. Classical methods, such as bounding volume hierarchies [27], become expensive, as the fixed object hierarchies have to be rebuilt frequently. One possibility for mitigating this cost is to let the hierarchies themselves deform continuously, by having the bounding volumes defined implicitly in terms of object features. Such an approach was developed for flexible linear objects (such as rope or macromolecules), using combinatorially defined sphere hierarchies in [28]. In that work a bounding sphere is defined not in the usual way, via its center and radius, but in an implicit combinatorial way, in terms of four feature points of the enclosed object geometry. As the object deforms these implicitly defined spheres automatically track their assigned features, and therefore the deformation. Of course the validity of the hierarchy has to be checked at each time step and repaired if necessary. What helps here is that the implicitly defined spheres change their combinatorial description rather infrequently, even under extreme deformation. An example is shown in Figure 23.8 where the rod shown is bent substantially, yet only the top-level sphere needs to update its description.

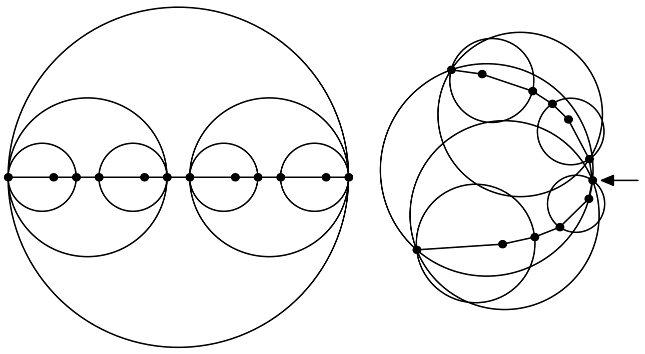


FIGURE 23.8: A thin rod bending from a straight configuration, and a portion of its associated bounding sphere hierarchy. The combinatorially defined sphere hierarchy is stable under deformation. Only the top level sphere differs between the two conformations.

The pseudotriangulation-based methods above can also be adapted to deal with object deformation.

23.5.5 Connectivity and Clustering

Closely related to proximity problems is the issue of maintaining structures encoding connectivity among moving geometric objects. Connectivity problems arise frequently in *ad hoc* mobile communication and sensor networks, where the viability of links may depend on proximity or direct line-of-sight visibility among the stations desiring to communicate. With some assumptions, the communication range of each station can be modeled by a geometric region, so that two stations can establish a link if and only if their respective regions overlap. There has been work on kinetically maintaining the connected components of the union of a set of moving geometric regions for the case of rectangles [36] and unit disks [33].

Clustering mobile nodes is an essential step in many algorithms for establishing communication hierarchies, or otherwise structuring *ad hoc* networks. Nodes in close proximity can communicate directly, using simpler protocols; correspondingly, well-separated clusters can reuse scarce resources, such the same frequency or time-division multiplexing communication scheme, without interference. Maintaining clusters of mobile nodes requires a tradeoff between the tightness, or optimality of the clustering, and its stability under motion. In [26] a randomized clustering scheme is discussed based on an iterated leader-election algorithm that produces a number of clusters within a constant factor of the optimum, and in which the number of cluster changes is also asymptotically optimal. This scheme was used in [25] to maintain a routing graph on mobile nodes that is always sparse and in which communication paths exist that are nearly as good as those in the full communication graph.

Another fundamental kinetic question is the maintenance of a minimum spanning tree (MST) among n mobile points in the plane, closely related to earlier work on parametric spanning trees [24] in a graph whose edge weights are functions of a parameter λ (λ is time in the kinetic setting). Since the MST is determined by the sorted order of the edge weights in the graph, a simple algorithm can be obtained by maintaining the sorted list of weights and some auxiliary data structures (such an algorithm is quadratic in the graph size, or $O(n^4)$ in our case). This was improved when the weights are linear functions of time to nearly $O(n^{11/6})$ (subquadratic) for planar graphs or other minor-closed families [6]. When the weights are the Euclidean distances between moving points, only approximation algorithms are known and the best event bounds are nearly cubic [17]. For many other optimization problems on geometric graphs, such as shortest paths for example, the corresponding kinetic questions are wide open.

23.5.6 Visibility

The problem of maintaining the visible parts of the environment when an observer is moving is one of the classic questions in computer graphics and has motivated significant developments, such as binary space partition trees, the hardware depth buffer, etc. The difficulty of the question increases significantly when the environment itself includes moving objects; whatever visibility structures accelerate occlusion culling for the moving observer, must now themselves be maintained under object motion.

Binary space partitions (BSP) are hierarchical partitions of space into convex tiles obtained by performing planar cuts (Chapter 20). Tiles are refined by further cuts until the interior of each tile is free of objects or contains geometry of limited complexity. Once a BSP tree is available, a correct visibility ordering for all geometry fragments in the tiles can be easily determined and incrementally maintained as the observer moves. A kinetic algorithm for visibility can be devised by maintaining a BSP tree as the objects move. The key insight is to certify the correctness of the BSP tree through certain combinatorial conditions, whose failure triggers localized tree rearrangements — most of the classical BSP construction algorithms do not have this property. In $\mathcal{R}^2$, a randomized algorithm for maintaining a BSP of moving disjoint line segments is given in [11]. The algorithm processes $O(n^2)$ events, the expected cost per tree update is $O(\log n)$, and the expected tree size is $O(n \log n)$. The maintenance cost increases to $O(n\lambda_{s+2}(n) \log^2 n)$ [7] for disjoint moving triangles in $\mathcal{R}^3$ (s is a constant depending on the triangle motion). Both of these algorithms are based on variants on vertical decompositions (many of the cuts are parallel to a given direction). It turns out that in practice these generate "sliver-like" BSP tiles that lead to robustness issues [19].

As the pioneering work on the visibility complex has shown [40], another structure that is well suited to visibility queries in $\mathcal{R}^2$ is an appropriate pseudotriangulation. Given a moving observer and convex moving obstacles, a full radial decomposition of the free space around the observer is quite expensive to maintain. One can build pseudotriangulations of the free space that become more and more like the radial decomposition as we get closer to the observer. Thus one can have a structure that compactly encodes the changing visibility polygon around the observer, while being quite stable in regions of the free space well-occluded from the observer [39].

23.5.7 Result Summary

We summarize in Table 23.9 the efficiency bounds on the main KDSs discussed above.

STRUCTURE	BOUNDS ON EVENTS	SOURCE
Convex hull	$\Omega(n^{2+\epsilon})$	[16]
Pseudotriangulation	$O(n^2)$	[5]
Triangulation (arb.)	$\Omega(n^{7/3})$	[5]
MST	$O(n^{11/6}\log^{3/2} n)$	[6]
BSP	$\tilde{O}(n^2)$	[7, 11]

FIGURE 23.9: Bounds on the number of combinatorial changes.

23.5.8 Open Problems

As mentioned above, we still lack efficient kinetic data structures for many fundamental geometric questions. Here is a short list of such open problems:

1. Find an efficient (and responsive, local, and compact) KDS for maintaining the convex hull of points moving in dimensions $d \geq 3$.
2. Find an efficient KDS for maintaining the smallest enclosing disk in $d \geq 2$. For $d = 2$, a goal would be an $O(n^{2+\epsilon})$ algorithm.
3. Establish tighter bounds on the number of Voronoi diagram events, narrowing the gap between quadratic and near-cubic.
4. Obtain a near-quadratic bound on the number of events maintaining an arbitrary triangulation of linearly moving points.*
5. Maintain a kinetic triangulation with a guarantee on the shape of the triangles, in subcubic time.
6. Find a KDS to maintain the MST of moving points under the Euclidean metric achieving subquadratic bounds.

Beyond specific problems, there are also several important structural issues that require further research in the KDS framework. These include:

Recovery after multiple certificate failures.
We have assumed up to now that the KDS assertion cache is repaired after each certificate failure. In many realistic scenarios, however, it is impossible to predict exactly when certificates will fail because explicit motion descriptions may not be available. In such settings we may need to sample the system and thus we must be prepared to deal with multiple (but hopefully few) certificate failures at each time step. A general area of research that this suggests is the study of how to efficiently update common geometric structures, such as convex hulls, Voronoi and Delaunay diagrams, arrangements, etc., after "small motions" of the defining geometric objects.

There is also a related subtlety in the way that a KDS assertion cache can certify the value, or a computation yielding the value, of the attribute of interest. Suppose our goal is to certify that a set of moving points in the plane, in a given circular order, always form a convex polygon. A plausible certificate set for convexity is that all interior angles of the polygon are convex. See Figure 23.10. In the normal KDS setting where we can always

*While this handbook was going to print, Agarwal, Wang and Yu, gave a near-quadratic such algorithm [44].

predict accurately the next certificate failure, it turns out that the above certificate set is sufficient, *as long as at the beginning of the motion the polygon was convex.* One can draw, however, nonconvex self-intersecting polygons all of whose interior angles are convex, as also shown in the same figure. The point here is that a standard KDS can offer a *historical* proof of the convexity of the polygon by relying on the fact that the certificate set is valid *and* that the polygon was convex during the prior history of the motion. Indeed the counterexample shown cannot arise under continuous motion without one of the angle certificates failing first. On the other hand, if an oracle can move the points when "we are not looking," we can wake up and find all the angle certificates to be valid, yet our polygon need not be convex. Thus in this oracle setting, since we cannot be sure that no certificates failed during the time step, we must insist on *absolute* proofs — certificate sets that in any state of the world fully validate the attribute computation or value.

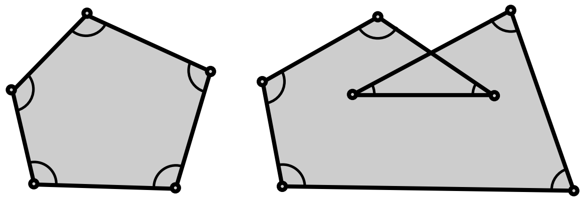


FIGURE 23.10: Certifying the convexity of a polygon.

Hierarchical motion descriptions.

Objects in the world are often organized into groups and hierarchies and the motions of objects in the same group are highly correlated. For example, though not all points in an elastic bouncing ball follow exactly the same rigid motion, the trajectories of nearby points are very similar and the overall motion is best described as the superposition of a global rigid motion with a small local deformation. Similarly, the motion of an articulated figure, such as a man walking, is most succinctly described as a set of relative motions, say that of the upper right arm relative to the torso, rather than by giving the trajectory of each body part in world coordinates.

What both of these examples suggest is that there can be economies in motion description, if the motion of objects in the environment can be described as a superposition of terms, some of which can be shared among several objects. Such hierarchical motion descriptions can simplify certificate evaluations, as certificates are often local assertions concerning nearby objects, and nearby objects tend to share motion components. For example, in a simple articulated figure, we may wish to assert $\textsc{ccw}(A, B, C)$ to indicate that an arm is not fully extended, where $\overline{AB}$ and $\overline{BC}$ are the upper and lower parts of the arm respectively. Evaluating this certificate is clearly better done in the local coordinate frame of the upper arm than in a world frame—the redundant motions of the legs and torso have already been factored out.

Motion sensitivity.

As already mentioned, the motions of objects in the world are often highly correlated and it behooves us to find representations and data structures that exploit such motion coherence. It is also important to find mathematical measures that capture the degree of coherence of a motion and then use this as a parameter to quantify the performance of motion algorithms. If we do not do this, our algorithm design may be aimed at unrealistic worst-case behavior, without capturing solutions that exploit the special structure of the motion data that actually arise in practice — as already discussed in a related setting in [20]. Thus it is important to develop a class of kinetic *motion-sensitive* algorithms, whose performance can be expressed a function of how coherent the motions of the underlying objects are.

Non-canonical structures.

The complexity measures for KDSs mentioned earlier are more suitable for maintaining *canonical* geometric structures, which are uniquely defined by the position of the data, e.g., convex hull, closest pair, and Delaunay triangulation. In these cases the notion of external events is well defined and is independent of the algorithm used to maintain the structure. On the other hand, as we already discussed, suppose we want to maintain a triangulation of a moving point set. Since the triangulation of a point set is not unique, the external events depend on the triangulation being maintained, and thus depend on the algorithm. This makes it difficult to analyze the efficiency of a kinetic triangulation algorithm. Most of the current approaches for maintaining noncanonical structures artificially impose canonicality and maintain the resulting canonical structure. But this typically increases the number of events. So it is entirely possible that methods in which the current form of the structure may depend on its past history can be more efficient. Unfortunately, we lack mathematical techniques for analyzing such history-dependent structures.

23.6 Querying Moving Objects

Continuous tracking of a geometric attribute may be more than is needed for some applications. There may be time intervals during which the value of the attribute is of no interest; in other scenarios we may be just interested to know the attribute value at certain discrete query times. For example, given n moving points in $\mathcal{R}^2$, we may want to pose queries asking for all points inside a rectangle R at time t, for various values of R and t, or for an interval of time Δt, etc. Such problems can be handled by a mixture of kinetic and static techniques, including standard range-searching tools such as partition trees and range trees [21]. They typically involve tradeoffs between evolving indices kinetically, or prebuilding indices for static snapshots. An especially interesting special case is when we want to be able answer queries about the near future faster than those about the distant future—a natural desideratum in many real-time applications.

A number of other classical range-searching structures, such as k-d-trees and R-trees have recently been investigated for moving objects [1, 2].

23.7 Sources and Related Materials

Results not given an explicit reference above may be traced in these surveys.

[32]: An early, and by now somewhat dated, survey of KDS work.

[10]: A report based on an NSF-ARO workshop, addressing several issues on modeling motion from the perspective of a variety of disciplines.

[34]: A "popular-science" type article containing material related to the costs of sensing and communication for tracking motion in the real world.

References

[1] P. Agarwal, J. Gao, and L. Guibas. Kinetic medians and *kd*-trees. In *Proc. 10-th Europ. Symp. on Algorithms (ESA)*, pages 5–16, 2002.

[2] P. Agarwal, S. Har-Peled, and M. Procopiuc. Star-tree: An efficent self-adjusting index for moving points. In *Workshop on Algorithms Engineering*, 2002.

[3] P. K. Agarwal, O. Schwarzkopf, and Micha Sharir. The overlay of lower envelopes and its applications. *Discrete Comput. Geom.*, 15:1–13, 1996.

[4] Pankaj K. Agarwal, J. Basch, Mark de Berg, L. J. Guibas, and J. Hershberger. Lower bounds for kinetic planar subdivisions. In *Proc. 15-th Annu. ACM Sympos. Comput. Geom. (SoCG)*, pages 247–254, 1999.

[5] Pankaj K. Agarwal, J. Basch, L. J. Guibas, J. Hershberger, and L. Zhang. Deformable free space tiling for kinetic collision detection. In *Fourth Workshop on Algorithmic Foundations of Robotics (WAFR)*, pages 83–96, 2000.

[6] Pankaj K. Agarwal, D. Eppstein, L. J. Guibas, and M. Henzinger. Parametric and kinetic minimum spanning trees. In *Proc. 39th Annu. IEEE Sympos. Found. Comput. Sci.*, pages 596–605, 1998.

[7] Pankaj K. Agarwal, Jeff Erickson, and Leonidas J. Guibas. Kinetic BSPs for intersecting segments and disjoint triangles. In *Proc. 9th ACM-SIAM Sympos. Discrete Algorithms*, pages 107–116, 1998.

[8] Pankaj K. Agarwal, Leonidas J. Guibas, John Hershberger, and Eric Veach:. Maintaining the extent of a moving point set. *Discrete and Computational Geometry*, 26(3):353–374, 2001.

[9] Pankaj K. Agarwal and Sariel Har-Peled. Maintaining approximate extent measures of moving points. In *Proc.12th ACM-SIAM Sympos. Discrete Algorithms (SODA)*, pages 148–157, 2001.

[10] P.K. Agarwal, L.J. Guibas, et al. Algorithmic issues in modeling motion. *ACM Computing Surveys*, 34(4):550–572, 2003.

[11] P.K. Agarwal, L.J. Guibas, T.M. Murali, and J.S. Vitter. Cylindrical static and kinetic binary space partitions. *Comp. Geometry, Theory and Appl.*, 16:103–127, 2000.

[12] G. Albers, Leonidas J. Guibas, Joseph S. B. Mitchell, and T. Roos. Voronoi diagrams of moving points. *Internat. J. Comput. Geom. Appl.*, 8:365–380, 1998.

[13] M. J. Atallah. Some dynamic computational geometry problems. *Comput. Math. Appl.*, 11(12):1171–1181, 1985.

[14] F. Aurenhammer. Power diagrams: properties, algorithms and applications. *SIAM J. Comput.*, 16:78–96, 1987.

[15] J. Basch, J. Erickson, L. J. Guibas, J. Hershberger, and L. Zhang. Kinetic collision detection between two simple polygons. In *Proc. 10-th ACM-SIAM Sympos. Discrete Algorithms (SODA)*, pages 327–336, 1999.

[16] J. Basch, L. J. Guibas, and J. Hershberger. Data structures for mobile data. *Journal of Algorithms*, 31:1–28, 1999.

[17] J. Basch, L. J. Guibas, and L. Zhang. Proximity problems on moving points. In *Proc. 13th Annu. ACM Sympos. Comput. Geom.*, pages 344–351, 1997.

[18] H.L. Cheng, Tamal K. Dey, Herbert Edelsbrunner, and John Sullivan. Dynamic skin triangulation. In *Proc. 12-th SIAM Symposium on Discrete Algorithms (SODA)*, pages 47–56, 2001.

[19] J.L.D. Comba. *Kinetic vertical decomposition trees.* PhD thesis, Stanford University, 1999.

[20] M. de Berg, M. J. Katz, A. F. van der Stappen, and J. Vleugels. Realistic input models for geometric algorithms. In *Proc. 13th Annu. ACM Sympos. Comput. Geom.*, pages

294–303, 1997.

[21] Mark de Berg, Marc van Kreveld, Mark Overmars, and Otfried Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, Berlin, Germany, 2nd edition, 2000.

[22] B. Delaunay. Sur la sphère vide. A la memoire de Georges Voronoi. *Izv. Akad. Nauk SSSR, Otdelenie Matematicheskih i Estestvennyh Nauk*, 7:793–800, 1934.

[23] J. Erickson, L. J. Guibas, J. Stolfi, and L. Zhang. Separation-sensitive collision detection for convex objects. In *Proc. 10-th ACM-SIAM Symp. Discrete Algorithms (SODA)*, pages 102–111, 1999.

[24] D. Fernàndez-Baca, G. Slutzki, and D. Eppstein. Using sparsification for parametric minimum spanning tree problems. *Nordic Journal of Computing*, 3:352–366, 1996.

[25] J. Gao, L. J. Guibas, J. Hershberger, L. Zhang, and A. Zhu. Geometric spanner for routing in mobile networks. In *Proc. 2n-d ACM Symp. on Ad-Hoc Networking and Computing MobiHoc)*, pages 45–55, Oct. 2001.

[26] J. Gao, L.J. Guibas, J. Hershberger, L. Zhang, and A. Zhu. Discrete mobile centers. In *Proc. 17-th ACM Symp. on Computational Geometry (SoCG)*, pages 190–198, Jun 2001.

[27] Stefan Gottschalk, Ming Lin, and Dinesh Manocha. OBB-Tree: A hierarchical structure for rapid interference detection. In *SIGGRAPH 96 Conference Proceedings*, pages 171–180, 1996.

[28] L. Guibas, A. Nguyen, D. Russell, and L. Zhang. Collision detection for deforming necklaces. In *Proc. 18-th Annu. ACM Sympos. Comput. Geom. (SoCG)*, pages 33–42, 2002.

[29] L. Guibas, J. Snoeyink, and L. Zhang. Compact Voronoi diagrams for moving convex polygons. In *Proc. Scand. Workshop on Alg. and Data Structures (SWAT)*, volume 1851 of *Lecture Notes Comput. Sci.*, pages 339–352. Springer-Verlag, 2000.

[30] L. Guibas, F. Xie, and L. Zhang. Kinetic collision detection: Algorithms and experiments. In *Proceedings of the International Conference on Robotics and Automation (ICRA)*, pages 2903–2910, 2001.

[31] L. Guibas and L. Zhang. Euclidean proximity and power diagrams. In *Proc. 10-th Canadian Conf. Computational Geometry*, pages 90–91, 1998.

[32] L. J. Guibas. Kinetic data structures — a state of the art report. In P. K. Agarwal, L. E. Kavraki, and M. Mason, editors, *Proc. Workshop Algorithmic Found. Robot.*, pages 191–209. A. K. Peters, Wellesley, MA, 1998.

[33] Leonidas Guibas, John Hershberger, Subash Suri, and Li Zhang. Kinetic connectivity for unit disks. In *Proc. 16th Annu. ACM Sympos. Comput. Geom.*, pages 331–340, 2000.

[34] Leonidas J. Guibas. Sensing, tracking, and reasoning with relations. *IEEE Signal Proc. Magazine*, pages 73–85, 2002.

[35] P. Gupta, R. Janardan, and M. Smid. Fast algorithms for collision and proximity problems involving moving geometric objects. *Comput. Geom. Theory Appl.*, 6:371–391, 1996.

[36] J. Hershberger and S. Suri. Kinetic connectivity of rectangles. In *Proc. 15-th Annu. ACM Sympos. Comput. Geom. (SoCG)*, pages 237–246, 1999.

[37] S. Kahan. A model for data in motion. In *Proc. 23th Annu. ACM Sympos. Theory Comput.*, pages 267–277, 1991.

[38] David Kirkpatrick, Jack Snoeyink, and Bettina Speckmann. Kinetic collision detection for simple polygons. In *Proc. 16th Annu. ACM Sympos. Comput. Geom.*, pages 322–330, 2000.

[39] O.Hall-Holt. *Kinetic visibility.* PhD thesis, Stanford University, 2002.

[40] M. Pocchiola and G. Vegter. The visibility complex. *Internat. J. Comput. Geom. Appl.*, 6(3):279–308, 1996.

[41] Elmar Schömer and Christian Thiel. Efficient collision detection for moving polyhedra. In *Proc. 11th Annu. ACM Sympos. Comput. Geom.*, pages 51–60, 1995.

[42] Elmar Schömer and Christian Thiel. Subquadratic algorithms for the general collision detection problem. In *Abstracts 12th European Workshop Comput. Geom.*, pages 95–101. Universität Münster, 1996.

[43] Micha Sharir. Almost tight upper bounds for lower envelopes in higher dimensions. *Discrete Comput. Geom.*, 12:327–345, 1994.

[44] P. Agarwal, Y. Wang, and H. Yu. A 2d kinetic triangulation with near quadratic topological changes. *Proc. 20th ACM Symp. on Computational Geometry (SoCG)*, 180–189, 2004.

24

Online Dictionary Structures

Teofilo F. Gonzalez
University of California, Santa Barbara

24.1 Introduction

Given an initially empty set S, the dictionary problem consists of executing on-line any sequence of operations of the form $S.membership(s)$, $S.insert(s)$ and $S.delete(s)$, where each element s is an object (or point in one dimensional space). Each object can be stored in a single word, and it takes $O(1)$ time to save and/or retrieve it. The set may be represented by using arrays (sorted or unsorted), linked lists (sorted or unsorted), hash tables, binary search trees, AVL-trees, B-trees, 2-3 trees, weighted balanced trees, or balanced binary search trees (i.e., 2-3-4 trees,symmetric B-trees, half balanced trees or red-black trees). The worst case time complexity for performing each of these operations is $O(\log n)$, where n is the maximum number of elements in a set, when using AVL-trees, B-trees (fixed order), 2-3 trees, weighted balanced trees, or balanced binary search trees. See Chapters 2, 3, 9, 10 and 15 for details on these structures. The insertion or deletion of elements in these structures requires a set of operations to preserve certain properties of the resulting trees. For binary search trees, these operations are called *rotations,* and for m-way search trees they are called *splitting or combining* nodes. The balanced binary search trees are the only trees that require a constant number of rotations for both the *insert* and *delete* operations [13, 15].

In this chapter we discuss several algorithms [4, 5] for the generalized dictionary problem when the data is multidimensional, rather than one dimensional. Each data element consists of d ordered components which we call *ordered d-tuple*, or simply d-tuple. Each component contains a value which can be stored in a memory word and which can be compared against another value to determine whether the values are identical, the first value is larger than the second one, or the first value is smaller than the second one. The comparison operation takes a constant amount of time. We show that the multidimensional dictionary operations can be implemented to take $O(d + \log n)$, where n is the number of d-tuples in the set. We also show that other common operations can also be executed within the same time complexity bounds.

Let us now discuss one of the applications of multidimensional dictionaries. We are given a set of n points in multidimensional space and an integer D, and the problem is to find the least number of orthogonal hypersquares (or d-boxes) of size D to cover all the points, i.e., each of the points must be in at least one of the d-boxes. This covering problem arises in image processing, and in locating emergency facilities so that all users are within a reasonable distance of one of the facilities [3]. This covering problem has been shown to be NP-hard and several polynomial time approximation schemes for its solution have been developed [3]. The simplest approximation algorithm defines d-boxes along a multidimensional grid with grid d-boxes of length D. The approximation algorithm takes every d-dimensional point and by using simple arithmetic operations, including the floor function, finds its appropriate (grid) d-box. The d-box, which is characterized by d integer components, is inserted into a multidimensional dictionary and the operation is repeated for each d-dimensional point. Then one just visits all the d-tuples in the set and the d-boxes they represent are part of the solution generated by this simple approximation algorithm. Note that when a d-tuple is inserted into a multidimensional dictionary that contains it, the d-tuple will not modify the dictionary because dictionaries store sets, i.e., multiple copies of the d-tuples are not allowed. Other application of multidimensional dictionaries are when accessing multi-attribute data by value. These applications include the management of geometrical objects and the solution of geometry search problems.

Given the d-tuple s in set S, one may access in constant time the i^{th} element in the d-tuple by using the function $s.x(i)$. In other words, the d-tuple s is simply $(s.x(1), s.x(2), \ldots, s.x(d))$ In this chapter we examine the methods given in [4, 5] to represent the data set and their algorithms to perform on-line any sequence of the multidimensional dictionary operations. The most efficient of the implementations performs each of the three dictionary operations in $O(d + \log n)$ time, where n is the number of d-tuples, and d is the number of dimensions. Each of the insert and delete operations requires no more than a constant number of rotations. The best of the algorithms requires dn words to represent the d-tuples, plus $O(n)$ additional space is required to keep additional pointers and data. Because we are using balanced binary search trees, we can also perform other operations efficiently. For example, find the (lexicographically) smallest or largest d-tuple ($O(\log n)$ time), print in lexicographic order ($O(dn)$ time), and concatenation ($O(d + \log n)$ time). By modifying slightly the representation and introducing additional information, one can also find the (lexicographically) k^{th} smallest or largest d-tuple in ($O(\log n)$ time). In Section 24.5 we show that the structure given in [5] may also be used to implement the split operation in $O(d + \log n)$ time, and that the approach can also be used in other balanced binary search trees, like AVL, weight balanced, etc.

24.2 Trie Implementations

It is interesting to note that two decades ago balanced structures were written off for this type of applications. As noted in [12], “balanced tree schemes based on key comparisons (e.g., AVL-trees, B-trees, etc.) lose some of their usefulness in this more general context”. At that time the approach was to use *tries* in conjunction with balanced tree schemes to represent multidimensional data.

Given a set of strings over an alphabet Σ, the tree of all their prefixes is called a *trie* (Chapter 28). In Figure 24.1 we depict the trie for the set of strings over the English alphabet $T = \{akam, aklm, cmgi, cmos, cors, corv, nort, novl, novn\}$. In this case all the elements in the trie have the same number of symbols, but in general a trie may contain string with different number of symbols. Each node x in a trie is the destination of all the strings with the same prefix (say p_x) and node x consists of a set of element-pointer pairs. The first component of the pair is an alphabet element and the second one is a pointer to a subtrie that contains all the strings with prefix p_x followed by the alphabet element. In Figure 24.1 the pairs for a trie node are represented by the edges emanating from the lower portion of the circle that represents the node. Note that no two element-pointer pairs for a trie node have the same first component or the same second component. The set of element-pointer pairs in a trie node may be represented in different ways. For example one may store the element-ponter pairs for each internal node as:

1. An array of m pointers, where m is the number of elements in the alphabet Σ. In this case one needs to define a function to translate each element in the alphabet to an integer in the range $[0, m-1]$. The function can normally be implemented to take constant time.

2. A sorted linked list with all the symbols and corresponding pointers of the branches emanating from the node (Sussenguth [14]).

3. A binary search tree with all the symbols and corresponding pointers of the branches emanating from the node. (Clampett [2])

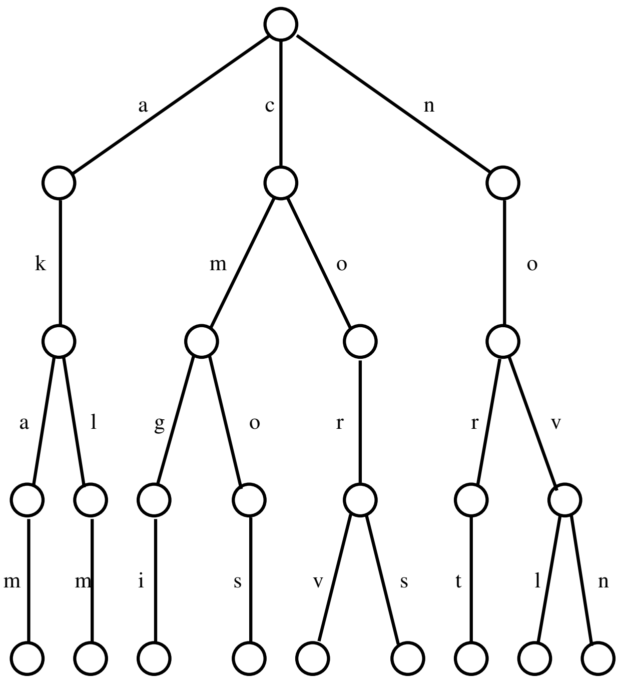


FIGURE 24.1: TRIE for set T

We shall refer to the resulting structures as trie-array, trie-list, and trie-bst, respectively.

For multidimensional dictionaries defined over the set of integers $[0, m)$, the trie method treats a point in d-space as a string with d elements defined over the alphabet $\Sigma = \{0, 1, \ldots, m-1\}$ (see Figure 24.1). For the trie-array representation the element-pointer pairs in a trie node are represented by an m-element vector of pointers. The i^{th} pointer corresponds to the i^{th} alphabet symbol, and a null pointer is used when the corresponding alphabet element is not part of any element-pointer pair. The space required to represent n d-tuples in this structure is about dnm pointers, but the total amount of information can be represented in $O(dn \log m)$ bits. The insert and delete operation takes $O(md)$ time, because either of these operations may add or delete $d-1$ trie nodes and each one has an m-component array of pointers. On the other hand, the membership operation takes only $O(d)$ time. This is the fastest possible implementation for the membership operation. The trie-array implementation is not possible when m is large because there will be too much space wasted. The trie-list representation mentioned above is much better for this scenario. In this case the list of element-pointer pairs is stored in a sorted linked list. The list is sorted with respect to the alphabet elements, i.e., the first component in the element-pointer pairs. In the trie-bst representation, the element-pointers are stored in a binary search tree. The ordering is with respect to the alphabet elements. In the former case, we use dn memory locations for the d-tuples, plus 2dn pointers, and in the latter case one needs $3dn$ pointers. The time complexity for insert, delete and membership in both of these representations is $O(d+n)$. It is important to note that there are two types of pointers, the trie pointers and the linked list or binary search tree pointers.

In practice one may use hybrid structures in which some nodes in the trie are trie-arrays and others are trie-lists, and depending on the number of element-pointer pairs one transforms from one representation to the other. For example, if the number of element-pointer pairs becomes more than some bound b_u one uses trie-array nodes, but if it is less then b_l then one uses the trie-list. If it is some number between these two bounds, then either representation is fine to use. By using appropriate values for b_l and b_u in this approach we can avoid "trashing" which means that one spends most of the time changing back and forth from one representations to the other.

Bentley and Saxe [1] used a modified version of the trie-bst implementation. In this case the binary search tree is replaced by a completely balanced binary tree. I.e., each binary search tree or subtree for trie node x with k element-pointer pairs has as root an element-pointer pair (y, ptr) such that the median of the ordered strings (with prefix equal to the prefix of the trie node x (denoted by p_x) plus any of the k alphabet elements in the k element-pointer pairs) has as prefix p_x followed by y. For example, the root of the trie in Figure 24.1 has the pair with alphabet element c, and the root of the subtrie for the prefix no is the pair with alphabet element v. This balanced structure is the best possible when the trie does not change or it changes very slowly like in multikey sorting or restricted searching [9, 10]. Since the overall structure is rigid, it can be shown that rebalancing after inserting or deleting an element can be very expensive and therefore not appropriate in a dynamic environment.

Research into using different balanced strategies for the trie-bst structures has lead into structures that use fixed order B-trees [7], weight balanced trees [12], AVL trees [16], and balanced binary search trees [17]. It has been shown that insert, delete and membership for multidimensional dictionaries for all of the above structures takes $O(d + \log n)$ time in the worst case, except for weight balanced trees in which case it is an expected time complexity bound. The above procedures are complex and require balancing criteria in addition to the obvious ones. Also, the number of rotations needed for these insert and delete operations is not bounded by a constant. Vaishnavi [17] used balanced binary search trees that require only a constant number of rotations, however since one may encounter d trees, the total number of rotations are no longer bounded by a constant. He left it as an open problem to design a structure such that multidimensional dictionaries can be implemented so that only a constant number of rotations are performed after each insert and delete operation.

24.3 Binary Search Tree Implementations

As pointed out by Gonzalez [4, 5], using a balanced binary search tree (without a trie) and storing each tuple at each node leads to membership (and also insert and delete) algorithms that take $O(d \log n)$ time, where n is the number of elements in the tree, because one needs to compare the element being searched with the d-tuples at each node. One can go further and claim that for some problem instances it actually requires $\Theta(d \log n)$ time. Gonzalez [5] also points out that simple shortcuts to the search process do not work. For example if we reach a node in the search such that the first k components are identical, one may be tempted to conclude that in the subtree rooted at that node one needs to search only from position $k+1$ to d in the d-tuples. This is false because the k-component prefixes of all the d-tuples in a subtree may vary considerable in a binary search tree. One can easily show that even the membership operation cannot be implemented this way. However, this variation is more predictable when comparing against the smallest or largest d-tuple in a subtree. This is a key idea exploited in [4].

Manber and Myers [11] developed an efficient algorithm that given an N symbol text it finds all the occurrences of any input word q. The scenario is that the text is static, but there will be many word searches. Their approach is to preprocess the text and generate a structure where the searching can be performed efficiently. In their preprocessing stage they construct a sorted list with all the N suffixes of the text. Locating all the occurrences of a string q reduces to performing two binary search operations in the list of suffixes, the first for the first suffix that contains as prefix q and the second search is for the last suffix that contains as prefix q. Both searches are similar, so lets discuss the first one. This operation is similar to the membership operation discussed in this chapter. Manber and Myers [11] binary search process begins by letting L (R) be the largest (smallest) string in the in the list. Then l (r), the index of the first element where L (R) differ from q is computed. Note that if two strings are identical the index of the first component where they differ is set to the length of string plus 1. We use this convention throughout this chapter. The middle entry M in the list is located and then they compute the index of the first component where M and q differ. If this value is computed the obvious way, then the procedure will not be efficient. To do this efficiently they compute it with l, r as well as index of the first component where M and L, and M and R differ. These last two values are precomputed in the preprocessing stage. This indirect computation may take $O(|q|)$ time; however, overall the phases of the computation the process takes at most $O(|q|)$ time. The advantage of this approach is that it requires only $O(N)$ space, and the preprocessing can be done in $O(N)$ expected time [11]. The disadvantage is that it is not a dynamic. Updating the text requires expensive recomputations in the precomputed data, i.e., one needs to find the first component where many pairs in the list differ in order to carry out efficiently the binary search process. For their application [11] the text is static. The time required to do the search for q is $O(|q| + \log N)$ time. This approach results in a structure that is similar to the fully balanced tree strategy in [1].

Gonzalez [4] solved Vaishnavi's open problem by designing a binary tree data structure where the multidimensional dictionary operations can be performed in $O(d + \log n)$ time while performing only a constant number of rotations. To achieve these goals he represents the set of d-tuples S in a balanced binary search tree that contains additional information at each node. This additional information is similar to the one in the lists of Manber and Myers [11], but it can be recomputed efficiently as we insert and delete elements. The disadvantage is that the membership operation is more complex. But all the procedures developed in [4] are simpler than the ones given in [7, 12, 16, 17]. One just needs to add a few instructions in addition to the normal code required to manipulate balanced binary search trees [15]. The additional information in [4] includes for every node v in the tree the index of the first element where v and the smallest (largest) d-tuple in the subtree rooted at v differ as well as a pointer to this d-tuple. As mentioned above, testing for membership is more complex. At each iteration in the search process [4] we are at the tree node t and we know that if q is in the tree then it is in the subtree rooted at t. The algorithm knows either the index of a component where q and the smallest d-tuple in the subtree t differ, or the index of a component where q and the largest d-tuple in the subtree t differ. Then the algorithm determines that q is in node t, or it advances to the left or right subtrees of node t. In either case it maintains the above invariant. It is important to point out the invariant is: "the index of a component where q and the smallest d-tuple in the subtree differ" rather than the "the first index of a ...". It does not seem possible to find "the first index ..." in this structure efficiently with the information stored in the tree. This is not a big problem when q is in the tree since it will be found quickly, but if it is not in the tree then in order to avoid reporting that it is in the tree one must perform an additional verification step at the end that takes $O(d)$ time. Gonzalez [4] calls this search strategy "assume, verify and conquer" (AVC). I.e., in order to avoid multiple expensive verification steps one assumes that some prefixes of strings match. The outcome of the search depends on whether or not these assumptions were valid. This can be determined by performing one simple verification step that takes $O(d)$ time. The elimination of multiple verifications is very important because in the worst case there are $\Omega(\log n)$ verifications, and each one could take $\Omega(d)$ time. The difference between this approach and the one in [11] is that Manber and Myers compute the first element where M and q differ, where as Gonzalez [4] computes an element where M and q differ. As we said before, in Gonzalez [4] structure one cannot compute efficiently the first element where M and q differ.

Gonzalez [5] modified the structure in [4] to one that follows the search process in [11]. The new structure, which we discuss in Section 24.4, is in general faster to update because for every node t one keeps the index of the first component where the d-tuple stored at node t and the smallest (largest) d-tuple greater (smaller) than all the d-tuples in the subtree rooted at t (if any) differ, rather than the one between node t and the smallest (largest) d-tuple in its subtree rooted at t as in [4]. In this structure only several nodes have to be modified when inserting a node or deleting a leaf node, but in the structure in [4] one may need to update $O(\log n)$ nodes. Deleting a non-leaf node from the tree requires more work in this structure than in [4], but membership testing is simpler in the structure in [5]. To summarize, the membership algorithm in [5] mimics the search procedure in [11], but follows the update approach developed in [4]. Gonzalez [5] established that the dictionary operations can be implemented to take $O(d + \log n)$ time while performing only a constant number of rotations for both insert and delete. Other operations which can be performed efficiently in this multidimensional balanced binary search trees are: find the (lexicographically) smallest (largest) d-tuple ($O(\log n)$ time), print in lexicographic order ($O(dn)$ time), and concatenation ($O(d + \log n)$ time). Finding the (lexicographically) k^{th} smallest (largest) d-tuple can also be implemented efficiently ($O(\log n)$ time) by adding to each node the number of nodes in its left subtree. The asymptotic time complexity bound for the procedures in [5] is identical to the ones in [4], but the procedures in [5] are simpler. To distinguish this new type of balanced binary search trees from the classic ones and the ones in [4], we refer to these trees as *multidimensional balanced binary search trees.* In this article we follow the notation in [5].

In Section 24.5 we show that the rotation operation in [5] can be implemented to take only constant time by making some rather simple operations that were first discussed in [6]. The implication of this, as pointed out in [6], is that the split operation can also be implemented to take $O(d + log\ n)$. Also, the efficient implementation of the rotation operation allows us to use the technique in [5] on many other binary search trees, like AVL, weight balanced, etc., since performing $O(\log n)$ rotations does not limit the applicability of the techniques in [5]. These observations were first reported in [6], where they present a similar approach, but in a more general setting.

24.4 Balanced BST Implementation

Let us now discuss the data structure and algorithms for the multidimensional dictionaries given in [5]. The representation is based on balanced binary search trees, without external nodes, i.e., each node represents one d-tuple. Balanced binary search trees and their algorithms ([15]) are like the typical "bottom-up" algorithms for the Red-Black trees. The ordering of the d-tuples is lexicographic. Each node t in the tree rooted at r has the following information in addition to the color bit required to manipulate balanced binary search trees [15]. Note that if two d-tuples are identical the index of the first component where they differ is set to $d + 1$. We use this convention through out this chapter.

s:	The d-tuple represented by the node. The individual elements may be accessed $s.x(1)$, $s.x(2)$, ..., $s.x(d)$.
lchild:	Pointer to the left subtree of t.
rchild:	Pointer to the right subtree of t.
lptr:	Pointer to the node with largest d-tuple in r with value smaller than all the d-tuples in the subtree rooted at t, or null if no such d-tuple exists.
hptr:	Pointer to the node with smallest d-tuple in r with value larger than all the d-tuples in the subtree rooted at t, or null if no such d-tuple exists.
lj:	Index of first component where s and the d-tuple at the node pointed at by *lptr* differ, or one if *lptr* = *null*.
hj:	Index of first component where s and the d-tuple at the node pointed at by *lptr* differ, or one if *hptr* = *null*.

The insert, delete and membership procedures perform the operations required to manipulate balanced binary search trees, and some new operations to update the structure. The basic operations to manipulate balanced binary search trees are well-known [13, 15]; so we only discuss in detail the new operations.

To show that membership, insert, and delete can be implemented to take $O(d + \log n)$ time, we only need to show that the following (new) operations can be performed $O(d+\log n)$ time.

A. Given the d-tuple q determine whether or not it is stored in the tree.

B. Update the structure after adding a node (just before rotation(s), if any).

C. Update the structure after performing a rotation.

D. Update the structure after deleting a leaf node (just before rotation(s), if any).

E. Transform the deletion problem to deletion of a leaf node.

The membership operation that tests whether or not the d-tuple q given by ($q.x(1)$, $q.x(2), \ldots,$ $q.x(d)$) is in the multidimensional binary search tree (or subtree) rooted at r appears in [5] and implements (A). The basic steps are as follows. Let t be any node encountered in the search process in the multidimensional balanced binary search tree rooted at r. Let $prev(t)$ to be the d-tuple in r with largest value but whose value is smaller than all the d-tuples stored in the subtree rooted at t, unless no such tuple exists in which case its value is $(-\infty, -\infty, \ldots, -\infty)$, and let $next(t)$ be the d-tuple in r with smallest value but whose value is larger than all the d-tuples stored in the subtree rooted by t, unless no such tuple exists in which case its value is $(+\infty, +\infty, \ldots, +\infty)$. The following invariant is maintained throughout the search process. During the search we will be visiting node t which is initially the root of the tree. The value of d_{low} is the index of the first component where q and $prev(t)$ differ, and variable d_{high} is the index of the first component where q and $next(t)$ differ. The d-tuple being search for, q, has value (lexicographically) greater than $prev(t)$ and (lexicographically) smaller than $next(t)$

The computation of j, the index of the first component where $t.s$ and q differ is like the one in [11]. When $d_{low} \geq d_{high}$, then either (1) $d_{low} \neq t.lj$ in which case j is just **min** $\{d_{low}, t.lj\}$, or (2) $d_{low} = t.lj$ in which case j is set to the index of the first component starting at position d_{low} where q and $t.s$ differ. The case when $d_{low} < d_{high}$ is similar and appears in [5]. Gonzalez [5] proved that by setting j by the above procedure will set it to the index of the first component where $t.s$ and q differ. When j is equal to $d + 1$, then q is the d-tuple stored in t and we return the value of **true**. Otherwise by comparing the jth element of q and t the procedure decides whether to search in the left or right subtrees of t. In either case d_{high} or d_{low} is set appropriately and the invariant holds at the next iteration. The time complexity at each level is not bounded by a constant; however, it is bounded by 1 plus the difference between the new and old value of $max\{d_{low}, d_{high}\}$. Since $max\{d_{low}, d_{high}\}$ does not decrease and it is at most $d + 1$ at the end of each operation, it follows that the total number of operations performed is of order d plus the height of the tree ($O(\log n)$). Correctness and efficiency are established in the following lemma whose proof appears in [5].

LEMMA 24.1 [5] Given a d-tuple q procedure membership(q, r) given in [5] determines whether or not q is in the multidimensional balanced binary search tree rooted at r in $O(d + \log n)$ time.

For operation (B), a node is added as a leaf node, the information that needs to be set for that node are the pointers *lptr* and *hptr* which can be copied from the parent node, unless there is no parent in which case they are set to null. The values lj, and hj can be computed directly in $O(d)$ time. A predecessor of the node added also needs its *lptr* and lj, or *hptr* and hj values changed. This can be easily done in $O(d)$, by remembering the the node in question during the search process. i.e., the last node where one makes a left turn (moving to the leftchild) or the last one where one makes a right turn (moving to the rightchild).

LEMMA 24.2 [5] After inserting a node q in a multidimensional balanced binary search tree and just before rotation the structure can be updated as mentioned above in $O(d+\log n)$ time.

The implementation of operation (D) is similar to the one for (B), therefore it will not be discussed further. The rotation operation (C) can be reduced to the simple rotation, because double and triple rotations can reduced to two and three simple rotations, respectively. A simple rotation is shown in Figure 24.2. It is simpler to implement the rotation operation by moving the nodes rather than just their values, because this reduces the number of updates that need to be performed. Clearly, the only nodes whose information needs to be updated are b, d and the parent of d. This just takes $O(d)$ time.

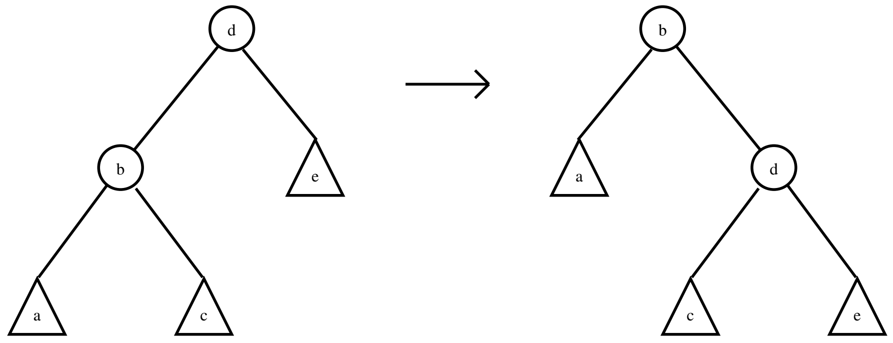


FIGURE 24.2: Rotation.

LEMMA 24.3 [5] After a rotation in a multidimensional balanced binary search tree the structure can be updated as mentioned above in $O(d)$ time.

Operation (E) is more complex to implement. It is well known [8] that the problem of deleting an arbitrary node from a balanced binary search tree can be reduced to deleting a leaf node by applying a simple transformation. Since all cases are similar, lets just discuss the case shown in Figure 24.3. The node to delete is a which is not a leaf node. In this case node b contents are moved to node a, and now we delete the old node b which is labeled x. As pointed out in [5], updating the resulting structure takes $O(d + \log n)$ time. Since the node labeled x will be deleted, we do not need to update it. For the new root (the one labeled b) we need to update the lj and hj values. Since we can use directly the old *lptr* and *hptr* pointers, the update can be done in $O(d)$ time. The lj (hj) value of the nodes (if any) in the path that starts at the right (left) child of the new root that continues through the leftchild (rightchild) pointers until the null pointer is reached needs to be updated. There are at most $O(\log n)$ of such nodes. However, the d-tuples stored at each of these nodes are decreasing (increasing) in lexicographic order when traversing the path top down. Therefore, the lj (hj) values appear in increasing (decreasing) order. The correct values can be easily computed in $O(d + \log n)$ time by reusing previously computed lj (hj) values while traversing the path top down. The following lemma, whose proof is omitted, summarizes these observations. Then we state the main result in [5] which is based on the above discussions and the lemmas.

LEMMA 24.4 [5] Transforming the deletion problem to deleting a leaf node can be performed, as mentioned above, in $O(d + \log n)$ time.

THEOREM 24.1 *[5] Any on-line sequence of operations of the form insert, delete and membership, for any d-tuple can be carried out by the procedure in [5] on a multidimensional balanced binary search tree in $O(d + \log n)$ time, where n is the current number of points, and each insert and delete operation requires no more than a constant number of rotations.*

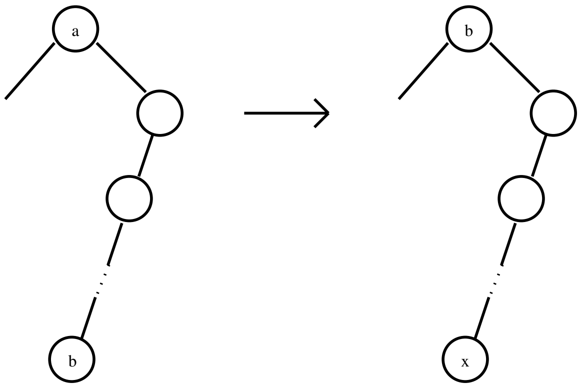


FIGURE 24.3: Transforming deletion of an arbitrary node to deletion of a leaf node.

24.5 Additional Operations

With respect to other operations, one can easily find the smallest or largest d-tuple in $O(\log n)$ time by just taking all the leftchild or rightchild pointers. By traversing the tree in inorder one can print all the d-tuples in increasing in $O(dn)$ time. An $O(d + \log n)$ time algorithm to concatenate two sets represented by the structure can be easily obtained by using standard procedures provided that all the d-tuples in one set are in lexicographic order smaller than the ones in the other set. The k^{th} smallest or largest d-tuple can be found in $O(\log n)$ time after adding to each node in the tree the number of nodes in its left subtree.

The split operation is given a d-tuple q and a set represented by a multidimensional balanced binary search tree t, split t into two multidimensional balanced binary search trees, one containing all the d-tuples in lexicographic order smaller than or equal to q, and the other one containing the remaining elements. At first glance, it seems that the split operation cannot be implemented within the $O(d+\log n)$ time complexity bound. The main reason is that there could be $\Omega(\log n)$ rotations and each rotation takes time proportional to d. However, the analysis in [5] for the rotation operation which is shown in Section 24.4 can be improved and one can show that it can be easily implemented to take constant time. The reason for this is that for a simple rotation, see Figure 24.2, the lj or the hj value need to be updated for the node labeled b, and we know the $lptr$ and $hptr$ pointers. This value can be computed from previous values before the rotation, i.e., the lj and hj values for node b and the fact that the $rptr$ value for b is node d before the rotation. The other value to be updated is the lj value for node d after the rotation, but this is simply the rj value for node d before the rotation.

The efficient implementation of the rotation operation allows us to use the technique in [5] on many other binary search trees, like AVL, weight balanced, etc., since having $O(\log n)$ rotations does not limit the applicability of the techniques in [5]. These observations were first made in [6] where they present a similar approach, but in a more general setting.

24.6 Discussion

On average the trie-bst approach requires less space to represent the d-tuples than our structures. However; multidimensional balanced binary search trees have simple procedures take only $O(d+\log n)$ time, and only a constant number of rotations are required after each insert and delete operations. Furthermore, operations like find the (lexicographically) smallest or largest d-tuple ($O(\log n)$ time), print in lexicographic order ($O(dn)$ time), concatenation ($O(d+\log n)$ time), and split ($O(d+\log n)$ time) can also be performed efficiently in this new structure. This approach can also be used in other balanced binary search trees, like AVL, weight balanced, etc.

References

[1] J. L. Bentley, and J. B. Saxe, Algorithms on Vector Sets, *SIGACT News*, (Fall 1979), pp. 36–39.

[2] H. A. Clampett, Randomized Binary Searching With the Tree Structures, *Comm. ACM*, 7, No. 3, (1964), pp. 163–165.

[3] T. Gonzalez, Covering a Set of Points with Fixed Size Hypersquares and Related Problems, *Information Processing Letters*, 40, (1991), pp. 181–188.

[4] T. Gonzalez, The On–Line D–Dimensional Dictionary Problem, Proceedings of the 3rd Symposium on Discrete Algorithms, January 1992, pp. 376 – 385.

[5] T. Gonzalez, Simple Algorithms for the On-Line Multidimensional Problem and Related Problems, *Algorithmica,* Vol. 28, (2000), pp. 255 – 267.

[6] R. Grossi and G. F. Italiano, International Colloquium on Automata, Languages and Programming (ICALP 99), Lecture Notes in Computer Science, Vol. 1644, (1999), pp 372 – 381.

[7] R. H. Gueting, and H. P. Kriegel, Multidimensional B-tree: An Efficient Dynamic File Structure for Exact Match Queries, Proceedings 10th GI Annual Conference, Informatik Fachberichte, Springer-Verlag, (1980), pp. 375–388.

[8] L. J. Guibas and R. Sedgewick, A Dichromatic Framework for Balanced Trees, Proceedings of the 19th Annual IEEE Symposium on Foundations of Computer Science, (1978), pp. 8–21.

[9] D. S. Hirschberg, On the Complexity of Searching a Set of Vectors, *SIAM J. on Computing* Vol. 9, No. 1, February (1980), pp. 126–129.

[10] S. R. Kosaraju, On a Multidimensional Search Problem, 1979 ACM Symposium on the Theory of Computing, pp. 67–73.

[11] U. Manber, and G. Myers, Suffix Arrays: A New Method for On–Line String Searches, *SIAM J. Comput,* Vol. 22, No. 5, Oct. 1993, pp. 935 – 948. Also in Proceedings of the First ACM-SIAM Symposium on Discrete Algorithms, Jan. 1990, pp. 319 – 327.

[12] K. Mehlhorn, Dynamic Binary Search, *SIAM J. Computing*, Vol. 8, No. 2, (May 1979), pp. 175–198.

[13] H. J. Olivie, A New Class of Balanced Search Trees: Half-Balanced Binary Search Trees, Ph.D. Thesis, University of Antwerp, U.I.A., Wilrijk, Belgium, (1980).

[14] E. H. Sussenguth, Use of Tree Structures for Processing Files, *Comm. ACM*, 6, No. 5, (1963), pp. 272–279.

[15] R. E. Tarjan, Updating a Balanced Search Tree in O(1) Rotations, *Information Processing Letters*, 16, (1983), pp. 253–257.

[16] V. Vaishnavi, Multidimensional Height-Balanced Trees, *IEEE Transactions on Computers*, Vol. c-33, No. 4, (April 1984), pp. 334–343.

[17] V. Vaishnavi, Multidimensional Balanced Binary Trees, *IEEE Transactions on Computers*, Vol. 38, No. 7, (April 1989), pp. 968–985.

25 Cuttings

Bernard Chazelle
Princeton University

25.1 Introduction

For divide-and-conquer purposes, it is often desirable to organize a set S of n numbers into a sorted list, or perhaps to partition it into two equal-sized groups with no element in one group exceeding any element in the other one. More generally, we might wish to break up S into k groups of size roughly n/k, with again a total ordering among the distinct groups. In the first case we sort; in the second one we compute the median; in the third one we compute quantiles. This is all well known and classical. Is it possible to generalize these ideas to higher dimension? Surprisingly the answer is yes. A geometric construction, known as an *ε-cutting*, provides a space partitioning technique that extends the classical notion of selection to any finite dimension. It is a powerful, versatile data structure with countless applications in computational geometry.

Let H be a set n hyperplanes in $\mathbf{R}^d$. Our goal is to divide up $\mathbf{R}^d$ into simplices, none of which is cut by too many of the n hyperplanes. By necessity, of course, some of the simplices need to be unbounded. We choose a parameter $\varepsilon > 0$ to specify the coarseness of the subdivision. A set $\mathcal{C}$ of closed full-dimensional simplices is called an ε-cutting for H (Fig. 25.1) if:

(i) the union of the simplices is $\mathbf{R}^d$, and their interiors are mutually disjoint;
(ii) the interior of any simplex is intersected by at most εn hyperplanes of H.

Historically, the idea of using sparsely intersected simplices for divide and conquer goes back to Clarkson [10] and Haussler and Welzl [15], among others. The definition of an ε-cutting given above is essentially due to Matoušek [18]. Efficient but suboptimal constructions were given by Agarwal [1, 2] for the two-dimensional case and Matoušek [17, 18, 21] for arbitrary dimension. The optimal ε-cutting construction cited in the theorem below, due to Chazelle [4], is a simplification of an earlier design by Chazelle and Friedman [7].

THEOREM 25.1 *Given a set H of n hyperplanes in $\mathbf{R}^d$, for any $0 < \varepsilon < 1$, there exists an ε-cutting for H of size $O(\varepsilon^{-d})$, which is optimal. The cutting, together with the list*

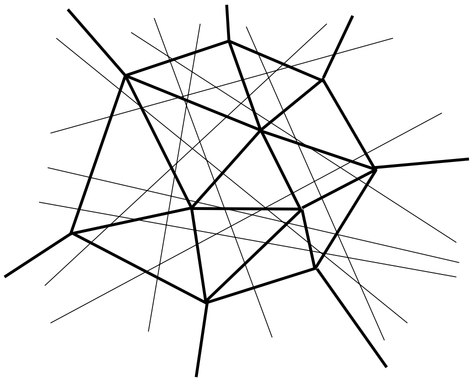


FIGURE 25.1: A two-dimensional cutting.

of hyperplanes intersecting the interior of each simplex, can be found deterministically in $O(n\varepsilon^{1-d})$ *time.*

25.2 The Cutting Construction

This section explains the main ideas behind the proof of Theorem 25.1. We begin with a quick overview of geometric sampling theory. For a comprehensive treatment of the subject, see [6, 23].

25.2.1 Geometric Sampling

A *set system* is a pair $\Sigma = (X, \mathcal{R})$, where X is a set and $\mathcal{R}$ is a collection of subsets of X. In our applications, $X \subset \mathbf{R}^d$ and each $R \in \mathcal{R}$ is of the form $X \cap f(K)$, where K is a fixed region of $\mathbf{R}^d$ and f is any member of a fixed group F of transformations. For example, we might consider n points in the plane, together with the subsets lying inside any triangle congruent to a fixed triangle.

Given $Y \subseteq X$, we define the set system "induced by Y" to be $(Y, \mathcal{R}|_Y)$, with $\mathcal{R}|_Y = \{\, Y \cap R \mid R \in \mathcal{R} \,\}$. The *VC-dimension* (named for Vapnik and Chervonenkis [28]) of Σ is defined as the maximum size of any Y such that $\mathcal{R}|_Y = 2^Y$. For example, the VC-dimension of the infinite geometric set system formed by points in the plane and halfplanes is 3. The *shatter function* $\pi_{\mathcal{R}}(m)$ of the set system $\Sigma = (X, \mathcal{R})$ is the maximum number of subsets in the set system $(Y, \mathcal{R}|_Y)$ induced by any $Y \subseteq X$ of size m. If $\pi_{\mathcal{R}}(m)$ is bounded by cm^d, for some constants $c, d > 0$, then the set system is said to have a *shatter function exponent* of at most d. It was shown in [26–28] that, if the shatter function exponent is $O(1)$, then so is the VC-dimension. Conversely, if the VC-dimension is $d \geq 1$ then, for any $m \geq d$, $\pi_{\mathcal{R}(m)} < (em/d)^d$.

We now introduce two fundamental notions: ε-nets and ε-approximations. For any $0 < \varepsilon < 1$, a set $N \subseteq X$ is called an *ε-net* for a finite set system $(X, \mathcal{R})$ if $N \cap R \neq \emptyset$ for any $R \in \mathcal{R}$ with $|R|/|X| > \varepsilon$. A finer (but more costly) sampling mechanism is provided by an

ε-*approximation* for $(X, \mathcal{R})$, which is a set $A \subseteq X$ such that, given any $R \in \mathcal{R}$,

$$\left| \frac{|R|}{|X|} - \frac{|A \cap R|}{|A|} \right| \leq \varepsilon.$$

Some simple structural facts about nets and approximations:

LEMMA 25.1 If X_1, X_2 are disjoint subsets of X of the same size, and A_1, A_2 are same-size ε-approximations for the subsystems induced by X_1, X_2 (respectively), then $A_1 \cup A_2$ is an ε-approximation for the subsystem induced by $X_1 \cup X_2$.

LEMMA 25.2 If A is an ε-approximation for $(X, \mathcal{R})$, then any ε'-approximation (resp. -net) for $(A, \mathcal{R}|_A)$ is also an $(\varepsilon + \varepsilon')$-approximation (resp. -net) for $(X, \mathcal{R})$.

In the absence of any restrictive assumption on the set system, it is natural to expect the sample size to depend on both the desired accuracy and the size of the set system itself.

THEOREM 25.2 *Given a set system $(X, \mathcal{R})$, where $|X| = n$ and $|\mathcal{R}| = m$, for any $1/n \leq \varepsilon < 1$, it is possible to find, in time $O(nm)$, an ε-net for $(X, \mathcal{R})$ of size $O(\varepsilon^{-1} \log m)$ and an ε-approximation for $(X, \mathcal{R})$ of size $O(\varepsilon^{-2} \log m)$.*

If we assume bounded VC-dimension, everything changes. In fact the key result in geometric sampling theory is that, for any given level of accuracy, the sample size need not depend on the size of the set system.

In practice, geometric set systems often are “accessible” via an *oracle* function that takes any $Y \subseteq X$ as input and returns the list of sets in $\mathcal{R}|_Y$ (each set represented explicitly). We assume that the time to do that is $O(|Y|^{d+1})$, which is linear in the maximum possible size of the oracle’s output, where d is the shatter function exponent. For example, in the case of points and disks in the plane, we have $d = 3$, and so this assumes that, given n points, we can enumerate all subsets enclosed by a disk in time $O(n^4)$. To do this, enumerate all k-tuples of points ($k \leq 3$) and, for each tuple, find which points lie inside the smallest disk enclosing the k points. The main result below is stated in terms of the shatter function exponent d, but the same results hold if d denotes the VC-dimension.

THEOREM 25.3 *Given a set system $(X, \mathcal{R})$ of shatter function exponent d, for any $\varepsilon \leq 1/2$, an ε-approximation for $(X, \mathcal{R})$ of size $O(d\varepsilon^{-2} \log d\varepsilon^{-1})$ and an ε-net for $(X, \mathcal{R})$ of size $O(d\varepsilon^{-1} \log d\varepsilon^{-1})$ can be computed in time $O(d)^{3d}(\varepsilon^{-2} \log d\varepsilon^{-1})^d |X|$.*

Vapnik and Chervonenkis [28] described a probabilistic construction of ε-approximations in bounded VC-dimension. The deterministic construction stated above is due to Chazelle and Matoušek [8], and builds on earlier work [7, 17, 18, 21]. Haussler and Welzl [15] proved the upper bound on the size of ε-nets. The running time for computing an ε-net was improved to $O(d)^{3d}(\varepsilon^{-1} \log d\varepsilon^{-1})^d |X|$ by Brönnimann, Chazelle, and Matoušek [3], using the concept of a *sensitive ε-approximation.* Komlós, Pach, and Woeginger [16] showed that, for any fixed d, the bound of $O(\varepsilon^{-1} \log \varepsilon^{-1})$ for ε-nets is optimal in the worst case (see also [25]). The situation is different with ε-approximations: if $d > 1$ is the VC dimension, then there exists an ε-approximation for $(X, \mathcal{R})$ of size $O(\varepsilon^{-2+2/(d+1)})$ [22, 24].

An important application of ε-approximations is for estimating how many vertices in an arrangement of hyperplanes in $\mathbf{R}^d$ lie within a given convex region. Let $\Sigma = (H, \mathcal{R})$ be the set system formed by a set H of hyperplanes in $\mathbf{R}^d$, where each $R \in \mathcal{R}$ is the subset of H intersected by an arbitrary line segment. Let σ be a convex body (not necessarily full-dimensional). In the arrangement formed by H within the affine span of σ, let $V(H,\sigma)$ be the set of vertices that lie inside σ. The following was proven in [3, 4].

THEOREM 25.4 *Given a set H of hyperplanes in $\mathbf{R}^d$ in general position, let A be an ε-approximation for $\Sigma = (H, \mathcal{R})$. Given any convex body σ of dimension $k \le d$,*

$$\left| \frac{|V(H,\sigma)|}{|H|^k} - \frac{|V(A,\sigma)|}{|A|^k} \right| \le \varepsilon.$$

25.2.2 Optimal Cuttings

For convenience of exposition, we may assume that the set H of n hyperplanes in $\mathbf{R}^d$ is in general position. Let $\mathcal{A}(H)$ denote the arrangement formed by H. Obviously, no simplex of an ε-cutting can enclose more than $O(\varepsilon n)^d$ vertices. Since $\mathcal{A}(H)$ itself has exactly $\binom{n}{d}$ vertices, we should expect to need at least on the order of ε^{-d} simplices. But this is precisely the upper bound claimed in Theorem 25.1, which therefore is asymptotically tight.

Our starting point is an ε-net N for H, where the underlying set system $(X, \mathcal{R})$ is formed by a set X of hyperplanes and the collection $\mathcal{R}$ of subsets obtained by intersecting X with all possible open d-simplices. Its VC-dimension is bounded, and so by Theorem 25.3 an ε-net N of size $O(\varepsilon^{-1}\log\varepsilon^{-1})$ can be found in $n\varepsilon^{-O(1)}$ time.

We need to use a systematic way to triangulate the arrangement formed by the ε-net. We build a *canonical triangulation* of $\mathcal{A}(N)$ by induction on the dimension d (Fig. 25.2). The case $d = 1$ is trivial, so we assume that $d > 1$.

1. Rank the vertices of $\mathcal{A}(N)$ by the lexicographic order of their coordinate sequences.
2. By induction, form a canonical triangulation of the $(d-1)$-dimensional arrangement made by each hyperplane with respect to the $n-1$ others.
3. For each cell (ie, full-dimensional face) σ of $\mathcal{A}(N)$, lift toward its lowest-ranked vertex v each k-simplex $(k = 0, \ldots, d-2)$ on the triangulated boundary of σ that does not lie in a $(d-1)$-face of $\mathcal{A}(N)$ that is incident to v.

It is not hard to see that the combinatorial complexity (ie, number of all faces of all dimensions) of the canonical triangulation of $\mathcal{A}(N)$ is asymptotically the same as that of $\mathcal{A}(N)$, which is $O(\varepsilon^{-1}\log\varepsilon^{-1})^d$. Therefore, the closures of its cells constitute an ε-cutting for H of size $O(\varepsilon^{-1}\log\varepsilon^{-1})^d$, which is good but not perfect. For optimality we must remove the log factor.

Assume that we have at our disposal an optimal method for building an ε_0-cutting of size $O(\varepsilon_0^{-d})$, for some suitably small constant ε_0. To bootstrap this into an optimal ε-cutting construction for any ε, we might proceed as follows: Beginning with a constant-size cutting, we progressively refine it by producing several generations of finer and finer cuttings, $\mathcal{C}_1, \mathcal{C}_2$, etc, where $\mathcal{C}_k$ is an ε_0^k-cutting for H of size $O(\varepsilon^{-dk})$. Specifically, assume that we have recursively computed the cutting $\mathcal{C}_k$ for H. For each $\sigma \in \mathcal{C}_k$, we have the incidence list H_σ of the hyperplanes intersecting the interior of σ. To compute the next-generation cutting $\mathcal{C}_{k+1}$, consider refining each σ in turn as follows:

1. Construct an ε_0-cutting for H_σ, using the algorithm whose existence is assumed.

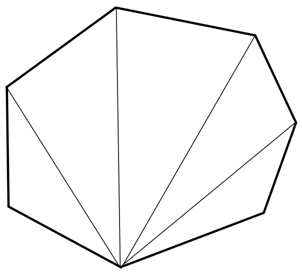


FIGURE 25.2: A canonical triangulation.

2. Retain only those simplices that intersect σ and clip them outside of σ.
3. In case the clipping produces nonsimplicial cells within σ, retriangulate them "canonically" (Fig. 25.3).

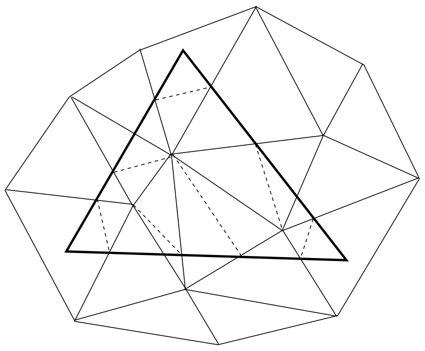


FIGURE 25.3: Clip and retriangulate.

Let $\mathcal{C}_{k+1}$ denote the collection of new simplices. A simplex of $\mathcal{C}_{k+1}$ in σ is cut (in its interior) by at most $\varepsilon_0|H_\sigma|$ hyperplanes of H_σ, and hence of H. By induction, this produces at most $n\varepsilon_0^{k+1}$ cuts; therefore, $\mathcal{C}_{k+1}$ is an ε_0^{k+1}-cutting. The only problem is that $\mathcal{C}_{k+1}$ might be a little too big. The reason is that excess in size builds up from generation to generation. We circumvent this difficulty by using a global parameter that is independent of the construction; namely, the total number of vertices.

Note that we may assume that $|H_\sigma| > n\varepsilon_0^{k+1}$, since σ would otherwise already satisfy the requirement of the next generation. We distinguish between *full* and *sparse* simplices. Given a set X of hyperplanes and a d-dimensional (closed) simplex σ, let $v(X, \sigma)$ be the number of vertices of $\mathcal{A}(X)$ in the interior of σ.

- The simplex $\sigma \in \mathcal{C}_k$ is *full* if $v(H, \sigma) \geq c_0|H_\sigma|^d$, where $c_0 = \varepsilon_0^2$. If so, we compute an ε_0-net for H_σ, and triangulate the portion of the net's arrangement within σ to form an ε_0-cutting of size $O(\varepsilon_0^{-1} \log \varepsilon_0^{-1})^d$. Its simplices form the elements of

$\mathcal{C}_{k+1}$ that lie within σ.

- A simplex σ that is not full is *sparse*. If so, we find a subset H_σ^o of H_σ that satisfies two conditions:

  (i) The canonically triangulated portion of $\mathcal{A}(H_\sigma^o)$ that lies inside σ consists of a set $\mathcal{C}_\sigma^o$ of at most $\frac{1}{2}\varepsilon_0^{-d}$ full-dimensional (closed) simplices.

  (ii) Each simplex of $\mathcal{C}_\sigma^o$ is intersected in its interior by at most $\varepsilon_0|H_\sigma|$ hyperplanes of H.

  The elements of $\mathcal{C}_{k+1}$ within σ are precisely the simplices of $\mathcal{C}_\sigma^o$.

LEMMA 25.3 $\mathcal{C}_{k+1}$ is an ε_0^{k+1}-cutting of size $O(\varepsilon_0^{-d(k+1)})$.

Next, we explain how to enforce conditions (i) and (ii) for sparse simplices. To be able to distinguish between full and sparse simplices, we use a $c_0/2$-approximation A_σ for H_σ of constant size, which we can build in $O(|H_\sigma|)$ time (Theorem 25.3). It follows from Theorem 25.4 that

$$\left| \frac{v(H,\sigma)}{|H_\sigma|^d} - \frac{v(A_\sigma,\sigma)|}{|A_\sigma|^d} \right| \le \frac{c_0}{2}\,; \tag{25.1}$$

therefore, we can estimate $v(H,\sigma)$ in constant time with an error of at most $\frac{c_0}{2}|H_\sigma|^d$, which for our purposes here is inconsequential.

How do we go about refining σ and how costly is it? If σ is a full simplex, then by Theorem 25.3, we can compute the required ε_0-net in $O(|H_\sigma|)$ time. Within the same amount of time, we can also find the new set of simplices in σ, together with all of their incidence lists.

The refinement of a sparse simplex σ is a little more involved. We begin with a randomized construction, from which we then remove all the randomness. We compute H_σ^o by choosing a random sample from A_σ of size $c_1\varepsilon_0^{-1}\log\varepsilon_0^{-1}$, for some constant c_1 large enough (independent of ε_0). It can be shown that, with probability at least 2/3, the sample forms an $(\varepsilon_0/2)$-net for A_σ. By Lemma 25.2, H_σ^o is a $(c_0/2+\varepsilon_0/2)$-net for H_σ; therefore, we ensure that (ii) holds with probability at least 2/3. A slightly more complex analysis shows that (i) also holds with probability at least 2/3; therefore (i,ii) are both true with probability at least 1/3. We derandomize the construction in a trivial manner by trying out all possible samples, which takes constant time; therefore, the running time for refining σ is $O(|H_\sigma|)$.

Putting everything together, we see that refining any simplex takes time proportional to the total size of the incidence lists produced. By Lemma 25.3, the time needed for building generation $k+1$ is $O(n\varepsilon_0^{-(d-1)(k+1)})$. The construction goes on until we reach the first generation such that $\varepsilon_0^k \le \varepsilon$. This establishes Theorem 25.1.

From the proof above it is not difficult to derive a rough estimate on the constant factor in the $O(\varepsilon^{-d})$ bound on the size of an ε-cutting. A thorough investigation into the smallest possible constant was undertaken by Har-Peled [14] for the two-dimensional case.

25.3 Applications

Cuttings have numerous uses in computational geometry. We mention just a handful: point location, Hopcroft's problem, convex hulls, Voronoi diagrams, and range searching. In many cases, cuttings allow us to derandomize existing probabilistic solutions, ie, to remove any need for random bits and thus produce deterministic algorithms. Many other applications are described in the survey [2].

25.3.1 Point Location

How do we preprocess n hyperplanes in $\mathbf{R}^d$, so that, given a query point q, we can quickly find the face of the arrangement formed by the hyperplanes that contains the point? For an answer, simply set $\varepsilon = 1/n$ in Theorem 25.1, and use the nesting structure of $\mathcal{C}_1$, $\mathcal{C}_2$, etc, to locate q in $\mathcal{C}_k$. Note that this can be done in constant time once we know the location in $\mathcal{C}_{k-1}$.

THEOREM 25.5 *Point location among n hyperplanes can be done in $O(\log n)$ query time, using $O(n^d)$ preprocessing.*

Observe that if we only wish to determine whether the point q lies on one of the hyperplanes, it is possible to cut down the storage requirement a little. To do that, we use an ε-cutting for $\varepsilon = (\log n)/n$. The cells associated with the bottom of the hierarchy are each cut by $O(\log n)$ hyperplanes, which we can therefore check one by one. This reduces the storage to $O(n^d/(\log n)^{d-1})$.

25.3.2 Hopcroft's problem

Given n points and n lines in $\mathbf{R}^2$, is there any incidence between points and lines? This is *Hopcroft's problem.* It is self-dual; therefore dualizing it won't help. A classical arrangement of n lines due to Erdős has the property that its n highest-degree vertices are each incident to $\Omega(n^{1/3})$ edges. By picking these n lines as input to Hopcroft's problem and positioning the n points in the near vicinity of these high-degree vertices, we get a sense (not a proof) that to solve the problem should require checking each point against the $\Omega(n^{1/3})$ lines incident to their nearby vertex. This leads to an $\Omega(n^{4/3})$ running time, which under some realistic (though restrictive) conditions, can be made into a rigorous lower bound [13]. At the very least this line of reasoning suggests that to beat $\Omega(n^{4/3})$ is unlikely to be easy. This bound has almost been achieved by an algorithm of Matoušek [20] with, at its heart, a highly intricate and subtle use of cuttings.

THEOREM 25.6 *To decide whether n points and n lines in the plane are free of any incidence can be done in $n^{4/3}\, 2^{O(\log^* n)}$ time.*

25.3.3 Convex Hulls and Voronoi Diagrams

Cuttings play a key role in computing convex hulls in higher dimension. Given n points in $\mathbf{R}^d$, their convex hull is a bounded convex polytope with $O(n^{\lfloor d/2 \rfloor})$ vertices. Of course, it may have much fewer of them: eg, $d+1$, if $n-d-1$ points lie strictly inside the convex hull of the $d+1$ others. It is notoriously difficult to design *output-sensitive* algorithms, the term designating algorithms whose running time is a function of both input and output sizes. In the "worst case" approach our goal is a simpler one: to design an optimal convex hull algorithm that runs in $O(n \log n + n^{\lfloor d/2 \rfloor})$ time. (The extra term $n \log n$ is unavoidable because sorting is easily embedded as a convex hull problem.)

Computing the convex hull of n points is equivalent by duality to computing the intersection of n halfspaces. A naive approach to this problem is to insert each halfspace one after the other while maintaining the intersection of previously inserted halfspaces incrementally. This can be done without difficulty if we maintain a canonical triangulation of the current intersection polyhedron and update a bipartite graph indicating which hyperplane intersects which cell of the triangulation. A surprising fact, first proven by Clarkson and Shor [11], is that if the halfspaces are inserted in random order, then the expected running time of the algorithm can be made optimal. By using an elaborate mix of ε-nets, ε-approximations, and ε-cuttings, Chazelle [5] showed how to compute the intersection deterministically in optimal time; his algorithm was subsequently simplified by Brönnimann, Chazelle, and Matoušek [3]; a complete description is also given in the book [6]. This implies the two theorems below.

THEOREM 25.7 *The polyhedron formed by the intersection of n halfspaces in $\mathbf{R}^d$ can be computed in $O(n \log n + n^{\lfloor d/2 \rfloor})$ time.*

Not only does this result give us an optimal deterministic solution for convex hulls, but it also solves the Voronoi diagram problem. Indeed, recall [12, 29] that a Voronoi diagram of n points in $\mathbf{R}^d$ can be "read off" from the facial structure of the convex hull of a lift of the n points into $\mathbf{R}^{d+1}$.

THEOREM 25.8 *The convex hull of a set of n points in $\mathbf{R}^d$ can be computed deterministically in $O(n \log n + n^{\lfloor d/2 \rfloor})$ time. By duality, the Voronoi diagram (or Delaunay triangulation) of a set of n points in $\mathbf{E}^d$ can be computed deterministically in $O(n \log n + n^{\lceil d/2 \rceil})$ time.*

25.3.4 Range Searching

Simplex range searching refers to the problem of preprocessing a set P of n points in $\mathbf{R}^d$ so that, given a query (closed) simplex σ, the size of $P \cap \sigma$ can be quickly evaluated. Variants of the problem include reporting the points of $P \cap \sigma$ explicitly or, assuming that each point p has a weight $w(p) \in \mathbf{R}$, computing $\sum \{ w(p) \mid p \in P \cap \sigma \}$. The most powerful data structure for solving simplex range searching, the *simplicial partition*, vividly illustrates the power of ε-cuttings. A collection $\{(P_i, R_i)\}$ is called a *simplicial partition* if

- the collection $\{P_i\}$ forms a partition of P; and
- each R_i is a relatively open simplex that contains P_i.

The simplices R_i can be of any dimension and, in fact, need not even be disjoint; furthermore the P_i's need not be equal to $P \cap R_i$. A hyperplane is said to *cut* R_i if it intersects, but does not contain, R_i. The *cutting number* of the simplicial partition refers to the maximum number of R_i's that can be cut by a single hyperplane. Matoušek [19] designed an optimal construction, which happens to be crucially based on ε-cuttings.

LEMMA 25.4 Given a set P of n points in $\mathbf{R}^d$ $(d > 1)$, for any integer $1 < r \le n/2$, there exists a simplicial partition of cutting number $O(r^{1-1/d})$ such that $n/r \le |P_i| < 2n/r$ for each (P_i, R_i) in the partition.

To understand the usefulness of simplicial partitions for range searching, one needs to learn about *partition trees*. A partition tree for P is a tree $\mathcal{T}$ whose root is associated with the point set P. The set P is partitioned into subsets $P_1, \ldots, P_m$, with each P_i associated with a distinct child v_i of the root. To each v_i corresponds a convex open set R_i, called the *region* of v_i, that contains P_i. The regions R_i are not necessarily disjoint. If $|P_i| > 1$, the subtree rooted at v_i is defined recursively with respect to P_i.

Armed with a partition tree, it is a simple matter to handle range search queries. In preprocessing, at each node we store the sum of the weights of the points associated with the corresponding region. To answer a query σ, we visit all the children v_i of the root and check whether σ intersects the region R_i of v_i: (i) if the answer is yes, but σ does not completely enclose the region R_i of v_i, then we visit v_i and recurse; (ii) if the answer is yes, but σ completely encloses R_i, we add to our current weight count the sum of the weights within P_i, which happens to be stored at v_i; (iii) if the answer is no, then we do not recurse at v_i.

It is straightforward to see that Lemma 25.4 can be used to construct partition trees. It remains for us to choose the branching factor. If we choose a large enough constant r, we end up with a partition tree that lets us answer simplex range search queries in $O(n^{1-1/d+\varepsilon})$ time for any fixed $\varepsilon > 0$, using only $O(n)$ storage. A more complex argument by Matoušek [19] removes the ε term from the exponent.

With superlinear storage, various space-time tradeoffs can be achieved. For example, as shown by Chazelle, Sharir, and Welzl [9], simplex range searching with respect to n points in $\mathbf{R}^d$ can be done in $O(n^{1+\varepsilon}/m^{1/d})$ query time, using a data structure of size m, for any $n \le m \le n^d$. Matoušek [20] slightly improved the query time to $O(n(\log m/n)^{d+1}/m^{1/d})$, for m/n large enough. These bounds are essentially optimal under highly general computational models [6].

Acknowledgments

This work was supported in part by NSF grant CCR-998817, and ARO Grant DAAH04-96-1-0181.

References

[1] Agarwal, P.K. Partitioning arrangements of lines II: Applications, *Disc. Comput. Geom.* 5 (1990), 533–573.

[2] Agarwal, P.K. Geometric partitioning and its applications, in *Computational Geometry: Papers from the DIMACS Special Year,* eds., Goodman, J.E., Pollack, R., Steiger, W., Amer. Math. Soc., 1991.

[3] Brönnimann, H., Chazelle, B., Matoušek, J. Product range spaces, sensitive sampling, and derandomization, *SIAM J. Comput.* 28 (1999), 1552–1575.

[4] Chazelle, B. Cutting hyperplanes for divide-and-conquer, *Disc. Comput. Geom.* 9 (1993), 145–158.

[5] Chazelle, B. An optimal convex hull algorithm in any fixed dimension, *Disc. Comput. Geom.* 10 (1993), 377–409.

[6] Chazelle, B. *The Discrepancy Method: Randomness and Complexity*, Cambridge University Press, hardcover 2000, paperback 2001.

[7] Chazelle, B., Friedman, J. A deterministic view of random sampling and its use in geometry, *Combinatorica* 10 (1990), 229–249.

[8] Chazelle, B., Matoušek, J. On linear-time deterministic algorithms for optimization problems in fixed dimension, *J. Algorithms* 21 (1996), 579–597.

[9] Chazelle, B., Sharir, M., Welzl, E. Quasi-optimal upper bounds for simplex range searching and new zone theorems, *Algorithmica* 8 (1992), 407–429.

[10] Clarkson, K.L. New applications of random sampling in computational geometry, *Disc. Comput. Geom.* 2 (1987), 195–222.

[11] Clarkson, K.L., Shor, P.W. Applications of random sampling in computational geometry, II, *Disc. Comput. Geom.* 4 (1989), 387–421.

[12] Edelsbrunner, H. *Algorithms in Combinatorial Geometry*, Springer, 1987.

[13] Erickson, J. New lower bounds for Hopcroft's problem, *Disc. Comput. Geom.* 16 (1996), 389–418.

[14] Har-Peled, S. Constructing planar cuttings in theory and practice, *SIAM J. Comput.* 29 (2000), 2016–2039.

[15] Haussler, D., Welzl, E. ε-nets and simplex range queries, *Disc. Comput. Geom.* 2 (1987), 127–151.

[16] Komlós, J., Pach, J., Woeginger, G. Almost tight bounds for ε-nets, *Disc. Comput. Geom.* 7 (1992), 163–173.

[17] Matoušek, J. Construction of ε-nets, *Disc. Comput. Geom.* 5 (1990), 427–448.

[18] Matoušek, J. Cutting hyperplane arrangements, *Disc. Comput. Geom.* 6 (1991), 385–406.

[19] Matoušek, J. Efficient partition trees, *Disc. Comput. Geom.* 8 (1992), 315–334.

[20] Matoušek, J. Range searching with efficient hierarchical cuttings, *Disc. Comput. Geom.* 10 (1993), 157–182.

[21] Matoušek, J. Approximations and optimal geometric divide-and-conquer, *J. Comput. Syst. Sci.* 50 (1995), 203–208.

[22] Matoušek, J. Tight upper bounds for the discrepancy of halfspaces, *Disc. Comput. Geom.* 13 (1995), 593–601.

[23] Matoušek, J. *Geometric Discrepancy: An Illustrated Guide*, Algorithms and Combinatorics, 18, Springer, 1999.

[24] Matoušek, J., Welzl, E., Wernisch, L. Discrepancy and ε-approximations for bounded VC-dimension, *Combinatorica* 13 (1993), 455–466.

[25] Pach, J., Agarwal, P.K. *Combinatorial Geometry*, Wiley-Interscience Series in Discrete Mathematics and Optimization, John Wiley & Sons, Inc., 1995.

[26] Sauer, N. On the density of families of sets, *J. Combinatorial Theory A* 13 (1972), 145–147.

[27] Shelah, S. A combinatorial problem; stability and order for models and theories in infinitary languages, *Pac. J. Math.* 41 (1972), 247–261.

[28] Vapnik, V.N., Chervonenkis, A.Ya. On the uniform convergence of relative frequencies of events to their probabilities, *Theory of Probability and its Applications* 16 (1971), 264–280.

[29] Ziegler, G.M. *Lectures on Polytopes*, Graduate Texts in Mathematics, 152, Springer-Verlag, 1995.

26

Approximate Geometric Query Structures

Christian A. Duncan
University of Miami

Michael T. Goodrich
University of California, Irvine

26.1 Introduction

Specialized data structures are useful for answering specific kinds of geometric queries. Such structures are tailor-made for the kinds of queries that are anticipated and even then there are cases when producing an exact answer is only slightly better than an exhaustive search. For example, Chazelle and Welzl [7] showed that triangle range queries can be solved in $O(\sqrt{n}\log n)$ time using linear space but this holds only in the plane. In higher dimensions, the running times go up dramatically, so that, in general, the time needed to perform an exact simplex range query and still use small linear space is roughly $\Omega(n^{1-1/d})$, ignoring logarithmic factors [6]. For orthogonal range queries, efficient query processing is possible if superlinear space is allowed. For example, range trees (Chapter 18) can answer orthogonal range queries in $O(\log^{d-1} n)$ time but use $O(n\log^{d-1} n)$ space [17].

In this chapter, we focus instead on general-purpose data structures that can answer nearest-neighbor queries and range queries using linear space. Since the lower-bound of Chazelle [6] applies in this context, in order to get query bounds that are significantly faster than exhaustive search, we need to compromise somewhat on the exactness of our answers. That is, we will answer all queries *approximately*, giving responses that are within an arbitrarily small constant factor of the exact solution. As we discuss, such responses can typically be produced in logarithmic or polylogarithmic time, using linear space. Moreover, in many practical situations, a good approximate solution is often sufficient.

In recent years several interesting data structures have emerged that efficiently solve several general kinds of geometric queries approximately. We review three major classes of such structures in this chapter. The first one we discuss is a structure introduced by Arya *et al.* [1] for efficiently approximating nearest-neighbor queries in low-dimensional space. Their work developed a new structure known as the balanced box decomposition (BBD) tree. The BBD tree is a variant of the quadtree and octree [14] but is most closely related to the fair-split tree of Callahan and Kosaraju [5]. In [3], Arya and Mount extend the structure to show that it can also answer approximate range queries. Their structure is based on the decomposition of space into "boxes" that may have a smaller box "cut out;" hence, the boxes may not be convex. The second general purpose data structure we discuss is the balanced aspect ratio (BAR) tree of Duncan *et al.* [11–13], which is a structure that has similar performance as the BBD tree but decomposes space into convex regions. Finally, we discuss an analysis of a type of k-d tree [16] that helps to explain why k-d trees have long been known to exhibit excellent performance bounds in practice for general geometric queries. In particular, we review a result of Dickerson *et al.* [9, 11], which shows that one of the more common variants, the maximum-spread k-d tree, exhibits properties similar to BBD trees and BAR trees; we present efficient bounds on approximate geometric queries for this variant. Unfortunately, the bounds are not as efficient as the BBD tree or BAR tree but are comparable.

26.2 General Terminology

In order to discuss approximate geometric queries and the efficient structures on them without confusion, we must cover a few fundamental terms. We distinguish between general points in $\mathbb{R}^d$ and points given as input to the structures.

For a given metric space $\mathbb{R}^d$, the coordinates of any point $p \in \mathbb{R}^d$ are $(p_1, p_2, \ldots, p_d)$. When necessary to avoid confusion, we refer to points given as input in a set S as *data points* and general points in $\mathbb{R}^d$ as *real points.* For two points $p, q \in \mathbb{R}^d$, the L_m *metric distance* between p and q is

$$\delta(p,q) = (\sum_{i=1}^{d} |p_i - q_i|^m)^{\frac{1}{m}}.$$

Although our analysis will concentrate on the Euclidean L_2 metric space, the data structures mentioned in this chapter work in all of the L_m metric spaces.

In addition, we use the standard notions of (convex) regions R, rectangular boxes, hyperplanes H, and hyperspheres B. For each of these objects we define two distance values. Let P and Q be any two regions in $\mathbb{R}^d$, the *minimum* and *maximum* metric distances between P and Q are

$$\delta(P,Q) = \min_{p \in P, q \in Q} \delta(p,q) \text{ and}$$

$$\Delta(P,Q) = \max_{p \in P, q \in Q} \delta(p,q) \text{ respectively.}$$

Notice that this definition holds even if one or both regions are simply points.

Let S be a finite data set $S \subset \mathbb{R}^d$. For a subset $S_1 \subseteq S$, the *size* of S_1, written $|S_1|$, is the number of distinct data points in S_1. More importantly, for any region $R \subset \mathbb{R}^d$, the size is $|R| = |R \cap S|$. That is, the *size* of a region identifies the number of data points in it. To refer to the physical size of a region, we define the *outer radius* as $O_R = \min_{R \subseteq B_r} r$, where B_r is defined as the hypersphere with radius r. The *inner radius* of a region is $I_R = \max_{B_r \subseteq R} r$. The outer radius, therefore, identifies the smallest ball that contains the region R whereas the inner radius identifies the largest ball contained in R.

In order to discuss balanced aspect ratio, we need to define the term.

DEFINITION 26.1 A convex region R in $\mathbb{R}^d$ has aspect ratio $\texttt{asp}(R) = O_R/I_R$ with respect to some underlying metric. For a given balancing factor α, if $\texttt{asp}(R) \leq \alpha$, R has *balanced aspect ratio* and is called an α*-balanced* region. Similarly, a collection of regions $\mathcal{R}$ has balanced aspect ratio for a given factor α if each region $R \in \mathcal{R}$ is an α-balanced region.

For simplicity, when referring to rectangular boxes, we consider the aspect ratio as simply the ratio of the longest side to the shortest side. It is fairly easy to verify that the two definitions are equal within a constant factor. As is commonly used, we refer to regions as being either *fat* or *skinny* depending on whether their aspect ratios are balanced or not.

The class of structures that we discuss in this chapter are all derivatives of binary space partition (BSP) trees, see for example [18]. See also Chapter 20. Each node u in a BSP tree T represents both a *region* R_u in space and the *data subset* $S_u \subseteq S$ of objects, points, lying inside R_u. For simplicity, regions are considered closed and points falling on the boundary of two regions can be in either of the two regions but not both. Each leaf node in T represents a region with a constant number of data objects, points, from S. Each internal node in T has an associated cut partitioning the region into two subregions, each a child node. The root of T is associated with some bounding (rectangular box) region containing S. In general, BSP trees can store any type of object, points, lines, solids, but in our case we focus on points. Typically, the partitioning cuts used are hyperplanes resulting in convex regions. However, the BBD tree presented in Section 26.5 is slightly different and can introduce regions with a single interior hole. Therefore, we have generalized slightly to accommodate this in our definition.

26.3 Approximate Queries

Before elaborating on the structures and search algorithms used to answer certain geometric queries, let us first introduce the basic definitions of approximate nearest-neighbor, farthest-neighbor, and range queries, see [1–3, 11, 13].

DEFINITION 26.2 Given a set S of points in $\mathbb{R}^d$, a query point $q \in \mathbb{R}^d$, a (connected) query region $Q \subset \mathbb{R}^d$, and $\epsilon > 0$, we define the following queries (see Figure 26.1):

- A point $p^* \in S$ is a *nearest neighbor of* q if $\delta(p^*, q) \leq \delta(p, q)$ for all $p \in S$.
- A point $p^* \in S$ is a *farthest neighbor of* q if $\delta(p^*, q) \geq \delta(p, q)$ for all $p \in S$.
- A point $p \in S$ is a $(1+\epsilon)$*-nearest neighbor of* q if $\delta(p, q) \leq (1+\epsilon)\delta(p^*, q)$, where p^* is the true nearest neighbor of q.
- A point $p \in S$ is a $(1-\epsilon)$*-farthest neighbor of* q if $\delta(p, q) \geq \delta(p^*, q) - \epsilon O_S$, where p^* is the true farthest neighbor of q.
- An ϵ*-approximate range query* returns or counts a subset $S' \subseteq S$ such that $S \cap Q \subseteq S'$ and for every point $p \in S'$, $\delta(p, Q) \leq \epsilon O_Q$.

To clarify further, a point p is a $(1+\epsilon)$-approximate nearest neighbor if its distance is within a constant error factor of the true nearest distance. Although we do not discuss it here, we can extend the definitions to report a sequence of k $(1+\epsilon)$-nearest (or $(1-\epsilon)$-farthest) neighbors. One may also wonder why the approximate farthest neighbor is defined in absolute terms instead of relative terms as with the nearest version. By observing that the distance from any query point to its farthest neighbor is always at least the radius of the point set O_S, one can see that the approximation is as good as the relative approximation. Moreover, if the query point is extremely far from the point set, then a relative approximation could return any point in S, whereas an absolute approximation would require a much

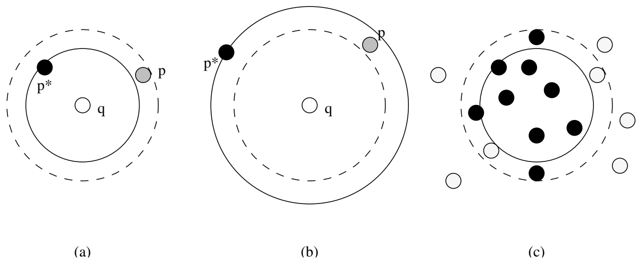


FIGURE 26.1: Examples of (a) an approximate nearest-neighbor p to q with the exact neighbor p^*, (b) an approximate farthest-neighbor p to q with the exact neighbor p^*, and (c) an approximate range query; here the dark points are reported or counted and the lighter points are not.

more accurate distance.

Although we do not modify our definition here, one can extend this notion and our later theorems to compensate for the problem of query points distant from the point set in nearest-neighbor queries as well. In other words, when the query point is relatively close to the entire data set, we can use the relative error bound, and when it is relatively far away from the entire data set, we can use the absolute error bound.

The approximate range searching problem described above has one-sided false-positive errors. We do not miss any valid points, but we may include erroneous points near the range boundary. It is a simple modification of the query regions to instead get false-negative errors. That is, we could instead require including only points that are inside of Q but allow missing some points that are inside of Q if they are near the border. In fact, for any region Q one could define two epsilon ranges ϵ_i and ϵ_o for both interior and exterior error bounds and then treat the approximation factor as $\epsilon = \epsilon_i + \epsilon_o$.

There are numerous approaches one may take to solving these problems. Arya *et al.* [1] introduced a priority search algorithm for visiting nodes in a partition tree to solve nearest-neighbor queries. Using their BBD tree structure, they were able to prove efficient query times. Duncan *et al.* [13] show how this priority search could also solve farthest-neighbor queries. The nearest and farthest neighbor priority searching algorithms shown in Figures 26.2 and 26.3 come from [11]. In the approximate nearest-neighbor, respectively farthest-neighbor, search, nodes are visited in order of closest node, respectively farthest node. Nodes are extracted via an efficient priority queue, such as the Fibonacci heap [10, 15].

Introduced by [3], the search technique used for the approximate range query is a modification to the standard range searching algorithm for regular partition trees. We present the algorithm from [11] in Figure 26.4. In this algorithm, we have two different query regions, the inner region Q and the outer region $Q' \supseteq Q$. The goal is to return all points in S that lie inside Q, allowing some points to lie inside Q' but none outside of Q'. That is $Q' - Q$ defines a buffer zone that is the only place allowed for erroneous points. Whenever a node u is visited, if u is a leaf node, we simply check all of u's associated data points. Otherwise, if R_u does not intersect Q, we know that none of its points can lie in Q, and we ignore u and its subtree. If R_u lies completely inside Q' then all of the data points in its subtree must lie inside Q', and we report all points. Otherwise, we repeat the process on u's two child nodes. For an ϵ-approximate range search, we define $Q' = \{p \in \mathbb{R}^d | \delta(p, Q) \leq \epsilon O_Q\}$. We note that this search algorithm can also be modified to return the count or sum of the weights of the

```
ApproximateNearestNeighbor($T$,$q$,$\epsilon$)
    Arguments: BSP tree, $T$, query point $q$, and error factor $\epsilon$
    Returns: A $(1+\epsilon)$-nearest neighbor $p$
    $Q \leftarrow$ root($T$)
    $p \leftarrow \infty$
    do $u \leftarrow Q$.extractMin()
       if $\delta(u,q) > \delta(p,q)/(1+\epsilon)$
          return $p$
       while $u$ is not a leaf
          $u_1 \leftarrow$ leftChild($u$)
          $u_2 \leftarrow$ rightChild($u$)
          if $\delta(u_1,q) \leq \delta(u_2,q)$
             $Q$.insert($\delta(u_2,q), u_2$)
             $u \leftarrow u_1$
          else
             $Q$.insert($\delta(u_1,q), u_1$)
             $u \leftarrow u_2$
       end while
       // $u$ is now a leaf
       for all $p'$ in dataSet($u$)
          if $\delta(p',q) < \delta(p,q)$
             $p \leftarrow p'$
    repeat
```

FIGURE 26.2: The basic algorithm to perform nearest-neighbor priority searching.

points inside the approximate range rather than explicitly reporting the points.

In all of these search algorithms, the essential criteria behind the running time is the observation that a non-terminating node in the search, one that requires expansion of its child nodes, is a node that must cross certain size boundaries. For example, in the approximate range searching algorithm, the only nodes expanded are those whose region lies partially inside Q, else it would be discarded, and partially outside Q', else it would be completely counted in the output size. A slightly more complex but similar argument applies for nearest and farthest neighbor algorithms. In the next section, we discuss a general theorem providing provable running time bounds for partition trees satisfying a fundamental packing argument.

26.4 Quasi-BAR Bounds

We are now ready to examine closely a sufficient condition for a data structure to guarantee efficient performance on the aforementioned searches. Before we can proceed, we must first discuss a few more basic definitions presented in Dickerson *et al.* [9].

DEFINITION 26.3 For any region R, the *region annulus with radius* r, denoted $A_{R,r}$ is the set of all points $p \in \mathbb{R}^d$ such that $p \notin R$ and $\delta(p,R) < r$. A region R' *pierces* an annulus $A_{R,r}$ if and only if there exist two points $p, q \in R'$ such that $p \in R$ and $q \notin R \cup A_{R,r}$.

In other words, an annulus $A_{R,r}$ contains all points outside but near the region R. If R were a sphere of radius r', this would be the standard definition of an annulus with inner radius r' and outer radius $r'+r$. For convenience, when the region and radius of an annulus are understood, we use A. Figure 26.5 illustrates the basic idea of a spherical annulus with multiple piercing regions.

The core of the performance analysis for the searches lies in a critical packing argument.

```
ApproximateFarthestNeighbor($T$,$q$,$\epsilon$)
    Arguments: BSP tree, $T$, query point $q$, and error factor $\epsilon$
    Returns: A $(1 - \epsilon)$-farthest neighbor $p$
    $Q \leftarrow$ root($T$)
    $p \leftarrow q$
    do $u \leftarrow Q$.extractMax()
       if $\Delta(u, q) \leq \delta(p, q) + \epsilon D$
          return $p$
       while $u$ is not a leaf
          $u_1 \leftarrow$ leftChild($u$)
          $u_2 \leftarrow$ rightChild($u$)
          if $\Delta(u_1, q) \geq \Delta(u_2, q)$
             $Q$.insert($\Delta(u_2, q), u_2$)
             $u \leftarrow u_1$
          else
             $Q$.insert($\Delta(u_1, q), u_1$)
             $u \leftarrow u_2$
       end while
       // $u$ is now a leaf
       for all $p'$ in dataSet($u$)
          if $\delta(p', q) > \delta(p, q)$
             $p \leftarrow p'$
    repeat
```

FIGURE 26.3: The basic algorithm to perform farthest-neighbor priority searching.

```
ApproximateRangeSearch($u$, $Q$, $Q'$)
    Arguments: Node $u$ in a BSP tree, inner region $Q$, outer region $Q'$
          Initially, $u \leftarrow$ root($T$).
    Reports: All points in the approximate range defined by $Q$ and $Q'$
    if $u$ is a leaf node
       for all $p$ in dataSet($u$)
          if $p \in Q$
             output $p$
    else if $R_u \subseteq Q'$
          // The region lies completely inside $Q'$
          output all points $p$ in the subtree of $u$
    else if $R_u \cap Q \neq \emptyset$
       // The region lies partially inside $Q$
       call ApproximateRangeSearch(leftChild($u$), $Q$, $Q'$)
       call ApproximateRangeSearch(rightChild($u$), $Q$, $Q'$)
```

FIGURE 26.4: The basic range search algorithm.

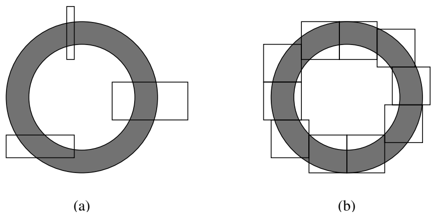


FIGURE 26.5: An example of a simple annulus region (a) with three other regions which pierce this annulus and (b) with several “fat” square regions. Observe that only a limited number of such “fat” squares can pierce the annulus.

The packing lemmas work by bounding the number of disjoint regions that can pierce an annulus and hence simultaneously fit inside the annulus, see Figure 26.5b. When this packing size is small, the searches are efficient. Rather than cover each structure's search analysis separately, we use the following more generalized notion from Dickerson *et al.* [9].

DEFINITION 26.4 Given a BSP tree T and a region annulus A, let $\mathcal{P}(A)$ denote the largest set of disjoint nodes in T whose associated regions pierce A. A class of BSP trees is a $\rho(n)$-*quasi-BAR tree* if, for any tree T in the class constructed on a set S of n points in $\mathbb{R}^d$ and any region annulus $A_{R,r}$, $|\mathcal{P}(A_{R,r})| \leq \rho(n)V_A/r^d$, where V_A is the volume of $A_{R,r}$. The function $\rho(n)$ is called the *packing* function.

Basically, the packing function $\rho(n)$ represents the maximum number of regions that can pierce any query annulus. By proving that a class of BSP trees is a $\rho(n)$-quasi-BAR tree, we can automatically inherit the following theorems proven in [1, 3, 13] and generalized in [9]:

THEOREM 26.1 *Suppose we are given a $\rho(n)$-quasi-BAR tree T with depth $D_T = \Omega(\log n)$ constructed on a set S of n points in $\mathbb{R}^d$. For any query point q, the priority search algorithms in Figures 26.2 and 26.3 find respectively a $(1+\epsilon)$-nearest and a $(1-\epsilon)$-farthest neighbor to q in $O(\epsilon^{1-d}\rho(n)D_T)$ time.*

THEOREM 26.2 *Suppose we are given a $\rho(n)$-quasi-BAR tree T with depth D_T constructed on a set S of n points in $\mathbb{R}^d$. For any convex query region Q, the search algorithm in Figure 26.4 solves an ϵ-approximate range searching query in T in $O(\epsilon^{1-d}\rho(n)D_T)$ time (plus output size in the reporting case). For any general non-convex query region Q, the time required is $O(\epsilon^{-d}\rho(n)D_T)$ (plus output size).*[*]

Trivially, $\rho(n)$ is always less than n but this accomplishes little in the means of getting good bounds. Having a class of trees with both a good packing function and low depth helps guarantee good asymptotic performance in answering geometric queries. One approach to finding such classes is to require that all regions produced by the tree be fat. The idea behind this is that there is a limit to the number of disjoint fat regions that pierce an annulus dependent upon the aspect ratio of the regions and the thickness of the annulus. Unfortunately, guaranteeing fat regions and good depth is not readily possible using the standard BSP trees like k-d trees and octrees. Imagine building a k-d tree using only axis-parallel partitioning cuts. Figure 26.6 illustrates such an example. Here the majority of the points are concentrated at a particular corner of a rectangular region. Now, any axis-parallel cut is either too close to the opposing face or does not partition the points in the region well, resulting in large tree depth.

Fortunately, there are structures that can provably be shown to be $\rho(n)$-quasi-BAR trees with small values for $\rho(n)$ and D_T. The next few sections discuss some of these structures.

[*]Actually, BBD trees and BAR trees have a slightly better running time for these searches and we mention this in the respective sections.

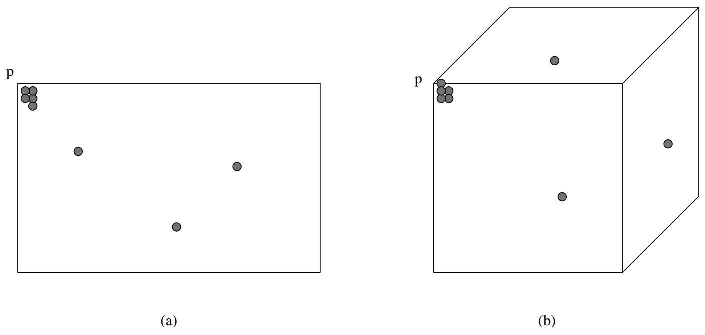


FIGURE 26.6: (a) A bad corner of a simple rectangular region with nearly all of the points clustered near a corner. Notice a cut in either the x or y direction dividing the points located inside p would cause a "skinny" region. (b) The same situation in $\mathbb{R}^3$.

26.5 BBD Trees

Arya *et al.* [1] introduced the first BSP tree structure to guarantee both balanced aspect ratio and $O(\log n)$ depth. In addition, the aspect ratio achieved allowed them to prove a packing constraint. From this, one can verify that the BBD tree has a packing function of $\rho(n) = O(1)$ where the constant factor depends on the dimension d. In the following section, we describe the basic construction of the BBD tree using terminology from [3].

Every region R_u associated with a node u in a BBD tree is either an *outer* rectangular box or the set theoretic difference between an *outer* rectangular box and an *inner* rectangular box. The *size* of a box is the length of its longest side and the *size* of R_u is the size of the outer box. In order to guarantee balanced aspect ratio for these cells, Arya *et al.* [1] introduced a *stickiness* restriction on the inner box. Briefly described, an inner box is *sticky* if the distance between the inner box and every face on the outer box is either 0 or not less than the size of the inner box. Although not necessary to the structure, we shall assume that the aspect ratio of the outer box is no more than two.

The construction of the BBD tree is done by a sequence of alternating splitting and shrinking operations. In the *(midpoint) split* operation, a region is bisected by a hyperplane cut orthogonal to one of the longest sides. This is essentially the standard type of cut used in a quadtree or octree. Its simplicity, speed of computation, and effectiveness are major reasons for preferring these operations.

The *shrink* operation partitions a region by a box lying inside the region, creating an inner region. The shrink operation is actually part of a sequence of up to three operations called a *centroid shrink*. The centroid shrink attempts to partition the region into a small number of subregions R_i such that $|R_i| \leq 2|R_u|/3$.

When R_u is simply an outer box, with no inner box, a centroid operation is performed with one shrink operation. The inner partitioning box is found by conceptually applying midpoint split operations recursively on the subregion with the larger number of points. The process stops when the subregion contains no more than $2|R_u|/3$ points. The outer box of this subregion is the inner partitioning box for the shrink operation. The other merely conceptual midpoint splits are simply ignored. Choosing this inner box guarantees that both subregions produced by the split have no more than $2|R_u|/3$ points. This can be seen by observing that the inner box has no more than $2|R_u|/3$ points and also must contain at least $|R_u|/3$ points. The technique as stated is not theoretically ideal because the number of midpoint split operations computed cannot be bounded. Each midpoint split

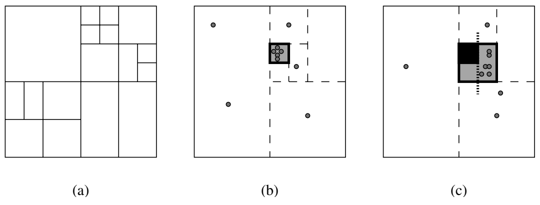


FIGURE 26.7: Examples of (a) multiple iterations of the midpoint split rule, (b) centroid shrinking, with dashed lines representing the conceptual midpoint splits and the highlighted inner box being the actual partition cut, (c) centroid shrinking with an inner box. In the final example, the original inner box is solid, the final midpoint split is shown with slightly extended dotted lines, and the new inner box partition cut is shown shaded in gray.

may not partition even a single point. Arya *et al.* [1, 3] describe a simple solution to this problem, which repeatedly computes the smallest bounding midpoint box using a technique due to Clarkson [8].

When R_u has an inner box associated with it, we cannot simply find another inner box as this would violate the restriction on having only one inner box. Let b_i represent the original inner box. The solution is to proceed as in the previous centroid shrink operation, repeatedly applying midpoint split operations on the subregion with the larger number of points. However, we now stop in one of two situations; when the size of the larger subregion either has no more than $2|R_u|/3$ points or no longer contains b_i. In the former case, let b be the outer box of this subregion. In the latter case, or in the event both cases happen, let b represent the outer box of the subregion prior to this final split. We perform a shrink operation using b as the partitioning box. Since b clearly contains b_i, the subregion associated with the original outer box continues to have one inner box, albeit a slightly larger one than its parent. The subregion R_1, whose outer box is b, also has one inner box, b_i. If $|R_1| > 2|R_u|/3$, we perform a midpoint split on this subregion. Let R_2 be the larger subregion formed by this last split, which we know does not contain b_i. Since R_2 does not contain an inner box, if R_2 contains more than $2|R_u|/3$ points, we simply perform the basic shrink operation thus dividing R_2 into two smaller subregions as well. Clearly, all the subregions produced by this centroid shrink have no more than $2|R_u|/3$ points. Figure 26.7 shows the three main operations, splitting, shrinking, and the three-step shrinking process.

In addition to this simple version of the BBD tree, there are more flexible variations on this approach. The reader should refer to [1, 3] for details on an efficient $O(dn \log n)$ construction algorithm and for discussions on some of the BBD variations. To highlight a few options, at any stage in the construction, rather than alternate between shrinking and splitting operations, it is preferable to perform split operations whenever possible, so long as the point set is divided evenly after every few levels, and to use the more costly shrinking operations only when necessary. Another approach is to use a more flexible split operation, a *fair split*, which attempts to partition the points in the region more evenly. In this case, more care has to be taken to avoid producing skinny regions and to avoid violating the stickiness property; however, as was shown experimentally, the flexibility provides for better experimental performance.

The following theorem summarizes the basic result [1, 3]:

THEOREM 26.3 *Given a set S of n data points in $\mathbb{R}^d$, in $O(dn \log n)$ time it is possible to construct a BBD tree such that*

1. *the tree has $O(n)$ nodes and depth $O(\log n)$,*
2. *the regions have outer boxes with balanced aspect ratio and inner boxes that are sticky to the outer box,*
3. *the sizes of the regions are halved after every $2d$ levels in the tree.*

The above conditions imply that the BBD tree is an $O(1)$-quasi-BAR tree.

The size reduction constraint above helps guarantee a slightly better performance for geometric queries than given for general quasi-BAR trees. In particular, Arya and Mount [3] show that the size reduction allows range queries on BBD trees to be solved in $O(2^d \log n + d(3\sqrt{d}/\epsilon)^d)$ time, or $O(2^d \log n + d^3(3\sqrt{d}/\epsilon)^{d-1})$ for convex queries plus output size. Duncan [11] later extended the separation of the n and ϵ dependencies to nearest and farthest neighbor queries showing that the running time for both is $O(\log n + \epsilon^{1-d} \log(1/\epsilon))$ for fixed dimension d.

26.6 BAR Trees

The balanced aspect ratio tree introduced in [12] for the basic two-dimensional case and subsequently revised to higher dimensions in [11, 13] can be shown to have a packing function of $\rho(n) = O(1)$ where the constant factor depends on the dimension d and a user-specified aspect ratio parameter α. In the following section, we borrow terminology from [11, 13].

Unlike BBD trees, k-d trees, and octrees, BAR trees do not exclusively use axis-orthogonal hyperplane cuts. Instead, to achieve simultaneously the goals of good aspect ratio, balanced depth, and convex regions, cuts in several different directions are used. These directions are called canonical cuts, and the particular choice and size of canonical cuts is essential in creating good BAR trees.

DEFINITION 26.5 The following terms relate to specific cutting directions:

- A *canonical cut set*, $\mathcal{C} = \{\vec{v}_1, \vec{v}_2, \ldots, \vec{v}_\gamma\}$, is a collection of γ not necessarily independent vectors that span $\mathbb{R}^d$ (thus, $\gamma \geq d$).
- A *canonical cut direction* is any vector $\vec{v}_i \in \mathcal{C}$.
- A *canonical cut* is any hyperplane, H, in $\mathbb{R}^d$ with a normal in $\mathcal{C}$.
- A *canonical region* is any region formed by the intersection of a set of hyperspaces defined by canonical cuts, i.e., a convex polyhedron in $\mathbb{R}^d$ with every facet having a normal in $\mathcal{C}$.

Figure 26.8a shows a region composed of three cut directions $(1, 0)$, $(0, 1)$, and $(1, -1)$, or simply cuts along the x, y, and $x - y$ directions. After cutting the region at the dashed line c, we have two regions R_1 and R_2. In R_2 notice the left side is replaced by the new cut c, and more importantly the diagonal cut is no longer tangential to R_2. The following definition describes this property more specifically.

DEFINITION 26.6 A canonical cut c *defines* a canonical region R, written $c \in R$, if and only if c is tangential to R. In other words, c intersects the border of R. For a canonical region R, any two parallel canonical cuts $b, c \in R$ are *opposing canonical cuts*. For any canonical region R, we define the *canonical bounding cuts* with respect to a direction $\vec{v}_i \in \mathcal{C}$ to be the two unique opposing canonical cuts normal to $\vec{v}_i$ and tangent to R. We often

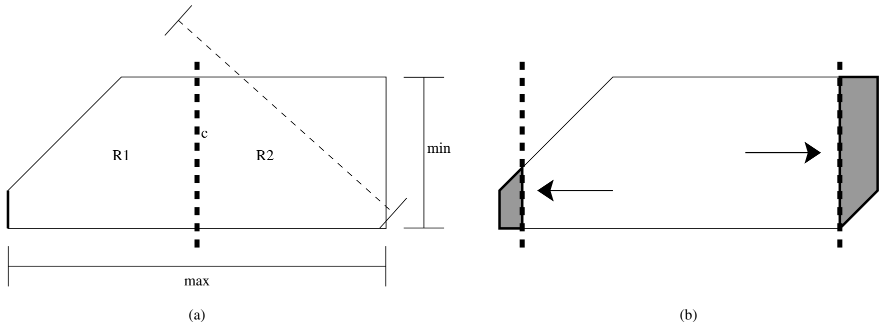


FIGURE 26.8: (a) An example of a canonical region of three cut directions, x,y, and $x-y$. Observe the three widths highlighted with lines and the min and max widths of the region. The left facet associated with the x direction is drawn in bold. The bold dashed line in the center represents a cut c and the respective subregions R_1 and R_2. (b) An example of a similar region highlighting the two shield regions associated with the x-direction for $\alpha \approx 4$. Notice the size difference between the two shield regions corresponding to the associated facet sizes.

refer to these cuts as b_i and c_i or simply b and c when i is understood from the context. Intuitively, R is "sandwiched" between b_i and c_i. To avoid confusion, when referring to a canonical cut of a region R, we *always* mean a canonical bounding cut.

For any canonical bounding cut, c, the *facet* of $c \in R$, $\texttt{facet}_c(R)$, is defined as the region formed by the intersection of R with c.

The canonical set used to define a partition tree can vary from method to method. For example, the standard k-d tree algorithm [4] uses a canonical set composed of all axis-orthogonal directions.

DEFINITION 26.7 For a canonical set $\mathcal{C}$ and a canonical region R, we define the following terms (see Figure 26.8a):

- For a canonical direction $\vec{v}_i \in \mathcal{C}$, the *width of R in the direction $\vec{v}_i$*, written $\texttt{width}_i(R)$, is the distance between b_i and c_i, i.e., $\texttt{width}_i(R) = \delta(b_i, c_i)$.
- The *maximum side of* R is $\max(R) = \max_{i\in\mathcal{C}}(\texttt{width}_i(R))$.
- The *minimum side of* R is $\min(R) = \min_{i\in\mathcal{C}}(\texttt{width}_i(R))$.

For simplicity, we also refer to the facets of a region in the same manner. We define the following terms for a facet of a region R, $f = \texttt{facet}_c(R)$:

- The *width of f in the direction $\vec{v}_i$* is $\texttt{width}_i(f) = \delta(b_i, c_i)$ where b_i and c_i are the opposing bounding cuts of f in the direction $\vec{v}_i$.
- The *maximum side of* f is $\max(f) = \max_{i\in\mathcal{C}}(\texttt{width}_i(R))$.
- In addition, for any canonical cut $c \in R$, the *length* of c, $\texttt{len}_c(R)$, is defined as $\max(\texttt{facet}_c(R))$.

When using a canonical cut c_i to partition a region R into two pieces R_1 and R_2 as the cut gets closer to a side of R, one of the two respective regions gets increasingly skinnier.

At some point, the region is no longer α-balanced, see Figure 26.8b. This threshold region is referred to as a shield region and is defined in [11] as the following:

DEFINITION 26.8 Given an α-balanced canonical region R and a canonical cut direction $\vec{v}_i$, sweep a cut c' from the opposing cut b_i toward c_i. Let P be the region of R between c' and c_i. Sweep c' until either region P is empty or just before $\mathtt{asp}(P) > \alpha$. If P is not empty, then P has maximum aspect ratio. Call the region P the *shield region of* c_i *in* R, $\mathtt{shield}_{c_i}(R)$. Let the *maximal outer shield*, $\mathtt{mos}_i(R)$, be the shield region $\mathtt{shield}_{b_i}(R)$ or $\mathtt{shield}_{c_i}(R)$ such that $|\mathtt{mos}_i(R)| = \max(|\mathtt{shield}_{b_i}(R)|, |\mathtt{shield}_{c_i}(R)|)$, i.e., the maximal outer shield is the shield region with the greater number of points.

DEFINITION 26.9 An α-balanced canonical region, R, is *one-cuttable* with reduction factor β, where $1/2 \leq \beta < 1$, if there exists a cut $c^1 \in \mathcal{C}$, called a *one-cut*, dividing R into two subregions R_1 and R_2 such that the following conditions hold:

1. R_1 and R_2 are α-balanced canonical regions,
2. $|R_1| \leq \beta|R|$ and $|R_2| \leq \beta|R|$.

DEFINITION 26.10 An α-balanced canonical region, R, is k*-cuttable* with reduction factor β, for $k > 1$, if there exists a cut $c^k \in \mathcal{C}$, called a k*-cut*, dividing R into two subregions R_1 and R_2 such that the following conditions hold:

1. R_1 and R_2 are α-balanced canonical regions,
2. $|R_2| \leq \beta|R|$,
3. Either $|R_1| \leq \beta|R|$ or R_1 is $(k-1)$-cuttable with reduction factor β.

In other words, the sequence of cuts, $c^k, c^{k-1}, \ldots, c^1$, results in $k+1$ α-balanced canonical regions each containing no more than $\beta|R|$ points. If the reduction factor β is understood, we simply say R is k-cuttable.

DEFINITION 26.11 For a canonical cut set, $\mathcal{C}$, a binary space partition tree T constructed on a set S is a *BAR tree with maximum aspect ratio* α if every region $R \in T$ is α-balanced.

Figure 26.9 illustrates an algorithm to construct a BAR tree from a sequence of k-cuttable regions.

THEOREM 26.4 *For a canonical cut set, $\mathcal{C}$, if every possible α-balanced canonical region is k-cuttable with reduction factor β, then a BAR tree with maximum aspect ratio α can be constructed with depth $O(k \log_{1/\beta} n)$, for any set S of n points in $\mathbb{R}^d$.*

The main challenge in creating a specific instance of a BAR tree is in defining a canonical set $\mathcal{C}$ such that every possible α-balanced canonical region is k-cuttable with reduction factor β for reasonable choices of α, β, and k. The α-balanced regions produced help BAR trees have the following packing function.

```
CreateBarTree($u$,$S_u$,$R_u$,$\alpha$,$\beta$)
    Arguments: Current node $u$ to build (initially the root),
        $S_u$ is the current point set (initially $S$)
        $R_u$ is the $\alpha$-balanced region containing $S_u$
            (initially a bounding hypercube of $S$)
        (Optional) node $u$ can contain any of the following:
            region $R_u$, sample point $p \in S_u$, size $|S_u|$
    if $|S_u| \leq$ leafSize then
        (leaf) node $u$ stores the set $S_u$
        return
    find $c^i$, an $i$-cut for $R_u$, for smallest value of $i$
    (internal) node $u$ stores $c^i$
    create two child nodes of $u$, $v$ and $w$
    partition $S_u$ into $S_v$ and $S_w$ by the cut $s^i$
    partition $R_u$ into $R_v$ and $R_w$ by the cut $s^i$
    call CreateBarTree($v$,$S_v$,$R_v$,$\alpha$,$\beta$)
    call CreateBarTree($w$,$S_w$,$R_w$,$\alpha$,$\beta$)
```

FIGURE 26.9: General BAR tree construction algorithm.

THEOREM 26.5 *For a canonical cut set, $\mathcal{C}$, if every possible α-balanced canonical region is k-cuttable with reduction factor β, then the class of BAR trees with maximum aspect ratio α has a packing function $\rho(n) = O(\alpha^d)$ where the hidden constant factor depends on the angles between the various cut directions. For fixed α, this is constant.*

Theorems 26.4 and 26.5 immediately show us that approximate geometric nearest-neighbor and farthest-neighbor queries can be solved in $O(\epsilon^{1-d} \log n)$ time and approximate geometric range searches for convex and non-convex regions take, respectively, $O(\epsilon^{1-d} \log n)$ and $O(\epsilon^{-d} \log n)$ plus the output size. As with the BBD tree, in fact, these structures can also be shown to have running times of $O(\log n + \epsilon^{1-d} \log \frac{1}{\epsilon})$ for nearest-neighbor and farthest-neighbor queries [11] and $O(\log n + \epsilon^{1-d})$ and $O(\log n + \epsilon^{-d})$ for convex and non-convex range queries [3].

Another examination of Figure 26.6 shows why simple axis-orthogonal cuts cannot guarantee k-cuttability. By concentrating a large number of points at an actual corner of the rectangular region, no sequence of axis-orthogonal cuts will divide the points and maintain balanced aspect ratio regions. We can further extend this notion of a bad corner to a general κ-corner associated with a canonical region R.

DEFINITION 26.12 For a canonical cut set $\mathcal{C}$ and a canonical region R, a *κ-corner* $B \in R$ is a ball with center ρ and radius κ such that, for every cut direction $\vec{v}_i \in \mathcal{C}$ with bounding cuts b_i and c_i, either b_i or c_i intersects B, i.e. $\min(\delta(\rho, b_i), \delta(\rho, c_i)) \leq \kappa$.

When $\kappa = 0$, we are not only defining a vertex of a region but a vertex which is tangential to one of every cut direction's bounding planes. As described in [11], these corners represent the worst-case placement of points in the region. These corners can always exist in regions. However, if one of the facets associated with this corner has size proportional to (or smaller than) the κ ball, then we can still get close enough to this facet and properly divide the point set without introducing unbalanced regions. The following property formalizes this notion more:

Property 26.13 *A canonical cut set $\mathcal{C}$ satisfies the κ-*Corner Property *if for any $\kappa \geq 0$ and any canonical region R containing a κ-corner $B \in R$, there exists a canonical cut $c \in R$*

intersecting B such that $\texttt{len}_c(R) \leq \mathcal{F}_\kappa \kappa$ for some constant $\mathcal{F}_\kappa$.

In particular, notice that if $\kappa = 0$, one of the bounding cutting planes must intersect at a single point. The advantage to this can be seen in the two-dimensional case. Construct any canonical region using any three cutting directions, for simplicity use the two axis-orthogonal cuts and one cut with slope $+1$. It is impossible to find a κ-corner without having at least one of the three bounding sides be small with respect to the corner. This small side has a matching very small shield region. Unfortunately, having a small shield region does not mean that the initial region is one-cuttable. The points may all still be concentrated within this small shield region. However, it is possible that this small shield region is one-cuttable. In fact, in [11], it is shown that there exist canonical cut sets that guarantee *two-cuttability* for sufficient values of α, β, and σ, where the σ parameter is used in the construction. The sufficiency requirements depend only on certain constant properties associated with the angles of the canonical cut set labeled here as $\mathcal{F}_{\min}, \mathcal{F}_{\max}, \mathcal{F}_{\texttt{box}}$, and $\mathcal{F}_\kappa$. For specific values of these constants, see [11]. Figure 26.10 describes an algorithm to find an appropriate cut.

The general idea is to find the smallest (in physical size) shield region containing a majority of the points. If none exist, the region must be one-cuttable. Otherwise, we take this shield region and pull the partitioning cut back slightly,[†] increasing the size of this particular shield region. Given an appropriate cut set and various constant bounds, we can guarantee that this new region is one-cuttable. The following theorems summarize this result [11]:

THEOREM 26.6 **(Two-Cuttable Theorem)** *Suppose we are given a canonical cut set, $\mathcal{C}$, which satisfies the κ-Corner Property 26.13. Any α-balanced canonical region R is two-cuttable if the following three conditions are met:*

$$\beta \geq (d+1)/(d+2), \tag{26.1}$$

$$\alpha \mathcal{F}_{\min}/4(\mathcal{F}_{\texttt{box}}+1) > \sigma > (2\mathcal{F}_{\max} + \mathcal{F}_\kappa), \text{ and} \tag{26.2}$$

$$\alpha > 4(\mathcal{F}_{\texttt{box}}+1)(2\mathcal{F}_{\max}+\mathcal{F}_\kappa)/\mathcal{F}_{\min} + \mathcal{F}_{\max}/\mathcal{F}_{\min}. \tag{26.3}$$

Theorems 26.6 and 26.4 can be combined to yield the following theorem:

THEOREM 26.7 *Suppose we are given a canonical cut set $\mathcal{C}$ that satisfies the κ-Corner Property and an $\alpha > f(\mathcal{C})$. A BAR tree with depth $O(d \log n)$ and balancing factor α can be constructed in $O(g(\mathcal{C}) dn \log n)$ time, where f and g are constant functions depending on properties of the canonical set. In particular, the running time of the algorithm is $O(n \log n)$ for fixed dimensions and fixed canonical sets.*

Let us now present two cut sets that do satisfy the κ-Corner Property. The two cut sets we present below are composed of axis-orthogonal cuts and one other set of cuts. Let us give specific names to a few vector directions.

DEFINITION 26.14 A vector $v = (x_0, x_1, x_2, \ldots, x_d)$ is

[†]This is actually only necessary in dimensions greater than 2.

```
ComputeTwoCut($u$)
    Arguments: An $\alpha$-balanced node $u$ in a BAR tree
    Returns: A one or two-cut for $u$
    for all $c_i \in \mathcal{C}$
        if $c_i$ is a one-cut, return $c_i$
    let $P$ be the smallest maximal outer shield of $R$
    let $c = c_i$ be the bounding cut associated with $P$
    let $c'$ be the cut parallel to $c$ intersecting $R$ such that
        $\delta(c, c') = \texttt{width}_i(P) + \texttt{len}_c(R)/\sigma$
    return $c'$
    // $c'$ partitions $R$ into two $\alpha$-balanced regions $R_1$ and $R_2$
    // $|R_2| \leq \beta|R|$
    // $R_1$ incident to $c_i$ is one-cuttable
```

FIGURE 26.10: An algorithm to find either a one or two cut in a region.

- an *axis-orthogonal cut* if $x_i = 0$ for all values except one where $x_j = 1$, e.g. $(0, 0, 1, 0)$,
- a *corner cut* if $x_i = \pm 1$ for all values of i, e.g. $(1, 1, -1, -1)$,
- a *wedge cut* if $x_i = 0$ for all values except two where $x_j, x_i = \pm 1$, e.g. $(0, 1, -1, 0)$.

The *Corner Cut Canonical Set* $\mathcal{C}_c$ is the set of all axis-orthogonal cuts and corner cuts. The *Wedge Cut Canonical Set* $\mathcal{C}_w$ is the set of all axis-orthogonal cuts and wedge cuts.

Notice that $|\mathcal{C}_c|$ is $\Theta(2^d)$ and $|\mathcal{C}_w|$ is $\Theta(d^2)$. Although the corner cut canonical set does not necessarily have to be as large as this, the complexity of the corner cut itself means sidedness tests take longer than axis-orthogonal and wedge cuts, namely d computations instead of 1 or 2. The above two canonical sets satisfy the κ-Corner Property 26.13 and from Theorem 26.7, we get the following two corollaries [11]:

COROLLARY 26.1 For the Corner Cut Canonical set $\mathcal{C}_c$, a BAR tree with depth $O(d \log n)$ and balancing factor $\alpha = \Omega(d^2)$ can be constructed in $O(n \log n)$ time.

COROLLARY 26.2 For the Wedge Cut Canonical set $\mathcal{C}_w$, a BAR tree with depth $O(d \log n)$ and balancing factor $\alpha = \Omega(\sqrt{d})$ can be constructed in $O(n \log n)$ time.

To get the exact values needed, see [11]. However, it is important to note that the α bounds above are overestimates of the minimum value needed. In practice, one should try an initially small value of α, say 6, and when that fails to provide two-cuttability double the value for the lower subtree levels. In this manner, one can arrive at the true minimum value in $O(\log \alpha)$ such iterations, if necessary, without having to calculate it. Since the minimum α needed in both cut sets is $O(d^2)$, this adds only an $O(\log(d))$ factor to the depth.

26.7 Maximum-Spread k-d Trees

One very popular class of BSP tree is the k-d tree, see Chapter 16. Although there are very few theoretical bounds known on these structures, there is a lot of empirical evidence that shows them to be extremely efficient for numerous geometric applications. In particular, one variant the maximum-spread k-d tree has long been considered an ideal k-d tree. Given a set of points S and a particular axis dimension x_d, define the *spread* of S in x_d to be the difference between the minimum and maximum coordinates of the points in that dimension. The maximum-spread k-d tree is formed by choosing at each internal node a cutting plane orthogonal to the axis of maximum spread placed at the median point in this direction, see for example [16]. Arya *et al.* [1] applied the maximum-spread k-d tree to their approximate nearest-neighbor searching algorithm and experimentally showed that they were comparable to the theoretically efficient BBD tree. Later Dickerson *et al.* [9, 11] proved the following theorem regarding maximum-spread k-d trees, referred to there as longest-side k-d trees:

THEOREM 26.8 *Suppose we are given a maximum-spread k-d tree T constructed on a set S of n points in $\mathbb{R}^d$. Then the packing function $\rho(n)$ of T for a region annulus A is $O(\log^{d-1} n)$. That is, the class of maximum-spread k-d trees is an $O(\log^{d-1} n)$-quasi-BAR tree.*

Although the bound is not as good as for BBD trees and BAR trees, the simplicity of the structure yields low constant factors and explains why in practice these trees perform so well. Experimental comparisons to BBD trees and BAR trees verified this result and showed that only for very highly clustered data did the dependency on $\log^{d-1} n$ become prominent [1, 11]. In practice, unless data is highly clustered and the dimension is moderately large, the maximal-spread k-d tree is an ideal structure to use. However, for such data sets both the BBD tree and the BAR tree revert to the same behavior as the maximal-spread tree, and they perform well even with highly clustered data. Because of its simpler structure, the BBD tree is potentially more practical than the BAR tree.

Acknowledgment

This work was supported, in part, by the National Science Foundation under grant CCR-0098068.

References

[1] Arya, Mount, Netanyahu, Silverman, and Wu. An optimal algorithm for approximate nearest neighbor searching in fixed dimensions. *J. ACM*, 45(6):891–923, 1998.

[2] S. Arya and D. M. Mount. Approximate nearest neighbor queries in fixed dimensions. In *Proc. 4th ACM-SIAM Sympos. Discrete Algorithms*, pages 271–280, 1993.

[3] Sunil Arya and David M. Mount. Approximate range searching. *Comput. Geom.*, 17(3-4):135–152, 2000.

[4] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, 1975.

[5] P. B. Callahan and S. R. Kosaraju. A decomposition of multidimensional point sets with applications to k-nearest-neighbors and n-body potential fields. *J. ACM*, 42:67–90, 1995.

[6] Bernard Chazelle. Lower bounds on the complexity of polytope range searching. *J. Amer. Math. Soc.*, 2:637–666, 1989.

[7] Bernard Chazelle and Emo Welzl. Quasi-optimal range searching in spaces of finite VC-dimension. *Discrete Comput. Geom.*, 4:467–489, 1989.

[8] K. L. Clarkson. Fast algorithms for the all nearest neighbors problem. In *Proc. 24th Annu. IEEE Sympos. Found. Comput. Sci.*, pages 226–232, 1983.

[9] M. Dickerson, C. A. Duncan, and M. T. Goodrich. K-D trees are better when cut on

the longest side. In *ESA: Annual European Symposium on Algorithms*, volume 1879 of *Lecture Notes Comput. Sci.*, pages 179–190, 2000.

[10] J. R. Driscoll, H. N. Gabow, R. Shrairaman, and R. E. Tarjan. Relaxed heaps: An alternative to Fibonacci heaps with applications to parallel computation. *Commun. ACM*, 31:1343–1354, 1988.

[11] C. A. Duncan. *Balanced Aspect Ratio Trees.* Ph.D. thesis, Dept. of Computer Science, Johns Hopkins Univ., 1999.

[12] C. A. Duncan, M. T. Goodrich, and S. G. Kobourov. Balanced aspect ratio trees and their use for drawing very large graphs. *J. Graph Algorithms and Applications*, 4:19–46, 2000.

[13] C. A. Duncan, M. T. Goodrich, and S. G. Kobourov. Balanced aspect ratio trees: Combining the advantages of k-d trees and octrees. *J. Algorithms*, 38:303–333, 2001.

[14] R. A. Finkel and J. L. Bentley. Quad trees: a data structure for retrieval on composite keys. *Acta Inform.*, 4:1–9, 1974.

[15] M. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization problems. *J. ACM*, 34:596–615, 1987.

[16] J. H. Friedman, J. L. Bentley, and R. A. Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Trans. Math. Softw.*, 3:209–226, 1977.

[17] F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction.* Springer-Verlag, New York, NY, 1985.

[18] H. Samet. *The Design and Analysis of Spatial Data Structures.* Addison-Wesley, Reading, MA, 1990.

27

Geometric and Spatial Data Structures in External Memory

Jeffrey Scott Vitter
Purdue University

27.1 Introduction

The *Input/Output* communication (or simply *I/O*) between the fast internal memory and the slow external memory (such as disk) can be a bottleneck when processing massive amounts of data, as is the case in many spatial and geometric applications [124]. Problems involving massive amounts of geometric data are ubiquitous in spatial databases [82, 106, 107], geographic information systems (GIS) [82, 106, 119], constraint logic programming [73, 74], object-oriented databases [130], statistics, virtual reality systems, and computer graphics [55]. NASA's Earth Observing System project, the core part of the Earth Science Enterprise (formerly Mission to Planet Earth), produces petabytes (10^{15} bytes) of raster data per year [53]. Microsoft's TerraServer online database of satellite images is over one terabyte in size [115]. A major challenge is to develop mechanisms for processing the data, or else much of the data will be useless.

One promising approach for efficient I/O is to design algorithms and data structures that bypass the virtual memory system and explicitly manage their own I/O. We refer to such algorithms and data structures as *external memory* (or *EM*) *algorithms and data structures*. (The terms *out-of-core algorithms* and *I/O algorithms* are also sometimes used.)

We concentrate in this chapter on the design and analysis of EM memory data structures for batched and online problems involving geometric and spatial data. Luckily, many problems on geometric objects can be reduced to a small core of problems, such as computing intersections, convex hulls, multidimensional search, range search, stabbing queries, point location, and nearest neighbor search. In this chapter we discuss useful paradigms for solving these problems in external memory.

27.1.1 Disk Model

The three primary measures of performance of an algorithm or data structure are *the number of I/O operations performed, the amount of disk space used, and the internal (parallel) computation time.* For reasons of brevity we shall focus in this chapter on only the first two measures. Most of the algorithms we mention run in optimal CPU time, at least for the single-processor case.

We can capture the main properties of magnetic disks and multiple disk systems by the commonly used *parallel disk model* (PDM) introduced by Vitter and Shriver [125]. Data is transferred in large units of *blocks* of size B so as to amortize the latency of moving the read-write head and waiting for the disk to spin into position. Storage systems such as RAID use multiple disks to get more bandwidth [38, 66]. The principal parameters of PDM are the following:

$$\begin{aligned} N &= \text{problem input data size (in terms of items)},\\ M &= \text{size of internal memory (in terms of items)},\\ B &= \text{size of disk block (in terms of items), and}\\ D &= \text{\# independent disks}, \end{aligned}$$

where $M < N$ and $1 \le DB \le M$.

Queries are naturally associated with online computations, but they can also be done in batched mode. For example, in the batched orthogonal 2-D range searching problem discussed in Section 27.2, we are given a set of N points in the plane and a set of Q queries in the form of rectangles, and the problem is to report the points lying in each of the Q query rectangles. In both the batched and online settings, the number of items reported in response to each query may vary. We thus define two more performance parameters:

$$\begin{aligned} Q &= \text{number of input queries (for a batched problem), and}\\ Z &= \text{query output size (in terms of items)}. \end{aligned}$$

If the problem does not involve queries, we set $Q = 0$.

It is convenient to refer to some of the above PDM parameters in units of disk blocks rather than in units of data items; the resulting formulas are often simplified. We define the lowercase notation

$$n = \frac{N}{B}, \qquad m = \frac{M}{B}, \qquad z = \frac{Z}{B}, \qquad q = \frac{Q}{B},$$

to be the problem input size, internal memory size, query output size, and number of queries, respectively, in units of disk blocks.

For simplicity, we restrict our attention in this chapter to the single-disk case $D = 1$. The batched algorithms we discuss can generally be sped up by using multiple disks in an optimal

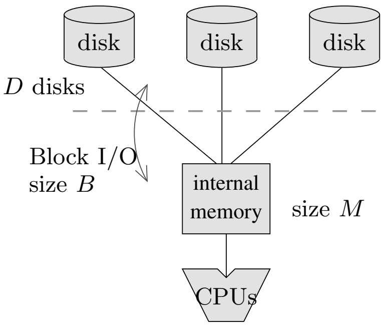


FIGURE 27.1: Parallel Disk Model.

manner using the load balancing techniques discussed in [124]. Online data structures that use a single disk can generally be transformed automatically by the technique of disk striping to make optimal use of multiple disks [124].

Programs that perform well in terms of PDM will generally perform well when implemented on real systems [124]. More complex and precise models have been formulated [22, 103, 111]. Hierarchical (multilevel) memory models are discussed in [124] and its references. Cache-oblivious models are discussed in Chapter 34. Many of the algorithm design techniques we discuss in this chapter, which exploit data locality so as to minimize I/O communication steps, form the basis for algorithms in the other models.

27.1.2 Design Criteria for External Memory Data Structures

The data structures we examine in this chapter are used in batched algorithms and in online settings. In batched problems, no preprocessing is done, and the entire file of data items must be processed, often by streaming the data through the internal memory in one or more passes. Queries are done in a batched manner during the processing. The goal is generally twofold:

B1. to solve the problem in $O\big((n+q)\log_m n + z\big)$ I/Os, and

B2. to use only a linear number $O(n+q)$ of blocks of disk storage.

Most nontrivial problems require the same number of I/Os as does sorting. In particular, criterion B1 is related to the I/O complexity of sorting N items in the PDM model, which is $O(n \log_m n)$ [124].

Online data structures support query operations in a continuous manner. When the data items do not change and the data structure can be preprocessed before any queries are made, the data structure is known as *static*. When the data structure supports insertions and deletions of items, intermixed with the queries, the data structure is called *dynamic*. The primary theoretical challenges in the design and analysis of online EM data structures are threefold:

O1. to answer queries in $O(\log_B N + z)$ I/Os,

O2. to use only a linear number $O(n)$ of blocks of disk storage, and

O3. to do updates (in the case of dynamic data structures) in $O(\log_B N)$ I/Os.

Criteria O1–O3 correspond to the natural lower bounds for online search in the comparison model. The three criteria are problem-dependent, and for some problems they cannot be met. For dictionary queries, we can do better using hashing, achieving $O(1)$ I/Os per query on the average.

Criterion O1 combines together the $O(\log_B N)$ I/O cost for the search with the $O(\lceil z \rceil)$ I/O cost for reporting the output. When one cost is much larger than the other, the query algorithm has the extra freedom to follow a *filtering* paradigm [35], in which both the search component and the output reporting are allowed to use the larger number of I/Os. For example, when the output size Z is large, the search component can afford to be somewhat sloppy as long as it doesn't use more than $O(z)$ I/Os; when Z is small, the Z output items do not have to reside compactly in only $O(\lceil z \rceil)$ blocks. Filtering is an important design paradigm in online EM data structures.

For many of the batched and online problems we consider, there is a data structure (such as a scanline structure or binary search tree) for the internal memory version of the problem that can answer each query in $O(\log N + Z)$ CPU time, but if we use the same data structure naively in an external memory setting (using virtual memory to handle page management), a query may require $\Omega(\log N + Z)$ I/Os, which is excessive.*

The goal for online algorithms is to build locality directly into the data structure and explicitly manage I/O so that the $\log N$ and Z terms in the I/O bounds of the naive approach are replaced by $\log_B N$ and z, respectively. The relative speedup in I/O performance, namely $(\log N + Z)/(\log_B N + z)$, is at least $(\log N)/\log_B N = \log B$, which is significant in practice, and it can be as much as $Z/z = B$ for large Z.

For batched problems, the I/O performance can be improved further, since the answers to the queries do not need to be provided immediately but can be reported as a group at the end of the computation. For the batched problems we consider in this chapter, the $Q = qB$ queries collectively use $O(q \log_m n + z)$ I/Os, which is about B times less than a naive approach.

27.1.3 Overview of Chapter

In the next section, we discuss batched versions of geometric search problems. One of the primary methods used for batched geometric problems is distribution sweeping, which uses a data structure reminiscent of the distribution paradigm in external sorting. Other useful batched techniques include persistent B-trees, batched filtering, external fractional cascading, external marriage-before-conquest, and batched incremental construction.

The most popular EM online data structure is the B-tree structure, which provides excellent performance for dictionary operations and one-dimensional range searching. We give several variants and applications of B-trees in Section 27.3. We look at several aspects of multidimensional range search in Section 27.4 and related problems such as stabbing queries and point location. Data structures for other variants and related problems such as nearest neighbor search are discussed in Section 27.5. Dynamic and kinetic data structures are discussed in Section 27.6. A more comprehensive survey of external memory data structures appears in [124].

27.2 EM Algorithms for Batched Geometric Problems

Advances in recent years have allowed us to solve a variety of batched geometric problems optimally, meeting both optimality Criteria B1 and B2 of Section 27.1.2. These problems include

*We use the notation $\log N$ to denote the binary (base 2) logarithm $\log_2 N$. For bases other than 2, the base will be specified explicitly, as in the base-B logarithm $\log_B N$.

1. Computing the pairwise intersections of N segments in the plane and their trapezoidal decomposition,
2. Finding all intersections between N nonintersecting red line segments and N nonintersecting blue line segments in the plane,
3. Answering Q orthogonal 2-D range queries on N points in the plane (i.e., finding all the points within the Q query rectangles),
4. Constructing the 2-D and 3-D convex hull of N points,
5. Voronoi diagram and Triangulation of N points in the plane,
6. Performing Q point location queries in a planar subdivision of size N,
7. Finding all nearest neighbors for a set of N points in the plane,
8. Finding the pairwise intersections of N orthogonal rectangles in the plane,
9. Computing the measure of the union of N orthogonal rectangles in the plane,
10. Computing the visibility of N segments in the plane from a point, and
11. Performing Q ray-shooting queries in 2-D Constructive Solid Geometry (CSG) models of size N.

Goodrich et al. [59], Zhu [131], Arge et al. [18], Arge et al. [14], and Crauser et al. [44, 45] develop EM algorithms for those problems using these EM paradigms for batched problems:

Distribution sweeping, a generalization of the sorting distribution paradigm [124] for "externalizing" plane sweep algorithms.

Persistent B-trees, an offline method for constructing an optimal-space persistent version of the B-tree data structure (see Section 27.3.1), yielding a factor of B improvement over the generic persistence techniques of Driscoll et al. [49].

Batched filtering, a general method for performing simultaneous EM searches in data structures that can be modeled as planar layered directed acyclic graphs; it is useful for 3-D convex hulls and batched point location. Multisearch on parallel computers is considered in [48].

External fractional cascading, an EM analogue to fractional cascading on a segment tree, in which the degree of the segment tree is $O(m^\alpha)$ for some constant $0 < \alpha \leq 1$. Batched queries can be performed efficiently using batched filtering; online queries can be supported efficiently by adapting the parallel algorithms of work of Tamassia and Vitter [114] to the I/O setting.

External marriage-before-conquest, an EM analogue to the technique of Kirkpatrick and Seidel [76] for performing output-sensitive convex hull constructions.

Batched incremental construction, a localized version of the randomized incremental construction paradigm of Clarkson and Shor [42], in which the updates to a simple dynamic data structure are done in a random order, with the goal of fast overall performance on the average. The data structure itself may have bad worst-case performance, but the randomization of the update order makes worst-case behavior unlikely. The key for the EM version so as to gain the factor of B I/O speedup is to batch together the incremental modifications.

For illustrative purposes, we focus in the remainder of this section primarily on the distribution sweep paradigm [59], which is a combination of the distribution paradigm for sorting [124] and the well-known sweeping paradigm from computational geometry [47, 98]. As an example, let us consider how to achieve optimality Criteria B1 and B2 for computing the pairwise intersections of N orthogonal segments in the plane, making use of the following recursive distribution sweep: At each level of recursion, the region under consideration is

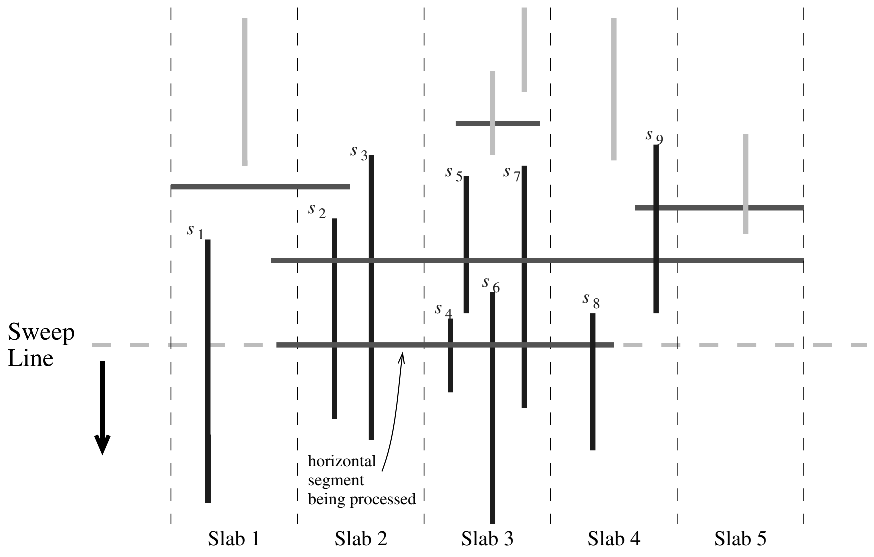


FIGURE 27.2: Distribution sweep used for finding intersections among N orthogonal segments. The vertical segments currently stored in the slabs are indicated in bold (namely s_1, s_2, ..., s_9). Segments s_5 and s_9 are not active, but have not yet been deleted from the slabs. The sweep line has just advanced to a new horizontal segment that completely spans slabs 2 and 3, so slabs 2 and 3 are scanned and all the active vertical segments in slabs 2 and 3 (namely s_2, s_3, s_4, s_6, s_7) are reported as intersecting the horizontal segment. In the process of scanning slab 3, segment s_5 is discovered to be no longer active and can be deleted from slab 3. The end portions of the horizontal segment that "stick out" into slabs 1 and 4 are handled by the lower levels of recursion, where the intersection with s_8 is eventually discovered.

partitioned into $\Theta(m)$ vertical *slabs*, each containing $\Theta(N/m)$ of the segments' endpoints. We sweep a horizontal line from top to bottom to process the N segments. When the sweep line encounters a vertical segment, we insert the segment into the appropriate slab. When the sweep line encounters a horizontal segment h, as pictured in Figure 27.2, we report h's intersections with all the "active" vertical segments in the slabs that are spanned *completely* by h. (A vertical segment is "active" if it intersects the current sweep line; vertical segments that are found to be no longer active are deleted from the slabs.) The remaining two end portions of h (which "stick out" past a slab boundary) are passed recursively to the next level, along with the vertical segments. The downward sweep then proceeds. After the initial sorting (to get the segments with respect to the y-dimension), the sweep at each of the $O(\log_m n)$ levels of recursion requires $O(n)$ I/Os, yielding the desired bound in B1 of $O\big((n+q)\log_m n + z\big)$. Some timing experiments on distribution sweeping appear in [39]. Arge et al. [14] develop a unified approach to distribution sweep in higher dimensions.

A central operation in spatial databases is spatial join. A common preprocessing step is to find the pairwise intersections of the bounding boxes of the objects involved in the spatial join. The problem of intersecting orthogonal rectangles can be solved by combining the previous sweep line algorithm for orthogonal segments with one for range searching. Arge et al. [14] take a more unified approach using distribution sweep, which is extendible to higher dimensions: The active objects that are stored in the data structure in this case are rectangles, not vertical segments. The authors choose the branching factor to be $\Theta(\sqrt{m}\,)$.

Each rectangle is associated with the largest contiguous range of vertical slabs that it spans. Each of the possible $\Theta\big(\binom{\sqrt{m}}{2}\big) = \Theta(m)$ contiguous ranges of slabs is called a *multislab*. The reason why the authors choose the branching factor to be $\Theta(\sqrt{m})$ rather than $\Theta(m)$ is so that the number of multislabs is $\Theta(m)$, and thus there is room in internal memory for a buffer for each multislab. The height of the tree remains $O(\log_m n)$.

The algorithm proceeds by sweeping a horizontal line from top to bottom to process the N rectangles. When the sweep line first encounters a rectangle R, we consider the multislab lists for all the multislabs that R intersects. We report all the active rectangles in those multislab lists, since they are guaranteed to intersect R. (Rectangles no longer active are discarded from the lists.) We then extract the left and right end portions of R that partially "stick out" past slab boundaries, and we pass them down to process in the next lower level of recursion. We insert the remaining portion of R, which spans complete slabs, into the list for the appropriate multislab. The downward sweep then continues. After the initial sorting preprocessing, each of the $O(\log_m n)$ sweeps (one per level of recursion) takes $O(n)$ I/Os, yielding the desired bound $O\big((n+q)\log_m n + z\big)$.

The resulting algorithm, called Scalable Sweeping-Based Spatial Join (SSSJ) [13, 14], outperforms other techniques for rectangle intersection. It was tested against two other sweep line algorithms: the Partition-Based Spatial-Merge (QPBSM) used in Paradise [97] and a faster version called MPBSM that uses an improved dynamic data structure for intervals [13]. The TPIE (Transparent Parallel I/O Environment) system [11, 117, 122] served as the common implementation platform. The algorithms were tested on several data sets. The timing results for the two data sets in Figures 27.3(a) and 27.3(b) are given in Figures 27.3(c) and 27.3(d), respectively. The first data set is the worst case for sweep line algorithms; a large fraction of the line segments in the file are active (i.e., they intersect the current sweep line). The second data set is a best case for sweep line algorithms, but the two PBSM algorithms have the disadvantage of making extra copies of the rectangles. In both cases, SSSJ shows considerable improvement over the PBSM-based methods. In other experiments done on more typical data, such as TIGER/line road data sets [116], SSSJ and MPBSM perform about 30% faster than does QPBSM. The conclusion we draw is that SSSJ is as fast as other known methods on typical data, but unlike other methods, it scales well even for worst-case data. If the rectangles are already stored in an index structure, such as the R-tree index structure we consider in Section 27.4.2, hybrid methods that combine distribution sweep with inorder traversal often perform best [12].

For the problem of finding all intersections among N line segments, Arge et al. [18] give an efficient algorithm based upon distribution sort, but the output component of the I/O bound is slightly nonoptimal: $z \log_m n$ rather than z. Crauser et al. [44, 45] attain the optimal I/O bound of criterion B1, namely $O\big((n+q)\log_m n + z\big)$, by constructing the trapezoidal decomposition for the intersecting segments using an incremental randomized construction. For I/O efficiency, they do the incremental updates in a series of batches, in which the batch size is geometrically increasing by a factor of m.

Online issues also arise in the analysis of batched EM algorithms: In practice, batched algorithms must adapt in a robust and online way when the memory allocation changes, and online techniques can play an important role. Some initial work has been done on memory-adaptive EM algorithms in a competitive framework [23].

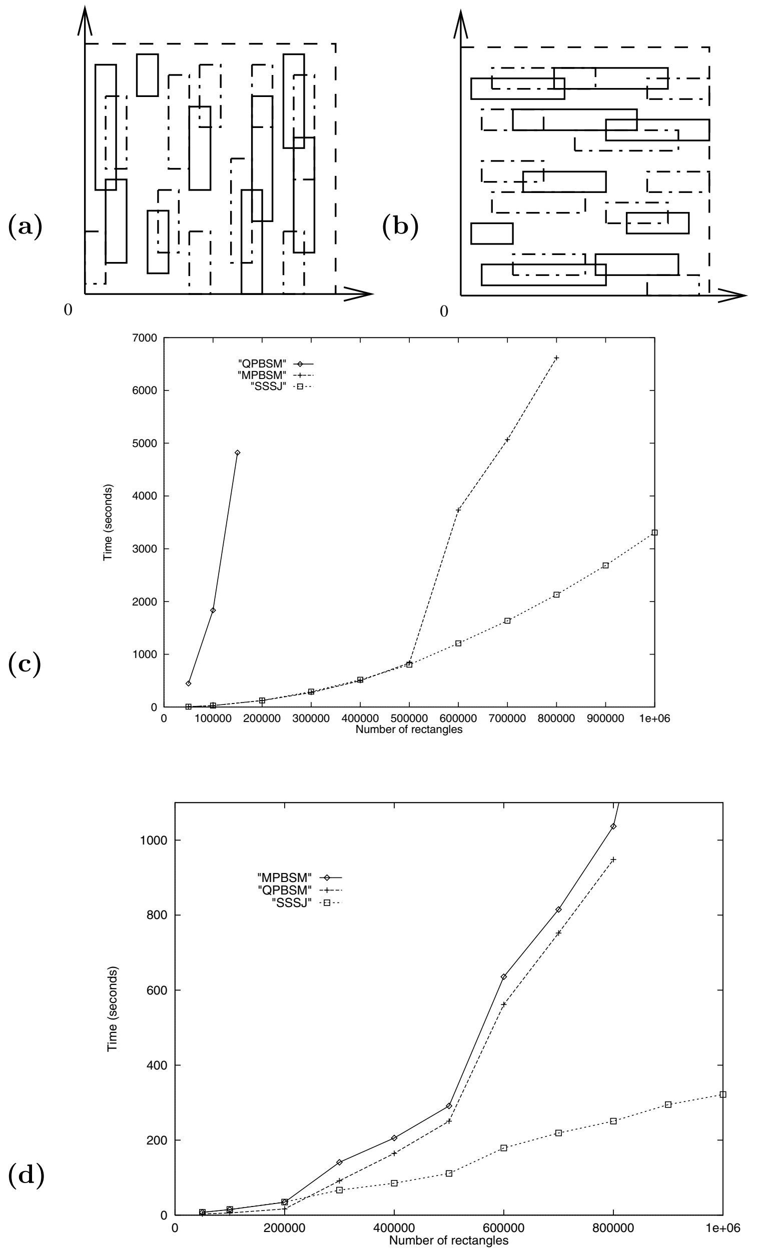


FIGURE 27.3: Comparison of Scalable Sweeping-Based Spatial Join (SSSJ) with the original PBSM (QPBSM) and a new variant (MPBSM). In this variant of the problem, each data set contains $N/2$ red rectangles (designated by solid sides) and $N/2$ blue rectangles (designated by dashed sides), and the goal is to find all intersections between red rectangles and blue rectangles. In each data set shown, the number of intersections is $O(N)$: (a) Data set 1 consists of tall, skinny (vertically aligned) rectangles; (b) Data set 2 consists of short, wide (horizontally aligned) rectangles; (c) Running times on data set 1; (d) Running times on data set 2.

27.3 External Memory Tree Data Structures

In this section we consider the basic online EM data structures for storing and querying spatial data in one dimension. The dictionary problem is an important special case, which can be solved efficiently in the average case by use of hashing. However, hashing does not support sequential search in a natural way, such as retrieving all the items with key value in a specified range. Some clever work has been done on order-preserving hash functions, in which items with sequential keys are stored in the same block or in adjacent blocks, but the search performance is less robust and tends to deteriorate because of unwanted collisions. (See [56, 124] for a survey.)

A more effective EM approach for geometric queries is to use multiway trees, which we explore in this section. For illustration, we use orthogonal range search as our canonical problem. It is a fundamental database primitive in spatial databases and geographic information systems (GIS), and it includes dictionary lookup as a special case. A range query, for a given d-dimensional rectangle, returns all the points in the interior of the rectangle. Other types of spatial queries include point location queries, ray shooting queries, nearest neighbor queries, and intersection queries, but for brevity we restrict our attention primarily to range searching.

Spatial data structures tend to be of two types: space-driven or data-driven. Quad trees and grid files are space-driven since they are based upon a partitioning of the embedding space, somewhat akin to using order-preserving hash functions, whereas methods like R-trees and kd-trees are organized by partitioning the data items themselves. We shall discuss primarily the latter type in this chapter.

27.3.1 B-trees and Variants

Tree-based data structures arise naturally in the online setting, in which the data can be updated and queries must be processed immediately. Binary trees have a host of applications in the (internal memory) RAM model. In order to exploit block transfer, trees in external memory generally use a block for each node, which can store $\Theta(B)$ pointers and data values.

The well-known balanced multiway *B-tree* due to Bayer and McCreight [25, 43, 77] is the most widely used nontrivial EM data structure. The degree of each node in the B-tree (with the exception of the root) is required to be $\Theta(B)$, which guarantees that the height of a B-tree storing N items is roughly $\log_B N$. B-trees support dynamic dictionary operations and one-dimensional range search optimally in linear space, $O(\log_B N)$ I/Os per insert or delete, and $O(\log_B N + z)$ I/Os per query, where $Z = zB$ is the number of items output. When a node overflows during an insertion, it splits into two half-full nodes, and if the splitting causes the parent node to overflow, the parent node splits, and so on. Splittings can thus propagate up to the root, which is how the tree grows in height. Deletions are handled in a symmetric way by merging nodes.

In the B^+*-tree* variant, pictured in Figure 27.4, all the items are stored in the leaves, and the leaves are linked together in symmetric order to facilitate range queries and sequential access. The internal nodes store only key values and pointers and thus can have a higher branching factor. In the most popular variant of B$^+$-trees, called *B*-trees*, splitting can usually be postponed when a node overflows by "sharing" the node's data with one of its adjacent siblings. The node needs to be split only if the sibling is also full; when that happens, the node splits into two, and its data and those of its full sibling are evenly redistributed, making each of the three nodes about two-thirds full. This local optimization reduces the number of times new nodes must be created and thus increases the storage utilization. And since there are fewer nodes in the tree, search I/O costs are lower. When no sharing is done (as in B$^+$-trees), Yao [129] shows that nodes are roughly $\ln 2 \approx 69\%$ full on the average, assuming random insertions. With sharing (as in B*-trees), the average storage utilization increases to about $2\ln(3/2) \approx 81\%$ [20, 81]. Storage utilization can be increased further by sharing among several siblings, at the cost of more complicated insertions and deletions. Some helpful space-saving techniques borrowed from hashing are partial expansions [21] and use of overflow nodes [112].

A cross between B-trees and hashing, where each subtree rooted at a certain level of the B-tree is instead organized as an external hash table, was developed by Litwin and Lomet [84] and further studied in [19, 85]. O'Neil [93] proposed a B-tree variant called the SB-tree that clusters together on the disk symmetrically ordered nodes from the same level so as to optimize range queries and sequential access. Rao and Ross [100, 101] use similar ideas to exploit locality and optimize search tree performance in internal memory. Reducing the number of pointers allows a higher branching factor and thus faster search.

Partially persistent versions of B-trees have been developed by Becker et al. [26] and Varman and Verma [120]. By persistent data structure, we mean that searches can be done with respect to any timestamp y [49, 50]. In a partially persistent data structure, only the most recent version of the data structure can be updated. In a fully persistent data structure, any update done with timestamp y affects all future queries for any time after y. An interesting open problem is whether B-trees can be made fully persistent. Salzberg and Tsotras [105] survey work done on persistent access methods and other techniques for time-evolving data. Lehman and Yao [83], Mohan [89], and Lomet and Salzberg [87] explore mechanisms to add concurrency and recovery to B-trees. Other variants are discussed in Chapter 15.

27.3.2 Weight-Balanced B-trees

Arge and Vitter [17] introduce a powerful variant of B-trees called *weight-balanced B-trees*, with the property that the weight of any subtree at level h (i.e., the number of nodes in the subtree rooted at a node of height h) is $\Theta(a^h)$, for some fixed parameter a of order B. By contrast, the sizes of subtrees at level h in a regular B-tree can differ by a multiplicative factor that is exponential in h. When a node on level h of a weight-balanced B-tree gets rebalanced, no further rebalancing is needed until its subtree is updated $\Omega(a^h)$ times. Weight-balanced B-trees support a wide array of applications in which the I/O cost to rebalance a node of weight w is $O(w)$; the rebalancings can be scheduled in an amortized (and often worst-case) way with only $O(1)$ I/Os. Such applications are very common when the

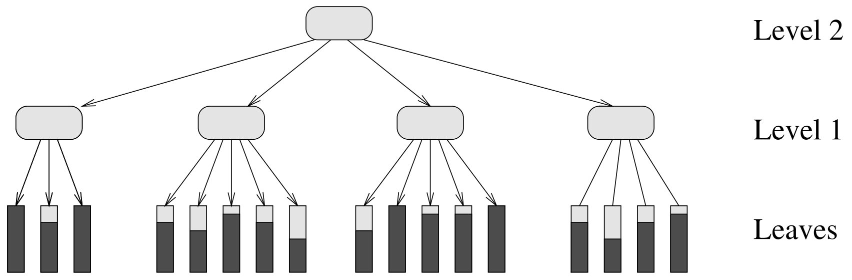


FIGURE 27.4: B$^+$-tree multiway search tree. Each internal and leaf node corresponds to a disk block. All the items are stored in the leaves; the darker portion of each leaf block indicates its relative fullness. The internal nodes store only key values and pointers, $\Theta(B)$ of them per node. Although not indicated here, the leaf blocks are linked sequentially.

nodes have secondary structures, as in multidimensional search trees, or when rebuilding is expensive. Agarwal et al. [6] apply weight-balanced B-trees to convert partition trees such as kd-trees, BBD trees, and BAR trees, which were designed for internal memory, into efficient EM data structures.

Weight-balanced trees called BB[α]-trees [31, 91] have been designed for internal memory; they maintain balance via rotations, which is appropriate for binary trees, but not for the multiway trees needed for external memory. In contrast, weight-balanced B-trees maintain balance via splits and merges.

Weight-balanced B-trees were originally conceived as part of an optimal dynamic EM interval tree structure for stabbing queries and a related EM segment tree structure. We discuss their use for stabbing queries and other types of range queries in Sections 27.4.3–27.4.5. They also have applications in the (internal memory) RAM model [17, 63], where they offer a simpler alternative to BB[α]-trees. For example, by setting a to a constant in the EM interval tree based upon weight-balanced B-trees, we get a simple worst-case implementation of interval trees [51, 52] in internal memory. Weight-balanced B-trees are also preferable to BB[α]-trees for purposes of augmenting one-dimensional data structures with range restriction capabilities [127].

27.3.3 Parent Pointers and Level-Balanced B-trees

It is sometimes useful to augment B-trees with parent pointers. For example, if we represent a total order via the leaves in a B-tree, we can answer order queries such as "Is $x < y$ in the total order?" by walking upwards in the B-tree from the leaves for x and y until we reach their common ancestor. Order queries arise in online algorithms for planar point location and for determining reachability in monotone subdivisions [2]. If we augment a conventional B-tree with parent pointers, then each split operation costs $\Theta(B)$ I/Os to update parent pointers, although the I/O cost is only $O(1)$ when amortized over the updates to the node. However, this amortized bound does not apply if the B-tree needs to support cut and concatenate operations, in which case the B-tree is cut into contiguous pieces and the pieces are rearranged arbitrarily. For example, reachability queries in a monotone subdivision are processed by maintaining two total orders, called the leftist and rightist orders, each of which is represented by a B-tree. When an edge is inserted or deleted, the tree representing each order is cut into four consecutive pieces, and the four pieces are rearranged via concatenate operations into a new total order. Doing cuts and concatenation via conventional B-trees augmented with parent pointers will require $\Theta(B)$ I/Os per level in the worst case. Node splits can occur with each operation (unlike the case where there are only inserts and deletes), and thus there is no convenient amortization argument that can be applied.

Agarwal et al. [2] describe an interesting variant of B-trees called *level-balanced B-trees* for handling parent pointers and operations such as cut and concatenate. The balancing condition is "global": The data structure represents a forest of B-trees in which the number of nodes on level h in the forest is allowed to be at most $N_h = 2N/(b/3)^h$, where b is some fixed parameter in the range $4 < b < B/2$. It immediately follows that the total height of the forest is roughly $\log_b N$.

Unlike previous variants of B-trees, the degrees of individual nodes of level-balanced B-trees can be arbitrarily small, and for storage purposes, nodes are packed together into disk blocks. Each node in the forest is stored as a node record (which points to the parent's node record) and a doubly linked list of child records (which point to the node records of the children). There are also pointers between the node record and the list of child records. Every disk block stores only node records or only child records, but all the child records for a given node must be stored in the same block (possibly with child records for other nodes). The advantage of this extra level of indirection is that cuts and concatenates can usually be done in only $O(1)$ I/Os per level of the forest. For example, during a cut, a node record gets split into two, and its list of child nodes is chopped into two separate lists. The parent node must therefore get a new child record to point to the new node. These updates require $O(1)$ I/Os except when there is not enough space in the disk block of the parent's child records, in which case the block must be split into two, and extra I/Os are needed to update the pointers to the moved child records. The amortized I/O cost, however, is only $O(1)$ per level, since each update creates at most one node record and child record at each level. The other dynamic update operations can be handled similarly.

All that remains is to reestablish the global level invariant when a level gets too many nodes as a result of an update. If level h is the lowest such level out of balance, then level h and all the levels above it are reconstructed via a postorder traversal in $O(N_h)$ I/Os so that the new nodes get degree $\Theta(b)$ and the invariant is restored. The final trick is to construct the new parent pointers that point from the $\Theta(N_{h-1}) = \Theta(bN_h)$ node records on level $h-1$ to the $\Theta(N_h)$ level-h nodes. The parent pointers can be accessed in a blocked manner with respect to the new ordering of the nodes on level h. By sorting, the pointers can be rearranged to correspond to the ordering of the nodes on level $h-1$, after which the parent pointer values can be written via a linear scan. The resulting I/O cost is $O\big((bN_h/B)\log_m(bN_h/B)\big)$, which can be amortized against the $\Theta(N_h)$ updates that occurred since the last time the level-h invariant was violated, yielding an amortized update cost of $O\big(1+(b/B)\log_m n\big)$ I/Os per level.

Order queries such as "Does leaf x precede leaf y in the total order represented by the tree?" can be answered using $O(\log_B N)$ I/Os by following parent pointers starting at x and y. The update operations insert, delete, cut, and concatenate can be done in $O\big(\big(1+(b/B)\log_m n\big)\log_b N\big)$ I/Os amortized, for any $2 \leq b \leq B/2$, which is never worse than $O\big((\log_B N)^2\big)$ by appropriate choice of b.

Using the multislab decomposition we discuss in Section 27.4.3, Agarwal et al. [2] apply level-balanced B-trees in a data structure for point location in monotone subdivisions, which supports queries and (amortized) updates in $O\big((\log_B N)^2\big)$ I/Os. They also use it to dynamically maintain planar st-graphs using $O\big((1+(b/B)(\log_m n)\log_b N\big)$ I/Os (amortized) per update, so that reachability queries can be answered in $O(\log_B N)$ I/Os (worst-case). (Planar st-graphs are planar directed acyclic graphs with a single source and a single sink.) An interesting open question is whether level-balanced B-trees can be implemented in $O(\log_B N)$ I/Os per update. Such an improvement would immediately give an optimal dynamic structure for reachability queries in planar st-graphs.

27.3.4 Buffer Trees

An important paradigm for constructing algorithms for batched problems in an internal memory setting is to use a dynamic data structure to process a sequence of updates. For example, we can sort N items by inserting them one by one into a priority queue, followed by a sequence of N *delete_min* operations. Similarly, many batched problems in computational geometry can be solved by dynamic plane sweep techniques. For example, in Section 27.2 we showed how to compute orthogonal segment intersections by dynamically keeping track of the active vertical segments (i.e., those hit by the horizontal sweep line); we mentioned a similar algorithm for orthogonal rectangle intersections.

However, if we use this paradigm naively in an EM setting, with a B-tree as the dynamic data structure, the resulting I/O performance will be highly nonoptimal. For example, if we use a B-tree as the priority queue in sorting or to store the active vertical segments hit by the sweep line, each update and query operation will take $O(\log_B N)$ I/Os, resulting in a total of $O(N \log_B N)$ I/Os, which is larger than the optimal bound $O(n \log_m n)$ by a substantial factor of roughly B. One solution suggested in [123] is to use a binary tree data structure in which items are pushed lazily down the tree in blocks of B items at a time. The binary nature of the tree results in a data structure of height $O(\log n)$, yielding a total I/O bound of $O(n \log n)$, which is still nonoptimal by a significant $\log m$ factor.

Arge [10] developed the elegant *buffer tree* data structure to support *batched dynamic* operations, as in the sweep line example, where the queries do not have to be answered right away or in any particular order. The buffer tree is a balanced multiway tree, but with degree $\Theta(m)$ rather than degree $\Theta(B)$, except possibly for the root. Its key distinguishing feature is that each node has a buffer that can store $\Theta(M)$ items (i.e., $\Theta(m)$ blocks of items). Items in a node are pushed down to the children when the buffer fills. Emptying a full buffer requires $\Theta(m)$ I/Os, which amortizes the cost of distributing the M items to the $\Theta(m)$ children. Each item thus incurs an amortized cost of $O(m/M) = O(1/B)$ I/Os per level, and the resulting cost for queries and updates is $O\big((1/B) \log_m n\big)$ I/Os amortized.

Buffer trees have an ever-expanding list of applications. They can be used as a subroutine in the standard sweep line algorithm in order to get an optimal EM algorithm for orthogonal segment intersection. Arge showed how to extend buffer trees to implement segment trees [28] in external memory in a batched dynamic setting by reducing the node degrees to $\Theta(\sqrt{m}\,)$ and by introducing *multislabs* in each node, which were explained in Section 27.2 for the related batched problem of intersecting rectangles. Buffer trees provide a natural amortized implementation of priority queues for *time-forward processing* applications such as discrete event simulation, sweeping, and list ranking [41]. Govindrajan et al. [60] use time-forward processing to construct a well-separated pair decomposition of N points in d dimensions in $O\big(n \log_m n\big)$ I/Os, and they apply it to the problems of finding the K nearest neighbors for each point and the K closest pairs. Brodal and Katajainen [33] provide a worst-case optimal priority queue, in the sense that every sequence of B *insert* and *delete_min* operations requires only $O(\log_m n)$ I/Os. Practical implementations of priority queues based upon these ideas are examined in [32, 109]. In Section 27.4.2 we report on some timing experiments involving buffer trees for use in bulk loading of R-trees. Further experiments on buffer trees appear in [68].

27.4 Spatial Data Structures and Range Search

In this section we consider online EM data structures for storing and querying spatial data. A fundamental database primitive in spatial databases and geographic information systems (GIS) is range search, which includes dictionary lookup as a special case. An orthogonal range query, for a given d-dimensional rectangle, returns all the points in the interior of the rectangle. In this section we use range searching (especially for the orthogonal 2-D case when $d = 2$) as the canonical query operation on spatial data. Other types of spatial queries include point location, ray shooting, nearest neighbor, and intersection queries, but for brevity we restrict our attention primarily to range searching.

There are two types of spatial data structures: data-driven and space-driven. R-trees and kd-trees are data-driven since they are based upon a partitioning of the data items themselves, whereas space-driven methods such as quad trees and grid files are organized by a partitioning of the embedding space, akin to order-preserving hash functions. In this section we discuss primarily data-driven data structures.

Multidimensional range search is a fundamental primitive in several online geometric applications, and it provides indexing support for constraint and object-oriented data models.

(See [74] for background.) We have already discussed multidimensional range searching in a batched setting in Section 27.2. In this section we concentrate on data structures for the online case.

For many types of range searching problems, it is very difficult to develop theoretically optimal algorithms and data structures. Many open problems remain. The goal for online data structures is typically to achieve the three optimality Criteria O1–O3 of Section 27.1.2.

We explain in Section 27.4.6 that under a fairly general computational model for general 2-D orthogonal queries, as pictured in Figure 27.5(d), it is impossible to satisfy Criteria O1 and O2 simultaneously. At least $\Omega\big(n(\log n)/\log(\log_B N + 1)\big)$ blocks of disk space must be used to achieve a query bound of $O\big((\log_B N)^c + z\big)$ I/Os per query, for any constant c [113]. Three natural questions arise:

- What sort of performance can be achieved when using only a linear amount of disk space? In Sections 27.4.1 and 27.4.2, we discuss some of the linear-space data structures used extensively in practice. None of them come close to satisfying Criteria O1 and O3 for range search in the worst case, but in typical-case scenarios they often perform well. We devote Section 27.4.2 to R-trees and their variants, which are the most popular general-purpose spatial structures developed to date.
- Since the lower bound applies only to general 2-D rectangular queries, are there any data structures that meet Criteria O1–O3 for the important special cases of 2-D range searching pictured in Figures 27.5(a), 27.5(b), and 27.5(c)? Fortunately the answer is yes. We show in Sections 27.4.3 and 27.4.4 how to use a "bootstrapping" paradigm to achieve optimal search and update performance.
- Can we meet Criteria O1 and O2 for general four-sided range searching if the disk space allowance is increased to $O\big(n(\log n)/\log(\log_B N + 1)\big)$ blocks? Yes again! In Section 27.4.5 we show how to adapt the optimal structure for three-sided searching in order to handle general four-sided searching in optimal search cost. The update cost, however, is not known to be optimal.

In Section 27.5 we discuss other scenarios of range search dealing with three dimensions and nonorthogonal queries. We discuss the lower bounds for 2-D range searching in Section 27.4.6.

27.4.1 Linear-Space Spatial Structures

Grossi and Italiano [62] construct an elegant multidimensional version of the B-tree called the *cross tree*. Using linear space, it combines the data-driven partitioning of weight-balanced B-trees (cf. Section 27.3.2) at the upper levels of the tree with the space-driven partitioning of methods such as quad trees at the lower levels of the tree. For $d > 1$, d-

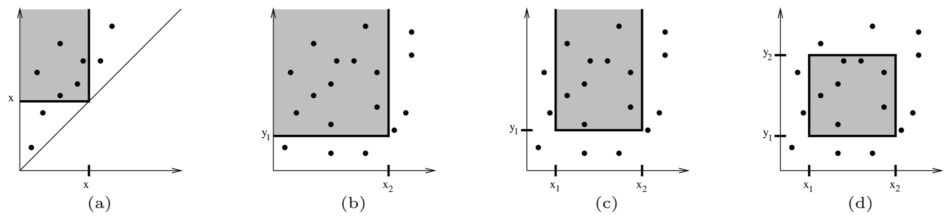


FIGURE 27.5: Different types of 2-D orthogonal range queries: (a) Diagonal corner two-sided 2-D query (equivalent to a stabbing query, cf. Section 27.4.3), (b) two-sided 2-D query, (c) three-sided 2-D query, and (d) general four-sided 2-D query.

dimensional orthogonal range queries can be done in $O(n^{1-1/d} + z)$ I/Os, and inserts and deletes take $O(\log_B N)$ I/Os. The O-tree of Kanth and Singh [75] provides similar bounds. Cross trees also support the dynamic operations of cut and concatenate in $O(n^{1-1/d})$ I/Os. In some restricted models for linear-space data structures, the 2-D range search query performance of cross trees and O-trees can be considered to be optimal, although it is much larger than the logarithmic bound of Criterion O1.

One way to get multidimensional EM data structures is to augment known internal memory structures, such as quad trees and kd-trees, with block-access capabilities. Examples include *kd-B-trees* [102], *buddy trees* [110], and *hB-trees* [54, 86]. *Grid files* [67, 80, 90] are a flattened data structure for storing the cells of a two-dimensional grid in disk blocks. Another technique is to "linearize" the multidimensional space by imposing a total ordering on it (a so-called space-filling curve), and then the total order is used to organize the points into a B-tree [57, 71, 95]. Linearization can also be used to represent nonpoint data, in which the data items are partitioned into one or more multidimensional rectangular regions [1, 94]. All the methods described in this paragraph use linear space, and they work well in certain situations; however, their worst-case range query performance is no better than that of cross trees, and for some methods, such as grid files, queries can require $\Theta(n)$ I/Os, even if there are no points satisfying the query. We refer the reader to [8, 56, 92] for a broad survey of these and other interesting methods. Space-filling curves arise again in connection with R-trees, which we describe next.

27.4.2 R-trees

The *R-tree* of Guttman [64] and its many variants are a practical multidimensional generalization of the B-tree for storing a variety of geometric objects, such as points, segments, polygons, and polyhedra, using linear disk space. Internal nodes have degree $\Theta(B)$ (except possibly the root), and leaves store $\Theta(B)$ items. Each node in the tree has associated with it a bounding box (or bounding polygon) of all the items in its subtree. A big difference between R-trees and B-trees is that in R-trees the bounding boxes of sibling nodes are allowed to overlap. If an R-tree is being used for point location, for example, a point may lie within the bounding box of several children of the current node in the search. In that case the search must proceed to all such children.

In the dynamic setting, there are several popular heuristics for where to insert new items into an R-tree and how to rebalance it; see Chapter 21 and [8, 56, 61] for a survey. The *R* tree* variant of Beckmann et al. [27] seems to give best overall query performance. To insert an item, we start at the root and recursively insert the item into the subtree whose bounding box would expand the least in order to accommodate the item. In case of a tie (e.g., if the item already fits inside the bounding boxes of two or more subtrees), we choose the subtree with the smallest resulting bounding box. In the normal R-tree algorithm, if a leaf node gets too many items or if an internal node gets too many children, we split it, as in B-trees. Instead, in the R*-tree algorithm, we remove a certain percentage of the items from the overflowing node and reinsert them into the tree. The items we choose to reinsert are the ones whose centroids are furthest from the center of the node's bounding box. This *forced reinsertion* tends to improve global organization and reduce query time. If the node still overflows after the forced reinsertion, we split it. The splitting heuristics try to partition the items into nodes so as to minimize intuitive measures such as coverage, overlap, or perimeter. During deletion, in both the normal R-tree and R*-tree algorithms, if a leaf node has too few items or if an internal node has too few children, we delete the node and reinsert all its items back into the tree by forced reinsertion.

The rebalancing heuristics perform well in many practical scenarios, especially in low

Data Set	Update Method	Update with 50% of the data		
		Building	Querying	Packing
RI	naive	259, 263	6, 670	64%
	Hilbert	15, 865	7, 262	92%
	buffer	13, 484	5, 485	90%
CT	naive	805, 749	40, 910	66%
	Hilbert	51, 086	40, 593	92%
	buffer	42, 774	37, 798	90%
NJ	naive	1, 777, 570	70, 830	66%
	Hilbert	120, 034	69, 798	92%
	buffer	101, 017	65, 898	91%
NY	naive	3, 736, 601	224, 039	66%
	Hilbert	246, 466	230, 990	92%
	buffer	206, 921	227, 559	90%

TABLE 27.1 Summary of the costs (in number of I/Os) for R-tree updates and queries. Packing refers to the percentage storage utilization.

dimensions, but they result in poor worst-case query bounds. An interesting open problem is whether nontrivial query bounds can be proven for the "typical-case" behavior of R-trees for problems such as range searching and point location. Similar questions apply to the methods discussed in Section 27.4.1. New R-tree partitioning methods by de Berg et al. [46] and Agarwal et al. [7] provide some provable bounds on overlap and query performance.

In the static setting, in which there are no updates, constructing the R*-tree by repeated insertions, one by one, is extremely slow. A faster alternative to the dynamic R-tree construction algorithms mentioned above is to bulk-load the R-tree in a bottom-up fashion [1, 70, 94]. Such methods use some heuristic for grouping the items into leaf nodes of the R-tree, and then recursively build the nonleaf nodes from bottom to top. As an example, in the so-called Hilbert R-tree of Kamel and Faloutsos [70], each item is labeled with the position of its centroid on the Peano-Hilbert space-filling curve, and a B^+-tree is built upon the totally ordered labels in a bottom-up manner. Bulk loading a Hilbert R-tree is therefore easy to do once the centroid points are presorted. These static construction methods algorithms are very different in spirit from the dynamic insertion methods: The dynamic methods explicitly try to reduce the coverage, overlap, or perimeter of the bounding boxes of the R-tree nodes, and as a result, they usually achieve good query performance. The static construction methods do not consider the bounding box information at all. Instead, the hope is that the improved storage utilization (up to 100%) of these packing methods compensates for a higher degree of node overlap. A dynamic insertion method related to [70] was presented in [71]. The quality of the Hilbert R-tree in terms of query performance is generally not as good as that of an R*-tree, especially for higher-dimensional data [30, 72].

In order to get the best of both worlds—the query performance of R*-trees and the bulk construction efficiency of Hilbert R-trees—Arge et al. [11] and van den Bercken et al. [118] independently devised fast bulk loading methods based upon buffer trees that do top-down construction in $O(n \log_m n)$ I/Os, which matches the performance of the bottom-up methods within a constant factor. The former method is especially efficient and supports dynamic batched updates and queries. In Figure 27.6 and Table 27.1, we report on some experiments that test the construction, update, and query performance of various R-tree methods. The experimental data came from TIGER/line data sets from four U.S. states [116]; the implementations were done using the TPIE system.

Figure 27.6 compares the construction cost for building R-trees and the resulting query performance in terms of I/Os for the naive sequential method for construction into R*-trees (labeled "naive") and the newly developed buffer R*-tree method [11] (labeled "buffer"). An R-tree was constructed on the TIGER road data for each state and for each of four possible buffer sizes. The four buffer sizes were capable of storing 0, 600, 1,250, and 5,000 rectangles,

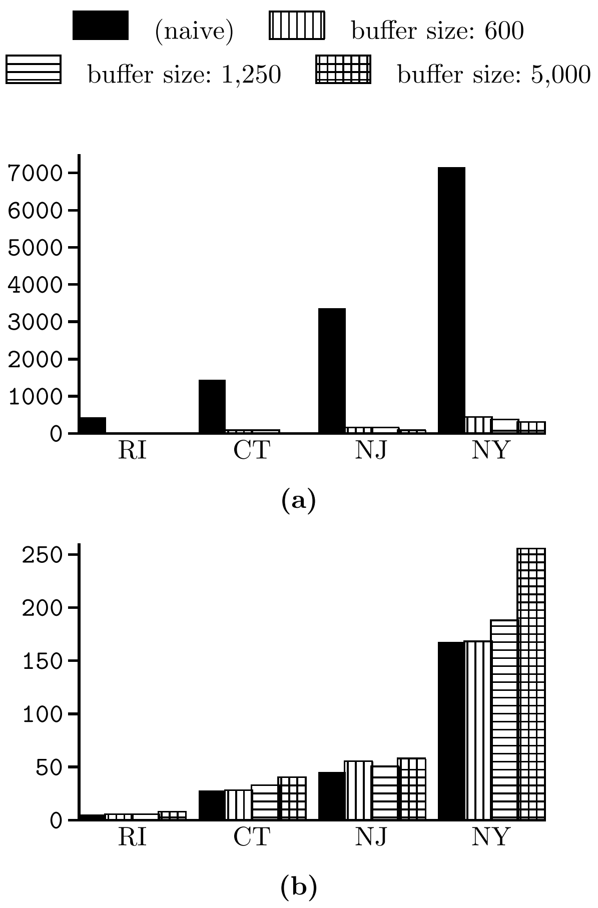


FIGURE 27.6: Costs for R-tree processing (in units of 1000 I/Os) using the naive repeated insertion method and the buffer R-tree for various buffer sizes: (a) cost for bulk-loading the R-tree, (b) query cost.

respectively; buffer size 0 corresponds to the naive method, and the larger buffers correspond to the buffer method. The query performance of each resulting R-tree was measured by posing rectangle intersection queries using rectangles taken from TIGER hydrographic data. The results, depicted in Figure 27.6, show that buffer R*-trees, even with relatively small buffers, achieve a tremendous speedup in number of I/Os for construction without any worsening in query performance, compared with the naive method. The CPU costs of the two methods are comparable. The storage utilization of buffer R*-trees tends to be in the 90% range, as opposed to roughly 70% for the naive method.

Bottom-up methods can build R-trees even more quickly and more compactly, but they generally do not support bulk dynamic operations, which is a big advantage of the buffer tree approach. Kamel et al. [72] develop a way to do bulk updates with Hilbert R-trees, but at a cost in terms of query performance. Table 27.1 compares dynamic update methods for the naive method, for buffer R-trees, and for Hilbert R-trees [72] (labeled “Hilbert”). A single R-tree was built for each of the four U.S. states, containing 50% of the road data objects for that state. Using each of the three algorithms, the remaining 50% of the objects were inserted into the R-tree, and the construction time was measured. Query performance was then tested as before. The results in Table 27.1 indicate that the buffer R*-tree and the

Hilbert R-tree achieve a similar degree of packing, but the buffer R*-tree provides better update and query performance.

27.4.3 Bootstrapping for 2-D Diagonal Corner and Stabbing Queries

An obvious paradigm for developing an efficient dynamic EM data structure, given an existing data structure that works well when the problem fits into internal memory, is to "externalize" the internal memory data structure. If the internal memory data structure uses a binary tree, then a multiway tree such as a B-tree must be used instead. However, when searching a B-tree, it can be difficult to report the outputs in an output-sensitive manner. For example, in certain searching applications, each of the $\Theta(B)$ subtrees of a given node in a B-tree may contribute one item to the query output, and as a result each subtree may need to be explored (costing several I/Os) just to report a single output item.

Fortunately, we can sometimes achieve output-sensitive reporting by augmenting the data structure with a set of filtering substructures, each of which is a data structure for a smaller version of the same problem. We refer to this approach, which we explain shortly in more detail, as the *bootstrapping* paradigm. Each substructure typically needs to store only $O(B^2)$ items and to answer queries in $O(\log_B B^2 + Z'/B) = O\big(\lceil Z'/B \rceil\big)$ I/Os, where Z' is the number of items reported. A substructure can even be static if it can be constructed in $O(B)$ I/Os, since we can keep updates in a separate buffer and do a global rebuilding in $O(B)$ I/Os whenever there are $\Theta(B)$ updates. Such a rebuilding costs $O(1)$ I/Os (amortized) per update. We can often remove the amortization and make it worst-case using the weight-balanced B-trees of Section 27.3.2 as the underlying B-tree structure.

Arge and Vitter [17] first uncovered the bootstrapping paradigm while designing an optimal dynamic EM data structure for diagonal corner two-sided 2-D queries (see Figure 27.5(a)) that meets all three design criteria for online data structures listed in Section 27.1.2. Diagonal corner two-sided queries are equivalent to stabbing queries, which have the following form: "Given a set of one-dimensional intervals, report all the intervals 'stabbed' by the query value x." (That is, report all intervals that contain x.) A diagonal corner query x on a set of 2-D points $\{(a_1, b_2),\ (a_2, b_2),\ \dots\ \}$ is equivalent to a stabbing query x on the set of closed intervals $\{[a_1, b_2],\ [a_2, b_2],\ \dots\ \}$.

The EM data structure for stabbing queries is a multiway version of the well-known interval tree data structure [51, 52] for internal memory, which supports stabbing queries in $O(\log N + Z)$ CPU time and updates in $O(\log N)$ CPU time and uses $O(N)$ space. We can externalize it by using a weight-balanced B-tree as the underlying base tree, where the nodes have degree $\Theta(\sqrt{B}\,)$. Each node in the base tree corresponds in a natural way to a one-dimensional range of x-values; its $\Theta(\sqrt{B}\,)$ children correspond to subranges called slabs, and the $\Theta(\sqrt{B}^2) = \Theta(B)$ contiguous sets of slabs are called *multislabs*, as in Section 27.2 for a similar batched problem. Each input interval is stored in the lowest node v in the base tree whose range completely contains the interval. The interval is decomposed by v's $\Theta(\sqrt{B}\,)$ slabs into at most three pieces: the middle piece that completely spans one or more slabs of v, the left end piece that partially protrudes into a slab of v, and the right end piece that partially protrudes into another slab of v, as shown in Figure 27.7. The three pieces are stored in substructures of v. In the example in Figure 27.7, the middle piece is stored in a list associated with the multislab it spans (corresponding to the contiguous range of slabs 3–5), the left end piece is stored in a one-dimensional list for slab 2 ordered by left endpoint, and the right end piece is stored in a one-dimensional list for slab 6 ordered by right endpoint.

Given a query value x, the intervals stabbed by x reside in the substructures of the nodes of the base tree along the search path from the root to the leaf for x. For each such

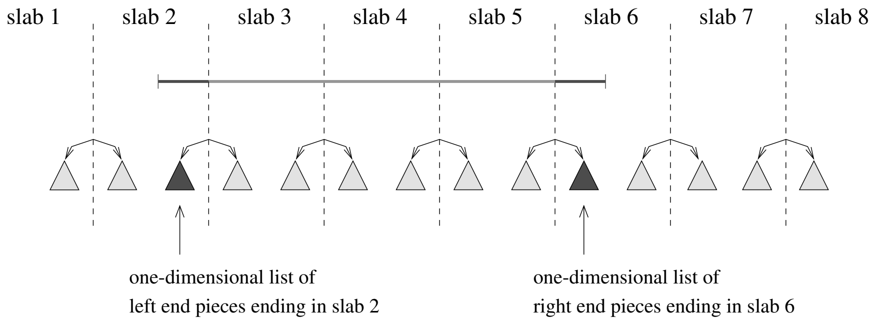

slab 1
slab 2
slab 3
slab 4
slab 5
slab 6
slab 7
slab 8
one-dimensional list of left end pieces ending in slab 2
one-dimensional list of right end pieces ending in slab 6

FIGURE 27.7: Internal node v of the EM priority search tree, for $B = 64$ with $\sqrt{B} = 8$ slabs. Node v is the lowest node in the tree completely containing the indicated interval. The middle piece of the interval is stored in the multislab list corresponding to slabs 3–5. (The multislab lists are not pictured.) The left and right end pieces of the interval are stored in the left-ordered list of slab 2 and the right-ordered list of slab 6, respectively.

node v, we consider each of v's multislabs that contains x and report all the intervals in the multislab list. We also walk sequentially through the right-ordered list and left-ordered list for the slab of v that contains x, reporting intervals in an output-sensitive way.

The big problem with this approach is that we have to spend at least one I/O per multislab containing x, regardless of how many intervals are in the multislab lists. For example, there may be $\Theta(B)$ such multislab lists, with each list containing only a few stabbed intervals (or worse yet, none at all). The resulting query performance will be highly nonoptimal. The solution, according to the bootstrapping paradigm, is to use a substructure in each node consisting of an optimal static data structure for a smaller version of the same problem; a good choice is the corner data structure developed by Kanellakis et al. [74]. The corner substructure in this case is used to store all the intervals from the "sparse" multislab lists, namely those that contain fewer than B intervals, and thus the substructure contains only $O(B^2)$ intervals. When visiting node v, we access only v's nonsparse multislab lists, each of which contributes $Z' \geq B$ intervals to the output, at an output-sensitive cost of $O(Z'/B)$ I/Os, for some Z'. The remaining Z'' stabbed intervals stored in v can be found by a single query to v's corner substructure, at a cost of $O(\log_B B^2 + Z''/B) = O\big(\lceil Z''/B \rceil\big)$ I/Os. Since there are $O(\log_B N)$ nodes along the search path in the base tree, the total collection of Z stabbed intervals is reported in $O(\log_B N + z)$ I/Os, which is optimal. Using a weight-balanced B-tree as the underlying base tree allows the static substructures to be rebuilt in worst-case optimal I/O bounds.

Stabbing queries are important because, when combined with one-dimensional range queries, they provide a solution to *dynamic interval management*, in which one-dimensional intervals can be inserted and deleted, and intersection queries can be performed. These operations support indexing of one-dimensional constraints in constraint databases. Other applications of stabbing queries arise in graphics and GIS. For example, Chiang and Silva [40] apply the EM interval tree structure to extract at query time the boundary components of the isosurface (or contour) of a surface. A data structure for a related problem, which in addition has optimal output complexity, appears in [5]. The above bootstrapping approach also yields dynamic EM segment trees with optimal query and update bound and $O(n \log_B N)$-block space usage.

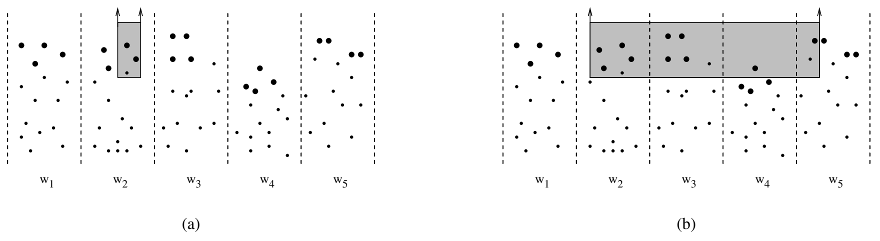


FIGURE 27.8: Internal node v of the EM priority search tree, with slabs (children) w_1, w_2, ..., w_5. The Y-sets of each child, which are stored collectively in v's child cache, are indicated by the bold points. (a) The three-sided query is completely contained in the x-range of w_2. The relevant (bold) points are reported from v's child cache, and the query is recursively answered in w_2. (b) The three-sided query spans several slabs. The relevant (bold) points are reported from v's child cache, and the query is recursively answered in w_2, w_3, and w_5. The query is *not* extended to w_4 in this case because not all of its Y-set $Y(w_4)$ (stored in v's child cache) satisfies the query, and as a result, none of the points stored in w_4's subtree can satisfy the query.

27.4.4 Bootstrapping for Three-Sided Orthogonal 2-D Range Search

Arge et al. [15] provide another example of the bootstrapping paradigm by developing an optimal dynamic EM data structure for three-sided orthogonal 2-D range searching (see Figure 27.5(c)) that meets all three design Criteria O1–O3. In internal memory, the optimal structure is the priority search tree [88], which answers three-sided range queries in $O(\log N + Z)$ CPU time, does updates in $O(\log N)$ CPU time, and uses $O(N)$ space. The EM structure of Arge et al. [15] is an externalization of the priority search tree, using a weight-balanced B-tree as the underlying base tree. Each node in the base tree corresponds to a one-dimensional range of x-values, and its $\Theta(B)$ children correspond to subranges consisting of vertical slabs. Each node v contains a small substructure called a *child cache* that supports three-sided queries. Its child cache stores the "Y-set" $Y(w)$ for each of the $\Theta(B)$ children w of v. The Y-set $Y(w)$ for child w consists of the highest $\Theta(B)$ points in w's slab that are not already stored in the child cache of some ancestor of v. There are thus a total of $\Theta(B^2)$ points stored in v's child cache.

We can answer a three-sided query of the form $[x_1, x_2] \times [y_1, +\infty)$ by visiting a set of nodes in the base tree, starting with the root. For each visited node v, we pose the query $[x_1, x_2] \times [y_1, +\infty)$ to v's child cache and output the results. The following rules are used to determine which of v's children to visit: We visit v's child w if either

1. w is along the leftmost search path for x_1 or the rightmost search path for x_2 in the base tree, or
2. the entire Y-set $Y(w)$ is reported when v's child cache is queried.

(See Figure 27.8.) There are $O(\log_B N)$ nodes w that are visited because of rule 1. When rule 1 is not satisfied, rule 2 provides an effective filtering mechanism to guarantee output-sensitive reporting: The I/O cost for initially accessing a child node w can be charged to the $\Theta(B)$ points of $Y(w)$ reported from v's child cache; conversely, if not all of $Y(w)$ is reported, then the points stored in w's subtree will be too low to satisfy the query, and there is no need to visit w. (See Figure 27.8(b).) Provided that each child cache can be queried in $O(1)$ I/Os plus the output-sensitive cost to output the points satisfying the query, the resulting overall query time is $O(\log_B N + z)$, as desired.

All that remains is to show how to query a child cache in a constant number of I/Os, plus the output-sensitive cost. Arge et al. [15] provide an elegant and optimal static data structure for three-sided range search, which can be used in the EM priority search tree described above to implement the child caches of size $O(B^2)$. The static structure is a persistent B-tree optimized for batched construction. When used for $O(B^2)$ points, it occupies $O(B)$ blocks, can be built in $O(B)$ I/Os, and supports three-sided queries in $O\big(\lceil Z'/B \rceil\big)$ I/Os per query, where Z' is the number of points reported. The static structure is so simple that it may be useful in practice on its own.

Both the three-sided structure developed by Arge et al. [15] and the structure for two-sided diagonal queries discussed in Section 27.4.3 satisfy Criteria O1–O3 of Section 27.1.2. So in a sense, the three-sided query structure subsumes the diagonal two-sided structure, since three-sided queries are more general. However, diagonal two-sided structure may prove to be faster in practice, because in each of its corner substructures, the data accessed during a query are always in contiguous blocks, whereas the static substructures used for three-sided search do not guarantee block contiguity. Empirical work is ongoing to evaluate the performance of these data structures.

On a historical note, earlier work on two-sided and three-sided queries was done by Ramaswamy and Subramanian [99] using the notion of *path caching*; their structure met Criterion O1 but had higher storage overheads and amortized and/or nonoptimal update bounds. Subramanian and Ramaswamy [113] subsequently developed the *p-range tree* data structure for three-sided queries, with optimal linear disk space and nearly optimal query and amortized update bounds.

27.4.5 General Orthogonal 2-D Range Search

The dynamic data structure for three-sided range searching can be generalized using the filtering technique of Chazelle [35] to handle general four-sided queries with optimal I/O query bound $O(\log_B N + z)$ and optimal disk space usage $O\big(n(\log n)/\log(\log_B N + 1)\big)$ [15]. The update bound becomes $O\big((\log_B N)(\log n)/\log(\log_B N + 1)\big)$, which may not be optimal.

The outer level of the structure is a balanced $(\log_B N + 1)$-way 1-D search tree with $\Theta(n)$ leaves, oriented, say, along the x-dimension. It therefore has about $(\log n)/\log(\log_B N + 1)$ levels. At each level of the tree, each input point is stored in four substructures (described below) that are associated with the particular tree node at that level that spans the x-value of the point. The space and update bounds quoted above follow from the fact that the substructures use linear space and can be updated in $O(\log_B N)$ I/Os.

To search for the points in a four-sided query rectangle $[x_1, x_2] \times [y_1, y_2]$, we decompose the four-sided query in the following natural way into two three-sided queries, a stabbing query, and $\log_B N - 1$ list traversals: We find the lowest node v in the tree whose x-range contains $[x_1, x_2]$. If v is a leaf, we can answer the query in a single I/O. Otherwise we query the substructures stored in those children of v whose x-ranges intersect $[x_1, x_2]$. Let $2 \leq k \leq \log_B N + 1$ be the number of such children. The range query when restricted to the leftmost such child of v is a three-sided query of the form $[x_1, +\infty] \times [y_1, y_2]$, and when restricted to the rightmost such child of v, the range query is a three-sided query of the form $[-\infty, x_2] \times [y_1, y_2]$. Two of the substructures at each node are devoted for three-sided queries of these types; using the linear-sized data structures of Arge et al. [15] in Section 27.4.4, each such query can be done in $O(\log_B N + z)$ I/Os.

For the $k - 2$ intermediate children of v, their x-ranges are completely contained inside the x-range of the query rectangle, and thus we need only do $k - 2$ list traversals in y-order and retrieve the points whose y-values are in the range $[y_1, y_2]$. If we store the points in each node in y-order (in the third type of substructure), the Z' output points from a node can be found in $O\big(\lceil Z'/B \rceil\big)$ I/Os, once a starting point in the linear list is found. We can find all

$k-2$ starting points via a single query to a stabbing query substructure S associated with v. (This structure is the fourth type of substructure.) For each two y-consecutive points (a_i, b_i) and (a_{i+1}, b_{i+1}) associated with a child of v, we store the y-interval $[b_i, b_{i+1}]$ in S. Note that S contains intervals contributed by each of the $\log_B N + 1$ children of v. By a single stabbing query with query value y_1, we can thus identify the $k-2$ starting points in only $O(\log_B N)$ I/Os [17], as described in Section 27.4.3. (We actually get starting points for all the children of v, not just the $k-2$ ones of interest, but we can discard the starting points we don't need.) The total number of I/Os to answer the range query is thus $O(\log_B N + z)$, which is optimal.

27.4.6 Lower Bounds for Orthogonal Range Search

We mentioned in Section 27.4 that Subramanian and Ramaswamy [113] prove that no EM data structure for 2-D range searching can achieve design Criterion O1 of Section 27.1.2 using less than $O\big(n(\log n)/\log(\log_B N + 1)\big)$ disk blocks, even if we relax the criterion and allow $O\big((\log_B N)^c + z\big)$ I/Os per query, for any constant c. The result holds for an EM version of the pointer machine model, based upon the approach of Chazelle [36] for the internal memory model.

Hellerstein et al. [65] consider a generalization of the layout-based lower bound argument of Kanellakis et al. [74] for studying the tradeoff between disk space usage and query performance. They develop a model for *indexability*, in which an "efficient" data structure is expected to contain the Z output points to a query compactly within $O\big(\lceil Z/B \rceil\big) = O\big(\lceil z \rceil\big)$ blocks. One shortcoming of the model is that it considers only data layout and ignores the search component of queries, and thus it rules out the important filtering paradigm discussed earlier in Section 27.4. For example, it is reasonable for any query to perform at least $\log_B N$ I/Os, so if the output size Z is at most B, a data structure may still be able to satisfy Criterion O1 even if the output is contained within $O(\log_B N)$ blocks rather than $O(z) = O(1)$ blocks. Arge et al. [15] modify the model to rederive the same nonlinear space lower bound $O\big(n(\log n)/\log(\log_B N + 1)\big)$ of Subramanian and Ramaswamy [113] for 2-D range searching by considering only output sizes Z larger than $(\log_B N)^c B$, for which the number of blocks allowed to hold the outputs is $Z/B = O\big((\log_B N)^c + z\big)$. This approach ignores the complexity of how to find the relevant blocks, but as mentioned in Section 27.4.5, the authors separately provide an optimal 2-D range search data structure that uses the same amount of disk space and does queries in the optimal $O(\log_B N + z)$ I/Os. Thus, despite its shortcomings, the indexability model is elegant and can provide much insight into the complexity of blocking data in external memory. Further results in this model appear in [79, 108].

One intuition from the indexability model is that less disk space is needed to efficiently answer 2-D queries when the queries have bounded aspect ratio (i.e., when the ratio of the longest side length to the shortest side length of the query rectangle is bounded). An interesting question is whether R-trees and the linear-space structures of Sections 27.4.1 and 27.4.2 can be shown to perform provably well for such queries. Another interesting scenario is where the queries correspond to snapshots of the continuous movement of a sliding rectangle.

When the data structure is restricted to contain only a single copy of each point, Kanth and Singh [75] show for a restricted class of index-based trees that d-dimensional range queries in the worst case require $\Omega(n^{1-1/d} + z)$ I/Os, and they provide a data structure with a matching bound. Another approach to achieve the same bound is the cross tree data structure [62] mentioned in Section 27.4.1, which in addition supports the operations of cut and concatenate.

27.5 Related Problems

For other types of range searching, such as in higher dimensions and for nonorthogonal queries, different filtering techniques are needed. So far, relatively little work has been done, and many open problems remain.

Vengroff and Vitter [121] develop the first theoretically near-optimal EM data structure for static three-dimensional orthogonal range searching. They create a hierarchical partitioning in which all the points that dominate a query point are densely contained in a set of blocks. Compression techniques are needed to minimize disk storage. With some recent modifications [126], queries can be done in $O(\log_B N + z)$ I/Os, which is optimal, and the space usage is $O\big(n(\log n)^{k+1}/(\log(\log_B N + 1))^k\big)$ disk blocks to support $(3+k)$-sided 3-D range queries, in which k of the dimensions $(0 \le k \le 3)$ have finite ranges. The result also provides optimal $O(\log N + Z)$-time query performance for three-sided 3-D queries in the (internal memory) RAM model, but using $O(N \log N)$ space.

By the reduction in [37], a data structure for three-sided 3-D queries also applies to *2-D homothetic range search*, in which the queries correspond to scaled and translated (but not rotated) transformations of an arbitrary fixed polygon. An interesting special case is "fat" orthogonal 2-D range search, where the query rectangles are required to have bounded aspect ratio. For example, every rectangle with bounded aspect ratio can be covered by two overlapping squares. An interesting open problem is to develop linear-sized optimal data structures for fat orthogonal 2-D range search. By the reduction, one possible approach would be to develop optimal linear-sized data structures for three-sided 3-D range search.

Agarwal et al. [4] consider halfspace range searching, in which a query is specified by a hyperplane and a bit indicating one of its two sides, and the output of the query consists of all the points on that side of the hyperplane. They give various data structures for halfspace range searching in two, three, and higher dimensions, including one that works for simplex (polygon) queries in two dimensions, but with a higher query I/O cost. They have subsequently improved the storage bounds for halfspace range queries in two dimensions to obtain an optimal static data structure satisfying Criteria O1 and O2 of Section 27.1.2.

The number of I/Os needed to build the data structures for 3-D orthogonal range search and halfspace range search is rather large (more than $\Omega(N)$). Still, the structures shed useful light on the complexity of range searching and may open the way to improved solutions. An open problem is to design efficient construction and update algorithms and to improve upon the constant factors.

Callahan et al. [34] develop dynamic EM data structures for several online problems in d dimensions. For any fixed $\epsilon > 0$, they can find an approximate nearest neighbor of a query point (within a $1 + \epsilon$ factor of optimal) in $O(\log_B N)$ I/Os; insertions and deletions can also be done in $O(\log_B N)$ I/Os. They use a related approach to maintain the closest pair of points; each update costs $O(\log_B N)$ I/Os. Govindrajan et al. [60] achieve the same bounds for closest pair by maintaining a well-separated pair decomposition. For finding nearest neighbors and approximate nearest neighbors, two other approaches are partition trees [3, 4] and locality-sensitive hashing [58]. Numerous other data structures have been developed for range queries and related problems on spatial data. We refer to [8, 56, 92] for a broad survey.

27.6 Dynamic and Kinetic Data Structures

In this section we consider two scenarios where data items change: *dynamic* (in which items are inserted and deleted), and *kinetic* (in which the data items move continuously along specified trajectories). In both cases, queries can be done at any time. It is often useful for kinetic data structures to allow insertions and deletions; for example, if the trajectory of an item changes, we must delete the old trajectory and insert the new one.

27.6.1 Logarithmic Method for Decomposable Search Problems

In previous sections, we've already encountered several dynamic data structures for the problems of dictionary lookup and range search. In Section 27.4 we saw how to develop optimal EM range search data structures by externalizing some known internal memory data structures. The key idea was to use the bootstrapping paradigm, together with weight-balanced B-trees as the underlying data structure, in order to consolidate several static data structures for small instances of range searching into one dynamic data structure for the full problem. The bootstrapping technique is specific to the particular data structures involved. In this section we look at another technique that is based upon the properties of the problem itself rather than upon that of the data structure.

We call a problem *decomposable* if we can answer a query by querying individual subsets of the problem data and then computing the final result from the solutions to each subset. Dictionary search and range searching are obvious examples of decomposable problems. Bentley developed the *logarithmic method* [29, 96] to convert efficient static data structures for decomposable problems into general dynamic ones. In the internal memory setting, the logarithmic method consists of maintaining a series of static substructures, at most one each of size 1, 2, 4, 8,.... When a new item is inserted, it is initialized in a substructure of size 1. If a substructure of size 1 already exists, the two substructures are combined into a single substructure of size 2. If there is already a substructure of size 2, they in turn are combined into a single substructure of size 4, and so on. For the current value of N, it is easy to see that the kth substructure (i.e., of size 2^k) is present exactly when the kth bit in the binary representation of N is 1. Since there are at most $\log N$ substructures, the search time bound is $\log N$ times the search time per substructure. As the number of items increases from 1 to N, the kth structure is built a total of $N/2^k$ times (assuming N is a power of 2). If it can be built in $O(2^k)$ time, the total time for all insertions and all substructures is thus $O(N \log N)$, making the amortized insertion time $O(\log N)$. If we use up to three substructures of size 2^k at a time, we can do the reconstructions in advance and convert the amortized update bounds to worst-case [96].

In the EM setting, in order to eliminate the dependence upon the binary logarithm in the I/O bounds, the number of substructures must be reduced from $\log N$ to $\log_B N$, and thus the maximum size of the kth substructure must be increased from 2^k to B^k. As the number of items increases from 1 to N, the kth substructure has to be built NB/B^k times (when N is a power of B), each time taking $O\big(B^k(\log_B N)/B\big)$ I/Os. The key point is that the extra factor of B in the numerator of the first term is canceled by the factor of B in the denominator of the second term, and thus the resulting total insertion time over all N insertions and all $\log_B N$ structures is $O\big(N(\log_B N)^2\big)$ I/Os, which is $O\big((\log_B N)^2\big)$ I/Os amortized per insertion. By global rebuilding we can do deletions in $O(\log_B N)$ I/Os amortized. As in the internal memory case, the amortized updates can typically be made worst-case.

Arge and Vahrenhold [16] obtain I/O bounds for dynamic point location in general planar subdivisions similar to those of [2], but without use of level-balanced trees. Their method uses a weight-balanced base structure at the outer level and a multislab structure for storing segments similar to that of Arge and Vitter [17] described in Section 27.4.3. They use the logarithmic method to construct a data structure to answer vertical rayshooting queries in the multislab structures. The method relies upon a total ordering of the segments, but such an ordering can be changed drastically by a single insertion. However, each substructure in the logarithmic method is static (until it is combined with another substructure), and thus a static total ordering can be used for each substructure. The authors show by a type of fractional cascading that the queries in the $\log_B N$ substructures do not have to be done independently, which saves a factor of $\log_B N$ in the I/O cost, but at the cost of making the data structures amortized instead of worst-case.

Agarwal et al. [6] apply the logarithmic method (in both the binary form and B-way variant) to get EM versions of kd-trees, BBD trees, and BAR trees.

27.6.2 Continuously Moving Items

Early work on temporal data generally concentrated on time-series or multiversion data [105]. A question of growing interest in this mobile age is how to store and index continuously moving items, such as mobile telephones, cars, and airplanes, using kinetic data structures (e.g., see [69, 104, 128]) . There are two main approaches to storing moving items: The first technique is to use the same sort of data structure as for nonmoving data, but to update it whenever items move sufficiently far so as to trigger important combinatorial *events* that are relevant to the application at hand [24]. For example, an event relevant for range search might be triggered when two items move to the same horizontal displacement (which happens when the x-ordering of the two items is about to switch). A different approach is to store each item's location and speed trajectory, so that no updating is needed as long as the item's trajectory plan does not change. Such an approach requires fewer updates, but the representation for each item generally has higher dimension, and the search strategies are therefore less efficient.

Kollios et al. [78] developed a linear-space indexing scheme for moving points along a (one-dimensional) line, based upon the notion of partition trees. Their structure supports a variety of range search and approximate nearest neighbor queries. For example, given a range and time, the points in that range at the indicated time can be retrieved in $O(n^{1/2+\epsilon}+k)$ I/Os, for arbitrarily small $\epsilon > 0$. Updates require $O\big((\log n)^2\big)$ I/Os. Agarwal et al. [3] extend the approach to handle range searches in two dimensions, and they improve the update bound to $O\big((\log_B n)^2\big)$ I/Os. They also propose an event-driven data structure with the same query times as the range search data structure of Arge and Vitter [15] discussed in Section 27.4.5, but with the potential need to do many updates. A hybrid data structure combining the two approaches permits a tradeoff between query performance and update frequency.

R-trees offer a practical generic mechanism for storing multidimensional points and are thus a natural alternative for storing mobile items. One approach is to represent time as a separate dimension and to cluster trajectories using the R-tree heuristics. However, the orthogonal nature of the R-tree does not lend itself well to diagonal trajectories. For the case of points moving along linear trajectories, Šaltenis et al. [104] build the R-tree upon only the spatial dimensions, but parameterize the bounding box coordinates to account for the movement of the items stored within. They maintain an outer approximation of the true bounding box, which they periodically update to refine the approximation. Agarwal and Har-Peled [9] show how to maintain a provably good approximation of the minimum bounding box with need for only a constant number of refinement events. Further discussion of kinetic data structures, primarily for internal memory, appears in Chapter 23.

27.7 Conclusions

In this chapter we have surveyed several useful paradigms and techniques for the design and implementation of efficient data structures for external memory. A variety of interesting challenges remain in geometric search applications, such as methods for high-dimensional and nonorthogonal range searches as well as the analysis of R-trees and linear-space methods for typical-case scenarios. A continuing goal is to translate theoretical gains into observable improvements in practice. For some of the problems that can be solved optimally up to a constant factor, the constant overhead is too large for the algorithm to be of practical use, and simpler approaches are needed.

Acknowledgment

The author wishes to thank the members of the Center for Geometric Computing at Duke University for helpful comments and suggestions. This work was supported in part by the Army Research Office through grant DAAD19–03–1–0321, and by the National Science Foundation through research grants CCR–9877133 and CCR–0082986.

References

[1] D. J. Abel. A B^+-tree structure for large quadtrees. *Computer Vision, Graphics, and Image Processing*, 27(1):19–31, July 1984.

[2] P. K. Agarwal, L. Arge, G. S. Brodal, and J. S. Vitter. I/O-efficient dynamic point location in monotone planar subdivisions. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, volume 10, pages 11–20, 1999.

[3] P. K. Agarwal, L. Arge, and J. Erickson. Indexing moving points. In *Proceedings of the ACM Symposium on Principles of Database Systems*, volume 19, pages 175–186, 2000.

[4] P. K. Agarwal, L. Arge, J. Erickson, P. G. Franciosa, and J. S. Vitter. Efficient searching with linear constraints. *Journal of Computer and System Sciences*, 61(2):194–216, October 2000.

[5] P. K. Agarwal, L. Arge, T. M. Murali, K. Varadarajan, and J. S. Vitter. I/O-efficient algorithms for contour line extraction and planar graph blocking. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, volume 9, pages 117–126, 1998.

[6] P. K. Agarwal, L. Arge, O. Procopiuc, and J. S. Vitter. A framework for index dynamization. In *Proceedings of the International Colloquium on Automata, Languages, and Programming*, volume 2076 of *Lecture Notes in Computer Science*, Crete, Greece, 2001. Springer-Verlag.

[7] P. K. Agarwal, M. de Berg, J. Gudmundsson, M. Hammar, and H. J. Haverkort. Constructing box trees and R-trees with low stabbing number. In *Proceedings of the ACM Symposium on Computational Geometry*, volume 17, June 2001.

[8] P. K. Agarwal and J. Erickson. Geometric range searching and its relatives. In B. Chazelle, J. E. Goodman, and R. Pollack, editors, *Advances in Discrete and Computational Geometry*, volume 23 of *Contemporary Mathematics*, pages 1–56. American Mathematical Society Press, Providence, RI, 1999.

[9] P. K. Agarwal and S. Har-Peled. Maintaining the approximate extent measures of moving points. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, volume 12, pages 148–157, Washington, January 2001.

[10] L. Arge. The buffer tree: A new technique for optimal I/O-algorithms. In *Proceedings*

of the Workshop on Algorithms and Data Structures, volume 955 of *Lecture Notes in Computer Science*, pages 334–345. Springer-Verlag, 1995. A complete version appears as BRICS technical report RS–96–28, University of Aarhus.

[11] L. Arge, K. H. Hinrichs, J. Vahrenhold, and J. S. Vitter. Efficient bulk operations on dynamic R-trees. *Algorithmica*, 33(1):104–128, January 2002.

[12] L. Arge, O. Procopiuc, S. Ramaswamy, T. Suel, J. Vahrenhold, and J. S. Vitter. A unified approach for indexed and non-indexed spatial joins. In *Proceedings of the International Conference on Extending Database Technology*, volume 7, Konstanz, Germany, March 2000.

[13] L. Arge, O. Procopiuc, S. Ramaswamy, T. Suel, and J. S. Vitter. Scalable sweeping-based spatial join. In *Proceedings of the International Conference on Very Large Databases*, volume 24, pages 570–581, New York, August 1998.

[14] L. Arge, O. Procopiuc, S. Ramaswamy, T. Suel, and J. S. Vitter. Theory and practice of I/O-efficient algorithms for multidimensional batched searching problems. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, volume 9, pages 685–694, 1998.

[15] L. Arge, V. Samoladas, and J. S. Vitter. Two-dimensional indexability and optimal range search indexing. In *Proceedings of the ACM Conference on Principles of Database Systems*, volume 18, pages 346–357, Philadelphia, PA, May–June 1999.

[16] L. Arge and J. Vahrenhold. I/O-efficient dynamic planar point location. In *Proceedings of the ACM Symposium on Computational Geometry*, volume 9, pages 191–200, June 2000.

[17] L. Arge and J. S. Vitter. Optimal dynamic interval management in external memory. *SIAM Journal on Computing*, 32(6):1488–1508, 2003.

[18] Lars Arge, Darren Erik Vengroff, and Jeffrey Scott Vitter. External-memory algorithms for processing line segments in geographic information systems. *Algorithmica*, to appear.

[19] R. Baeza-Yates. Bounded disorder: The effect of the index. *Theoretical Computer Science*, 168:21–38, 1996.

[20] R. A. Baeza-Yates. Expected behaviour of B^+-trees under random insertions. *Acta Informatica*, 26(5):439–472, 1989.

[21] R. A. Baeza-Yates and P.-A. Larson. Performance of B^+-trees with partial expansions. *IEEE Transactions on Knowledge and Data Engineering*, 1(2):248–257, June 1989.

[22] R. D. Barve, E. A. M. Shriver, P. B. Gibbons, B. K. Hillyer, Y. Matias, and J. S. Vitter. Modeling and optimizing I/O throughput of multiple disks on a bus: The long version. Technical report, Bell Labs, 1997.

[23] Rakesh D. Barve and Jeffrey S. Vitter. External memory algorithms with dynamically changing memory allocations: Long version. Technical Report CS–1998–09, Duke University, 1998.

[24] J. Basch, L. J. Guibas, and J. Hershberger. Data structures for mobile data. *Journal of Algorithms*, 31:1–28, 1999.

[25] R. Bayer and E. McCreight. Organization of large ordered indexes. *Acta Informatica*, 1:173–189, 1972.

[26] Bruno Becker, Stephan Gschwind, Thomas Ohler, Bernhard Seeger, and Peter Widmayer. An asymptotically optimal multiversion B-tree. *VLDB Journal*, 5(4):264–275, December 1996.

[27] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: An efficient and robust access method for points and rectangles. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 322–331, 1990.

[28] J. L. Bentley. Multidimensional divide and conquer. *Communications of the ACM*,

23(6):214–229, 1980.

[29] J. L. Bentley and J. B. Saxe. Decomposable searching problems I: Static-to-dynamic transformations. *Journal of Algorithms*, 1(4):301–358, December 1980.

[30] S. Berchtold, C. Böhm, and H-P. Kriegel. Improving the query performance of high-dimensional index structures by bulk load operations. In *Proceedings of the International Conference on Extending Database Technology*, volume 1377 of *Lecture Notes in Computer Science*, pages 216–230, 1998.

[31] Norbert Blum and Kurt Mehlhorn. On the average number of rebalancing operations in weight-balanced trees. *Theoretical Computer Science*, 11(3):303–320, July 1980.

[32] K. Brengel, A. Crauser, P. Ferragina, and U. Meyer. An experimental study of priority queues. In J. S. Vitter and C. Zaroliagis, editors, *Proceedings of the Workshop on Algorithm Engineering*, volume 1668 of *Lecture Notes in Computer Science*, pages 345–359, London, July 1999. Springer-Verlag.

[33] Gerth S. Brodal and Jyrki Katajainen. Worst-case efficient external-memory priority queues. In *Proceedings of the Scandinavian Workshop on Algorithmic Theory*, volume 1432 of *Lecture Notes in Computer Science*, pages 107–118, Stockholm, Sweden, July 1998. Springer-Verlag.

[34] P. Callahan, M. T. Goodrich, and K. Ramaiyer. Topology B-trees and their applications. In *Proceedings of the Workshop on Algorithms and Data Structures*, volume 955 of *Lecture Notes in Computer Science*, pages 381–392. Springer-Verlag, 1995.

[35] Bernard Chazelle. Filtering search: A new approach to query-answering. *SIAM Journal on Computing*, 15:703–724, 1986.

[36] Bernard Chazelle. Lower bounds for orthogonal range searching: I. The reporting case. *Journal of the ACM*, 37(2):200–212, April 1990.

[37] Bernard Chazelle and H. Edelsbrunner. Linear space data structures for two types of range search. *Discrete and Computational Geometry*, 2:113–126, 1987.

[38] P. M. Chen, E. K. Lee, G. A. Gibson, R. H. Katz, and D. A. Patterson. RAID: high-performance, reliable secondary storage. *ACM Computing Surveys*, 26(2):145–185, June 1994.

[39] Y.-J. Chiang. Experiments on the practical I/O efficiency of geometric algorithms: Distribution sweep vs. plane sweep. *Computational Geometry: Theory and Applications*, 8(4):211–236, 1998.

[40] Y.-J. Chiang and C. T. Silva. External memory techniques for isosurface extraction in scientific visualization. In J. Abello and J. S. Vitter, editors, *External Memory Algorithms and Visualization*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, pages 247–277, Providence, RI, 1999. American Mathematical Society Press.

[41] Yi-Jen Chiang, Michael T. Goodrich, Edward F. Grove, Roberto Tamassia, Darren Erik Vengroff, and Jeffrey Scott Vitter. External-memory graph algorithms. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, volume 6, pages 139–149, January 1995.

[42] K. L. Clarkson and P. W. Shor. Applications of random sampling in computational geometry, II. *Discrete and Computational Geometry*, 4:387–421, 1989.

[43] D. Comer. The ubiquitous B-tree. *ACM Computing Surveys*, 11(2):121–137, 1979.

[44] A. Crauser, P. Ferragina, K. Mehlhorn, U. Meyer, and E. A. Ramos. Randomized external-memory algorithms for geometric problems. In *Proceedings of the ACM Symposium on Computational Geometry*, volume 14, pages 259–268, June 1998.

[45] A. Crauser, P. Ferragina, K. Mehlhorn, U. Meyer, and E. A. Ramos. I/O-optimal computation of segment intersections. In J. Abello and J. S. Vitter, editors, *External Memory Algorithms and Visualization*, DIMACS Series in Discrete Mathematics

and Theoretical Computer Science, pages 131–138. American Mathematical Society Press, Providence, RI, 1999.

[46] M. de Berg, J. Gudmundsson, M. Hammar, and M. Overmars. On R-trees with low stabbing number. In *Proceedings of the European Symposium on Algorithms*, volume 1879 of *Lecture Notes in Computer Science*, pages 167–178, Saarbrücken, Germany, September 2000. Springer-Verlag.

[47] Mark de Berg, Mark van Kreveld, Mark Overmars, and Otfried Schwarzkopf. *Computational Geometry Algorithms and Applications*. Springer-Verlag, Berlin, 1997.

[48] W. Dittrich, D. Hutchinson, and A. Maheshwari. Blocking in parallel multisearch problems. In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures*, volume 10, pages 98–107, 1998.

[49] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan. Making data structures persistent. *Journal of Computer and System Sciences*, 38:86–124, 1989.

[50] M. C. Easton. Key-sequence data sets on indelible storage. *IBM Journal of Research and Development*, 30:230–241, 1986.

[51] H. Edelsbrunner. A new approach to rectangle intersections, part I. *International Journal of Computer Mathematics*, 13:209–219, 1983.

[52] H. Edelsbrunner. A new approach to rectangle intersections, part II. *International Journal of Computer Mathematics*, 13:221–229, 1983.

[53] NASA's Earth Observing System (EOS) web page, NASA Goddard Space Flight Center, http://eospso.gsfc.nasa.gov/.

[54] G. Evangelidis, D. B. Lomet, and B. Salzberg. The hB$^{\Pi}$-tree: A multi-attribute index supporting concurrency, recovery and node consolidation. *VLDB Journal*, 6:1–25, 1997.

[55] Thomas A. Funkhouser, Carlo H. Sequin, and Seth J. Teller. Management of large amounts of data in interactive building walkthroughs. In *Proceedings of the ACM SIGGRAPH Conference on Computer Graphics*, pages 11–20, Boston, March 1992.

[56] V. Gaede and O. Günther. Multidimensional access methods. *ACM Computing Surveys*, 30(2):170–231, June 1998.

[57] I. Gargantini. An effective way to represent quadtrees. *Communications of the ACM*, 25(12):905–910, December 1982.

[58] Aristides Gionis, Piotr Indyk, and Rajeev Motwani. Similarity search in high dimensions via hashing. In *Proceedings of the International Conference on Very Large Databases*, volume 25, pages 78–89, Edinburgh, Scotland, 1999. Morgan Kaufmann Publishers.

[59] M. T. Goodrich, J.-J. Tsay, D. E. Vengroff, and J. S. Vitter. External-memory computational geometry. In *Proceedings of the IEEE Symposium on Foundations of Computer Science*, volume 34, pages 714–723, Palo Alto, November 1993.

[60] Sathish Govindarajan, Tamás Lukovszki, Anil Maheshari, and Norbert Zeh. I/O-efficient well-separated pair decomposition and its applications. In *Proceedings of the European Symposium on Algorithms*, volume 1879 of *Lecture Notes in Computer Science*, Saarbrücken, Germany, September 2000. Springer Verlag.

[61] D. Greene. An implementation and performance analysis of spatial data access methods. In *Proceedings of IEEE International Conference on Data Engineering*, volume 5, pages 606–615, 1989.

[62] Roberto Grossi and Giuseppe F. Italiano. Efficient cross-trees for external memory. In J. Abello and J. S. Vitter, editors, *External Memory Algorithms and Visualization*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, pages 87–106. American Mathematical Society Press, Providence, RI, 1999.

[63] Roberto Grossi and Giuseppe F. Italiano. Efficient splitting and merging algorithms

for order decomposable problems. *Information and Computation*, in press. An earlier version appears in *Proceedings of the International Colloquium on Automata, Languages and Programming*, volume 1256 of Lecture Notes in Computer Science, Springer Verlag, 605–615, 1997.

[64] Antonin Guttman. R-trees: A dynamic index structure for spatial searching. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 47–57, 1984.

[65] Joseph M. Hellerstein, Elias Koutsoupias, and Christos H. Papadimitriou. On the analysis of indexing schemes. In *Proceedings of the ACM Symposium on Principles of Database Systems*, volume 16, pages 249–256, Tucson, AZ, May 1997.

[66] L. Hellerstein, G. Gibson, R. M. Karp, R. H. Katz, and D. A. Patterson. Coding techniques for handling failures in large disk arrays. *Algorithmica*, 12(2–3):182–208, 1994.

[67] K. H. Hinrichs. *The grid file system: Implementation and case studies of applications.* Ph.d., Dept. Information Science, ETH, Zürich, 1985.

[68] D. Hutchinson, A. Maheshwari, J-R. Sack, and R. Velicescu. Early experiences in implementing the buffer tree. In *Proceedings of the Workshop on Algorithm Engineering*, volume 1, 1997.

[69] D. Pfoser C. S. Jensen and Y. Theodoridis. Novel approaches to the indexing of moving object trajectories. In *Proceedings of the International Conference on Very Large Databases*, volume 26, pages 395–406, Cairo, 2000.

[70] I. Kamel and C. Faloutsos. On packing R-trees. In *Proceedings of the International ACM Conference on Information and Knowledge Management*, volume 2, pages 490–499, 1993.

[71] I. Kamel and C. Faloutsos. Hilbert R-tree: An improved R-tree using fractals. In *Proceedings of the International Conference on Very Large Databases*, volume 20, pages 500–509, 1994.

[72] I. Kamel, M. Khalil, and V. Kouramajian. Bulk insertion in dynamic R-trees. In *Proceedings of the International Symposium on Spatial Data Handling*, volume 4, pages 3B, 31–42, 1996.

[73] P. C. Kanellakis, G. M. Kuper, and P. Z. Revesz. Constraint query languages. In *Proceedings of the ACM Conference on Principles of Database Systems*, volume 9, pages 299–313, 1990.

[74] P. C. Kanellakis, S. Ramaswamy, D. E. Vengroff, and J. S. Vitter. Indexing for data models with constraints and classes. *Journal of Computer and System Sciences*, 52(3):589–612, 1996.

[75] K. V. R. Kanth and A. K. Singh. Optimal dynamic range searching in non-replicating index structures. In *Proceedings of the International Conference on Database Theory*, volume 1540 of *Lecture Notes in Computer Science*, pages 257–276. Springer-Verlag, January 1999.

[76] D. G. Kirkpatrick and R. Seidel. The ultimate planar convex hull algorithm? *SIAM Journal on Computing*, 15:287–299, 1986.

[77] D. E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming.* Addison-Wesley, Reading, MA, 2nd edition, 1998.

[78] G. Kollios, D. Gunopulos, and V. J. Tsotras. On indexing mobile objects. In *Proceedings of the ACM Symposium on Principles of Database Systems*, volume 18, pages 261–272, 1999.

[79] E. Koutsoupias and D. S. Taylor. Tight bounds for 2-dimensional indexing schemes. In *Proceedings of the ACM Symposium on Principles of Database Systems*, volume 17, pages 52–58, Seattle, WA, June 1998.

[80] R. Krishnamurthy and K.-Y. Wang. Multilevel grid files. Tech. report, IBM T. J. Watson Center, Yorktown Heights, NY, November 1985.

[81] K. Küspert. Storage utilization in B*-trees with a generalized overflow technique. *Acta Informatica*, 19:35–55, 1983.

[82] Robert Laurini and Derek Thompson. *Fundamentals of Spatial Information Systems.* Academic Press, 1992.

[83] Philip L. Lehman and S. Bing Yao. Efficient locking for concurrent operations on B-Trees. *ACM Transactions on Database Systems*, 6(4):650–570, December 1981.

[84] W. Litwin and D. Lomet. A new method for fast data searches with keys. *IEEE Software*, 4(2):16–24, March 1987.

[85] D. Lomet. A simple bounded disorder file organization with good performance. *ACM Transactions on Database Systems*, 13(4):525–551, 1988.

[86] D. B. Lomet and B. Salzberg. The hB-tree: A multiattribute indexing method with good guaranteed performance. *ACM Transactions on Database Systems*, 15(4):625–658, 1990.

[87] David B. Lomet and Betty Salzberg. Concurrency and recovery for index trees. *VLDB Journal*, 6(3):224–240, 1997.

[88] E. M. McCreight. Priority search trees. *SIAM Journal on Computing*, 14(2):257–276, May 1985.

[89] C. Mohan. ARIES/KVL: A key-value locking method for concurrency control of multiaction transactions on B-tree indices. In *Proceedings of the International Conference on Very Large Databases*, volume 16, page 392, Brisbane, Australia, August 1990.

[90] J. Nievergelt, H. Hinterberger, and K. C. Sevcik. The grid file: An adaptable, symmetric multi-key file structure. *ACM Transactions on Database Systems*, 9:38–71, 1984.

[91] J. Nievergelt and E. M. Reingold. Binary search tree of bounded balance. *SIAM Journal on Computing*, 2(1), 1973.

[92] J. Nievergelt and P. Widmayer. Spatial data structures: Concepts and design choices. In M. van Kreveld, J. Nievergelt, T. Roos, and P. Widmayer, editors, *Algorithmic Foundations of GIS*, volume 1340 of *Lecture Notes in Computer Science*, page 153 ff. Springer-Verlag, 1997.

[93] Patrick E. O'Neil. The SB-tree. an index-sequential structure for high-performance sequential access. *Acta Informatica*, 29(3):241–265, June 1992.

[94] J. A. Orenstein. Redundancy in spatial databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 294–305, Portland, OR, June 1989.

[95] J. A. Orenstein and T. H. Merrett. A class of data structures for associative searching. In *Proceedings of the ACM Conference on Principles of Database Systems*, volume 3, pages 181–190, 1984.

[96] M. H. Overmars. *The Design of Dynamic Data Structures.* Lecture Notes in Computer Science. Springer-Verlag, 1983.

[97] Jignesh M. Patel and David J. DeWitt. Partition based spatial-merge join. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 259–270, June 1996.

[98] F. P. Preparata and M. I. Shamos. *Computational Geometry.* Springer-Verlag, Berlin, 1985.

[99] S. Ramaswamy and S. Subramanian. Path caching: A technique for optimal external searching. In *Proceedings of the ACM Conference on Principles of Database Systems*, volume 13, pages 25–35, Minneapolis, MN, 1994.

[100] J. Rao and K. Ross. Cache conscious indexing for decision-support in main memory. In Malcolm Atkinson et al., editors, *Proceedings of the International Conference on Very Large Databases*, volume 25, pages 78–89, Los Altos, CA, 1999. Morgan Kaufmann Publishers.

[101] Jun Rao and Kenneth A. Ross. Making B^+-trees cache conscious in main memory. In Weidong Chen, Jeffery Naughton, and Philip A. Bernstein, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 475–486, Dallas, Texas, 2000.

[102] J. T. Robinson. The k-d-b-tree: A search structure for large multidimensional dynamic indexes. In *Proceedings of the ACM Conference on Principles of Database Systems*, volume 1, pages 10–18, 1981.

[103] C. Ruemmler and J. Wilkes. An introduction to disk drive modeling. *IEEE Computer*, pages 17–28, March 1994.

[104] Simonas Šaltenis, Christian S. Jensen, Scott T. Leutenegger, and Mario A. Lopez. Indexing the positions of continuously moving objects. In Weidong Chen, Jeffery Naughton, and Philip A. Bernstein, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 331–342, Dallas, Texas, 2000.

[105] B. Salzberg and V. J. Tsotras. Comparison of access methods for time-evolving data. *ACM Computing Surveys*, 31:158–221, June 1999.

[106] Hanan Samet. *Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS.* Addison-Wesley, 1989.

[107] Hanan Samet. *The Design and Analysis of Spatial Data Structures.* Addison-Wesley, 1989.

[108] V. Samoladas and D. Miranker. A lower bound theorem for indexing schemes and its application to multidimensional range queries. In *Proceedings of the ACM Symposium on Principles of Database Systems*, volume 17, pages 44–51, Seattle, WA, June 1998.

[109] Peter Sanders. Fast priority queues for cached memory. In *Workshop on Algorithm Engineering and Experimentation*, volume 1619 of *Lecture Notes in Computer Science*, pages 312–327. Springer-Verlag, January 1999.

[110] B. Seeger and H.-P. Kriegel. The buddy-tree: An efficient and robust access method for spatial data base systems. In *Proceedings of the International Conference on Very Large Databases*, volume 16, pages 590–601, 1990.

[111] Elizabeth Shriver, Arif Merchant, and John Wilkes. An analytic behavior model for disk drives with readahead caches and request reordering. In *Proceedings of ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems*, pages 182–191, June 1998.

[112] B. Srinivasan. An adaptive overflow technique to defer splitting in b-trees. *The Computer Journal*, 34(5):397–405, 1991.

[113] S. Subramanian and S. Ramaswamy. The P-range tree: A new data structure for range searching in secondary memory. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, volume 6, pages 378–387, 1995.

[114] R. Tamassia and J. S. Vitter. Optimal cooperative search in fractional cascaded data structures. *Algorithmica*, 15(2):154–171, February 1996.

[115] Microsoft's TerraServer online database of satellite images, available on the World-Wide Web at http://terraserver.microsoft.com/.

[116] TIGER/Line (tm). 1992 technical documentation. Technical report, U. S. Bureau of the Census, 1992.

[117] TPIE user manual and reference, 1999. The manual and software distribution are

available on the web at http://www.cs.duke.edu/TPIE/.

[118] J. van den Bercken, B. Seeger, and P. Widmayer. A generic approach to bulk loading multidimensional index structures. In *Proceedings of the International Conference on Very Large Databases*, volume 23, pages 406–415, 1997.

[119] M. van Kreveld, J. Nievergelt, T. Roos, and P. Widmayer, editors. *Algorithmic Foundations of GIS*, volume 1340 of *Lecture Notes in Computer Science*. Springer-Verlag, 1997.

[120] Peter J. Varman and Rakesh M. Verma. An efficient multiversion access structure. *IEEE Transactions on Knowledge and Data Engineering*, 9(3):391–409, May–June 1997.

[121] D. E. Vengroff and J. S. Vitter. Efficient 3-d range searching in external memory. In *Proceedings of the ACM Symposium on Theory of Computing*, volume 28, pages 192–201, Philadelphia, PA, May 1996.

[122] Darren Erik Vengroff and Jeffrey Scott Vitter. I/O-efficient scientific computation using TPIE. In *Proceedings of NASA Goddard Conference on Mass Storage Systems*, volume 5, pages II, 553–570, September 1996.

[123] J. S. Vitter. Efficient memory access in large-scale computation. In *Proceedings of the Symposium on Theoretical Aspects of Computer Science*, volume 480 of *Lecture Notes in Computer Science*, pages 26–41. Springer-Verlag, 1991. Invited paper.

[124] J. S. Vitter. External memory algorithms and data structures: Dealing with MASSIVE DATA. *ACM Computing Surveys*, 33(2):209–271, June 2001.

[125] J. S. Vitter and E. A. M. Shriver. Algorithms for parallel memory I: Two-level memories. *Algorithmica*, 12(2–3):110–147, 1994.

[126] J. S. Vitter and D. E. Vengroff. Notes, 1999.

[127] D.E. Willard and G.S. Lueker. Adding range restriction capability to dynamic data structures. *Journal of the ACM*, 32(3):597–617, 1985.

[128] Ouri Wolfson, Prasad Sistla, Bo Xu, Jutai Zhou, and Sam Chamberlain. DOMINO: Databases fOr MovINg Objects tracking. In Alex Delis, Christos Faloutsos, and Shahram Ghandeharizadeh, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 547–549, May 1999.

[129] A. C. Yao. On random 2-3 trees. *Acta Informatica*, 9:159–170, 1978.

[130] Stanley B. Zdonik and David Maier, editors. *Readings in Object-Oriented Database Systems*. Morgan Kauffman, 1990.

[131] B. Zhu. Further computational geometry in secondary memory. In *Proceedings of the International Symposium on Algorithms and Computation*, volume 834 of *Lecture Notes in Computer Science*, page 514 ff. Springer-Verlag, 1994.