27 Spatial access methods

P VAN OOSTEROM

This chapter first summarises why spatial access methods are needed. It is important to note that spatial access methods are not only useful for spatial data. Some early main memory spatial access methods are described (section 2), followed by an overview of space-filling curves (section 3). As it is impossible to present all of the spatial access methods described in the recent literature, only the following characteristic families are presented: quadtree, grid-based methods, and R-tree (sections 4–6). Special attention is paid to spatial access methods taking multiple scales into account (section 7). Besides the theory of spatial access methods, the issue of using them in a database in practice is treated in the conclusion of this chapter (section 8).

1 WHY ARE SPATIAL ACCESS METHODS NEEDED?

The main purpose of spatial access methods is to support efficient selection of objects based on spatial properties. For example, a range query selects objects lying within specified ranges of coordinates; a nearest neighbour query finds the object lying closest to a specified object (see Worboys, Chapter 26). Further, spatial access methods are also used to implement efficiently such spatial analyses as map overlay, and other types of spatial joins. Two characteristics of spatial datasets are that they are frequently large and that the data are quite often distributed in an irregular manner. A spatial access method needs to take into account both spatial indexing and clustering techniques. Without a spatial index, every object in the database has to be checked to see whether it meets the spatial selection criterion; a 'full table scan' in a relational database. As spatial datasets are usually very large, such checking is unacceptable in practice for interactive use and most other applications. Therefore, a spatial index is required, in order to find the required objects efficiently without looking at every object. In cases when the whole spatial dataset resides in main memory it is sufficient to know the addresses of the requested objects, as main memory storage allows random access and does not introduce significant delays. However, most spatial datasets are so large that they cannot reside in the main memory of the computer and must be stored in secondary memory, such as its hard disk. Clustering is needed to group those objects which are often requested together. Otherwise, many different disk pages will have to be fetched, resulting in slow response. In a spatial context, clustering implies that objects which are close together in reality are also stored close together in memory. Many strategies for clustering objects in spatial databases adopt some form of 'space-filling curve' by ordering objects according to their sequence along a path that traverses all parts of the space.

In traditional database systems, sorting (or ordering) of the data forms the basis for efficient searching, as in the B-tree approach (Bayer and Creight 1973). Although there are obvious bases for sorting text strings, numbers, or dates (1-dimensional data), there are no such simple solutions for sorting higher-dimensional spatial data. Computer memory is 1-dimensional but spatial data is 2-dimensional or 3-dimensional (or even higher dimensioned), and must be organised somehow in memory. An intuitive solution is to use a regular grid just as on a paper map. Each grid cell has a unique name, e.g. 'A3', 'C6', or 'D5'. The cells are stored in some order in memory and can each contain a (fixed) number of object references. In a grid cell, a reference is stored to an object whenever the object (partially) overlaps the cell. However, this will not be very efficient because of the irregular data distribution of spatial data: many cells will be empty (e.g. in the ocean), while many other cells will be over-full (e.g. in the city centre). Therefore, more advanced techniques have been developed.

2 MAIN MEMORY ACCESS METHODS

Though originally not designed for handling very large datasets, main memory data structures show several interesting techniques with respect to handling spatial data. In this section the KD-tree (adaptive, bintree) and the BSP-tree will be illustrated.

2.1 The KD-tree

The basic form of the KD-tree stores *K*-dimensional points (Bentley 1975). This section concentrates on the 2-dimensional case. Each internal node of the KD-tree contains one point and also corresponds to a rectangular region. The root of the tree corresponds to the whole region of interest. The rectangular region is divided into two parts by the *x*-coordinate of the stored point on the odd levels and by the *y*-coordinate on the even levels in the tree; see Figure 1. A new point is inserted by descending the tree until a leaf node is reached. At each internal node the value of the proper coordinate of the stored point is compared with the corresponding coordinate of the new point and the proper path is chosen. This continues until a leaf node is reached. This leaf also represents a rectangular region, which in turn will be divided into two parts by the new point. The insertion of a new point results in one new internal node. Range searching in the KD-tree starts at the root, checks whether the stored node (split point) is included in the search range and whether there is overlap with the left or right subtree. For each subtree which overlaps the search region, the procedure is repeated until the leaf level is reached.

A disadvantage of the KD-tree is that the shape of the tree depends on the order in which the points are inserted. In the worst case, a KD-tree of *n* points has *n* levels. The adaptive KD-tree (Bentley and Friedman 1979) solves this problem by choosing a splitting point (which is not an element of the input set of data points), which divides the set of points into two sets of (nearly) equal size. This process is repeated until each set contains one point at most; see Figure 2. The adaptive KD-tree is not dynamic: it is hard to insert or delete points while keeping the tree balanced. The adaptive KD-tree for *n* points can be built in O(*n* log *n*) time and takes O(*n*) space for *K*=2. A range query takes O(sqrt(*n*)+*t*) time in two dimensions where *t* is the number of points found. Another variant of the KD-tree is the bintree (Tamminen 1984). Here the space is divided into two equal-sized rectangles instead of two rectangles with equal numbers of points. This is repeated until each leaf contains one point at the most.

A modification that makes the KD-tree suitable for secondary memory is described by Robinson (1981) and is called the KDB-tree. For practical use, it is more convenient to use leaf nodes containing more than one data point. The maximum number of points that a leaf may contain is called the 'bucket size'. The bucket size is chosen in such a way that it fits within one disk page. Moreover, internal nodes are grouped and each group is stored on one page in order to minimise the number of disk accesses. Robinson describes algorithms for deletions and insertions under which the KDB-tree remains balanced. Unfortunately, no reasonable upper bound for memory usage can be guaranteed.

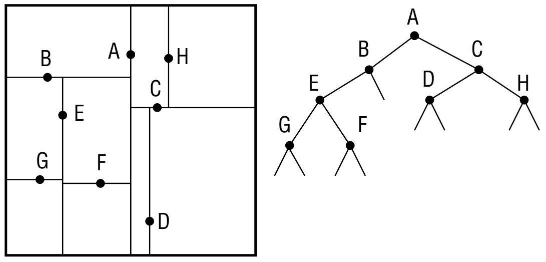


Fig 1. The KD-tree.

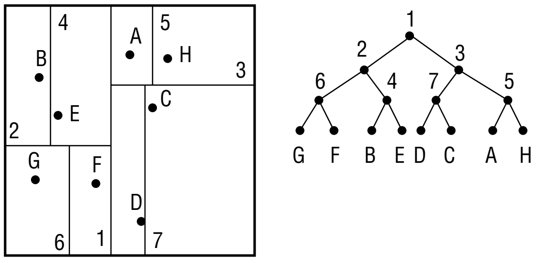


Fig 2. The adaptive KD-tree.

Matsuyama et al (1984) show how the geometric primitives polyline and polygon may be incorporated using the centroids of bounding boxes in the 2-D-tree. Rosenberg (1985) uses a 4-D-tree to store a bounding box by putting the minimum and maximum points together in one 4-dimensional point. This technique can be used to generalise other geometric data structures that are originally suited only for storing and retrieving points. The technique works well for exact-match queries, but is often more complicated in the case of range queries. In general, geometrically close 2-dimensional rectangles do not map into geometrically close 4-dimensional points (Hutflesz et al 1990). The ranges are transformed into complex search regions in the higher-dimensional space, which in turn results in slow query responses.

2.2 The BSP-tree

We begin here by describing the binary space partitioning (BSP)-tree, before presenting a variant suitable for GIS applications: the multi-object BSP-tree for storing polylines and polygons. The original use of the BSP-tree was in 3-dimensional computer graphics (Fuchs et al 1980; Fuchs et al 1983). The BSP-tree was used by Fuchs to produce a hidden surface image of a static 3-dimensional scene. After a preprocessing phase it is possible to produce an image from any view angle in O(n) time, with n the number of polygons in the BSP-tree.

In this chapter the 2-dimensional BSP-tree is used for the structured storage of geometric data. It is a data structure that is not based on a rectangular division of space. Rather, it uses the line segments of the polylines and the edges of the polygons to divide the space in a recursive manner. The BSP-tree reflects this recursive division of space. Each time a (sub)space is divided into two subspaces by a so-called splitting primitive, a corresponding node is added to the tree. The BSP-tree represents an organisation of space by a set of convex subspaces in a binary tree. This tree is useful during spatial search and other spatial operations. Figure 3(a) shows a scene with some directed line segments. The 'left' side of the line segment is marked with an arrow. From this scene, line segment A is selected and space is split into two parts by the supporting line of A. This process is repeated for each of the two sub-spaces with the other line segments. The splitting of space continues until there are no line segments left. Note that sometimes the splitting of a

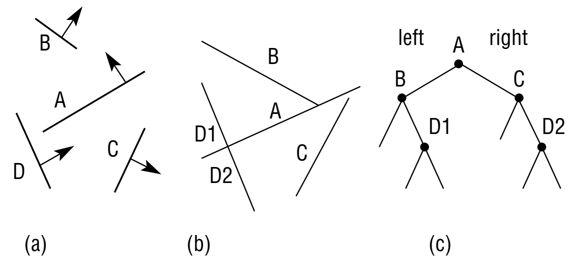


Fig 3. The building of a BSP-tree: (a) 2-dimensional scene; (b) convex sub-spaces; and (c) BSP-tree.

space implies that a line segment (which has not yet been used for splitting itself) is split into two parts. Line D, for example, is split into D1 and D2. Figure 3(b) shows the resulting organisation of the space, as a set of (possibly open) convex subspaces. The corresponding BSP-tree is drawn in Figure 3(c).

The BSP-tree, as discussed so far, is suitable only for storing a collection of (unrelated) line segments. In GIS it must be possible to represent objects, such as polygons. The multi-object BSP-tree (Oosterom 1990) is an extension of the BSP-tree which caters for object representation. It stores the line segments that together make up the boundary of the polygon. The multi-object BSP-tree has explicit leaf nodes which correspond to the convex subspaces created by the BSP-tree. Figure 4(a) presents a 2-dimensional scene with two objects, triangle T with sides abc, and rectangle R with sides defg. The method divides the space in the convex subspaces of Figure 4(b). The BSP-tree of Figure 4(c) is extended with explicit leaf nodes, each representing a convex part of the space. If a convex subspace corresponds to the 'outside' region, no label is drawn in the figure. If no more than one identification tag per leaf is allowed, only mutually exclusive objects can be stored in the multi-object BSP-tree, otherwise it would be possible also to deal with objects that overlap. A disadvantage of this

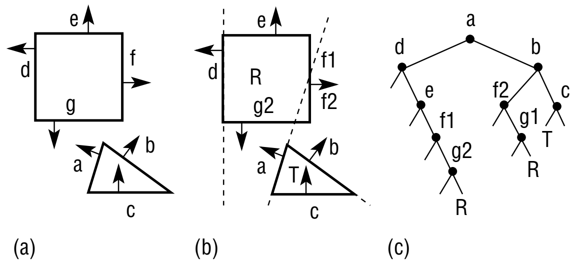


Fig 4. The building of a multi-object BSP-tree: (a) object scenes; (b) convex sub-spaces; and (c) multi-object BSP-tree.

BSP-tree is that the representation of one object is scattered over several leaves, as illustrated by rectangle R in Figure 4. The (multi-object) BSP-tree allows efficient implementation of spatial operations, such as pick and rectangle search.

The choice of which line segment to use for dividing the space very much influences the building of the tree. It is preferable to have a balanced BSP-tree with as few nodes as possible. This is a very difficult requirement to fulfil, because balancing the tree requires that line segments from the middle of the dataset be used to split the space. These line segments will probably split other line segments. Each split of a line segment introduces an extra node in the BSP-tree. However, Paterson and Yao (1989) prove that, if the original line segments are disjoint, then it is possible to build a BSP-tree with $O(n \log n)$ nodes and depth $O(\log n)$ using an algorithm requiring only $O(n \log n)$ time.

3 SPACE-FILLING CURVES

This section presents an overview and some properties of space-filling curves. Space-filling curves order the points in a discrete 2-dimensional space. This technique is also called tile indexing. It transforms a 2-dimensional problem into a 1-dimensional one, so it can be used in combination with a well known data structure for 1-dimensional storage and retrieval, such as the B-tree (Bayer and McCreight 1973). This presentation is based on several papers (Abel and Mark 1990; Goodchild and Grandfield 1983; Jagadish 1990; Nulty and Barholdi 1994) and the book by Samet (1989).

Row ordering simply numbers the cells row by row, and within each row the points are numbered from left to right; see Figure 5(a). Row-prime (or snake like, or boustrophedon) ordering is a variant in which alternate rows are traversed in opposite directions; see Figure 5(b). Obvious variations are column and column-prime orderings in which the roles of row and column are transposed. Bitwise interleaving of the two coordinates results in a 1-dimensional key, called the Morton key (Orenstein and Manola 1988). The Morton key is also known as the Peano key, or N-order, or Z-order. For example, row 2 = 10_{bin} column 3 = 11_{bin} has Morton key 14 = 1110_{bin}; see Figure 5(c). Hilbert ordering is based on the classic Hilbert-Peano curve, as drawn in Figure 5(d). Gray ordering is obtained by bitwise interleaving the Gray codes of the x and y coordinates. As Gray codes have the property that successive codes differ in exactly one bit position, a 4-neighbour cell differs only in one bit; see Figure 5(e) (Faloutsos 1988). In Figure 5(f) the Cantor-diagonal ordering is shown. Note that the numbering of the points is adapted to the fact that we are dealing with a space that is bounded in all directions; for example, row 3 column 1 has order number 10 instead of 11 and row 3 column 3 has order number 15 instead of 24. Spiral ordering is depicted in Figure 5(g). Finally, Figure 5(h) shows the Sierpinski curve, which is based on a recursive triangle subdivision.

Figure 6 shows the geometric construction of the Peano curve: at each step of the refinement each vertex of the basic curve is replaced by the previous order curve. A similar method, but now also including reflection, is used to construct the reflected binary Gray curve; see Figure 7. The Hilbert curve is constructed by rotating the previous order curves at vertex 0 by –90 degrees and at vertex 3 by 90 degrees; see Figure 8. The Sierpinski curve starts with two triangles; each triangle is split into two new triangles and this is repeated until the required resolution is obtained. Note that the orientation and ordering of the triangles is important; see Figure 9.

Abel and Mark (1990) have identified the following desirable qualitative properties of spatial orderings:

- An ordering is continuous if, and only if, the cells in every pair with consecutive keys are 4-neighbours.
- An ordering is quadrant-recursive if the cells in any valid sub-quadrant of the matrix are assigned a set of consecutive integers as keys.
- An ordering is monotonic if, and only if, for every fixed x, the keys vary monotonically with y in some particular way, and vice versa.
- An ordering is stable if the relative order of points is maintained when the resolution is doubled.

Ordering techniques are very efficient for exact-match queries for points, but there is quite a difference in their efficiency for other types of geometric queries, for example range queries. Abel and Mark (1990) conclude from their practical comparative analysis of five orderings (they do not consider the Cantor-diagonal, the spiral and the Sierpinski orderings) that the Morton ordering and the Hilbert ordering are, in general, the best options. Some quantitative properties of curves are: the total length of the curve, the variability in unit lengths (path between two cells next in order), the average of

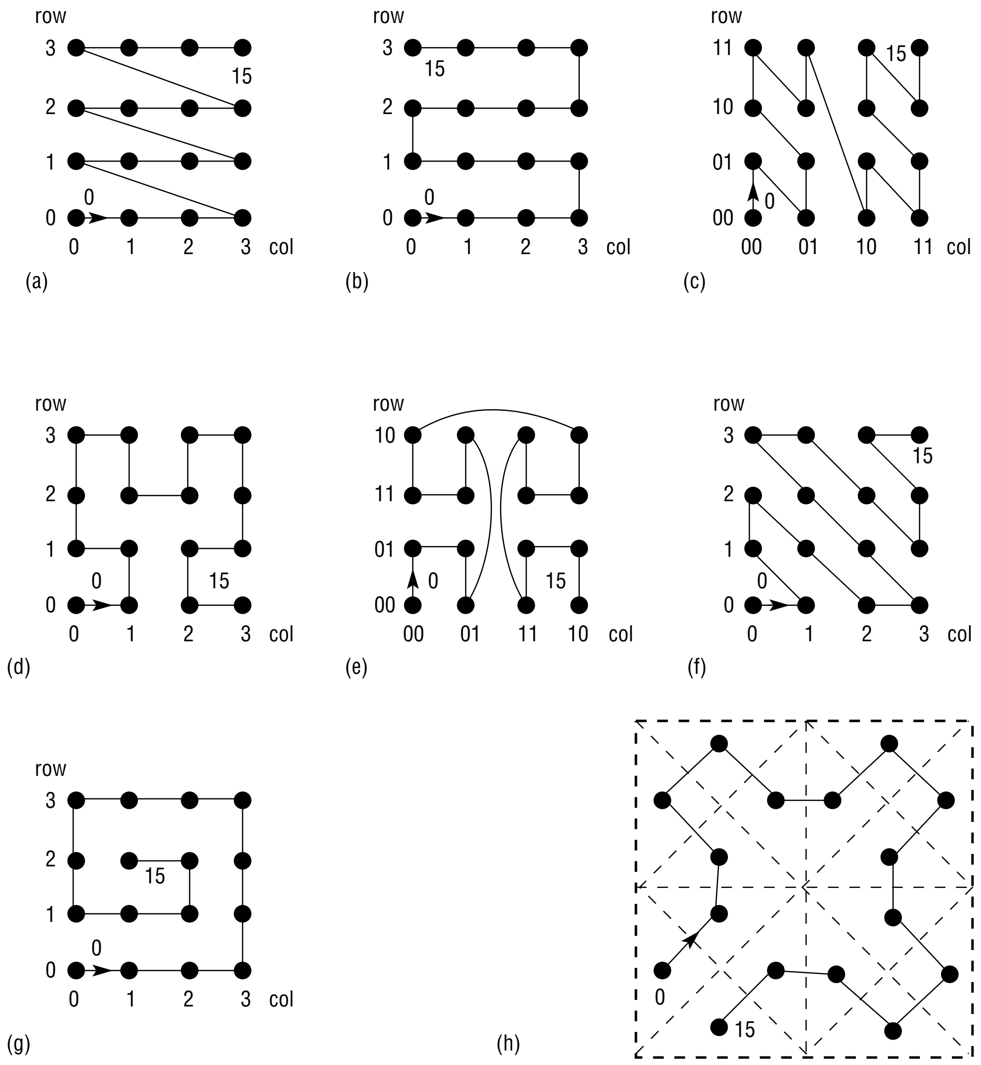


Fig 5. Eight different orderings: (a) row; (b) row prime; (c) Peano; (d) Hilbert; (e) Gray; (f) Cantor-original; (g) spiral; and (h) Sierpinski (triangle).

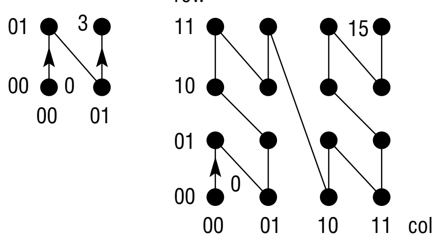


Fig 6. Geometric construction of the Peano curve.

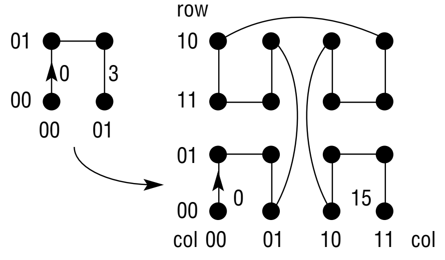


Fig 7. Geometric construction of the Gray curve.

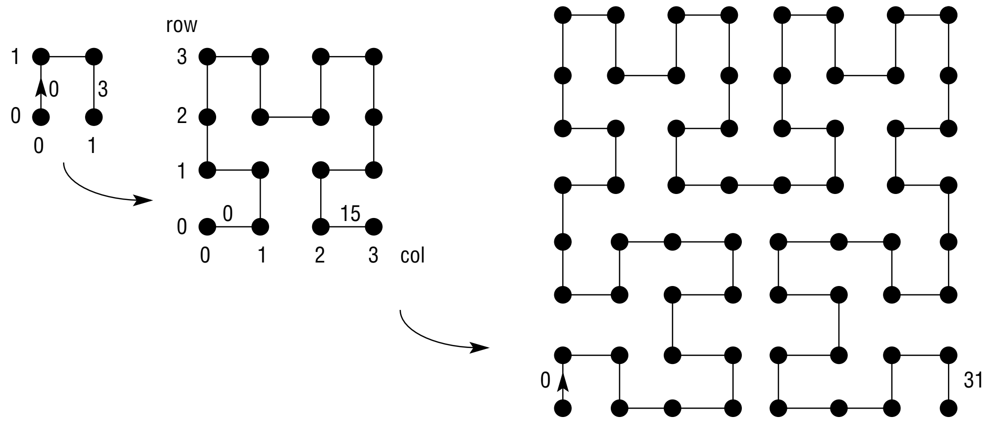


Fig 8. Geometric construction of the Hilbert curve.

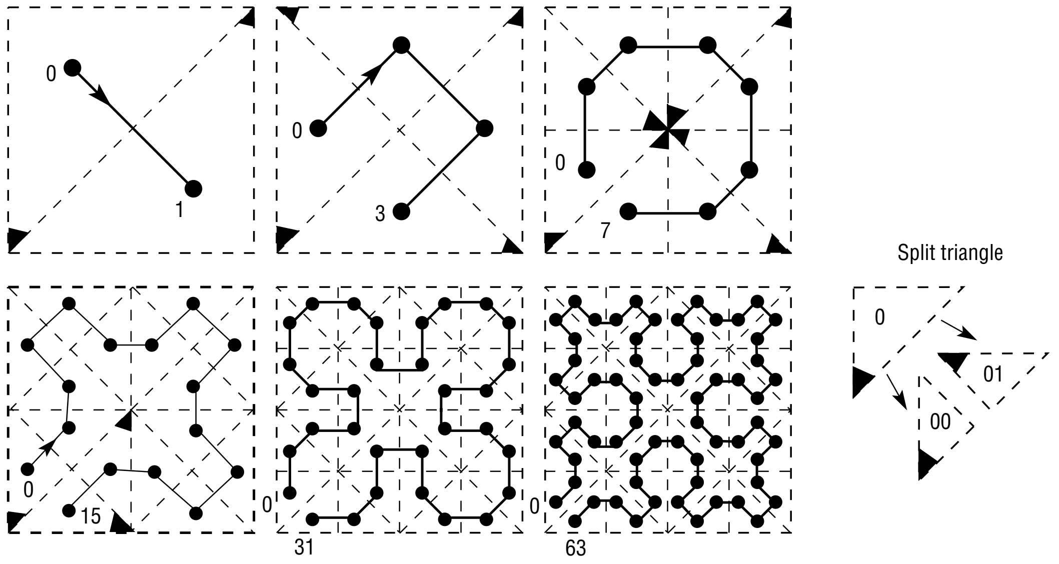


Fig 9. Geometric construction of the Sierpinski curve.

the average distance between 4-neighbours, and the average of the maximum distance between 4-neighbours. Goodchild (1989) proved that the expected difference of 4-neighbour keys of an *n* by *n* matrix is (*n*+1)/2 for Peano, Hilbert, row and row-prime orderings, indicating that this is not a very discriminating property. Therefore, Faloutsos and Roseman (1989) suggest a better measure for spatial clustering: the average (Manhattan) maximum distance of all cells within *N*/2 key value of a given cell on a *N* by *N* grid; see Table 1. Another measure for clustering is the average number of clusters for all possible range queries. Note that a cluster is defined as a group of cells with consecutive key value; see Table 2 and Figure 10.

Table 1 Average Manhatten maximum distance of cell within N/2 key value (after Faloutsos and Roseman 1989).

grid	*Hilbert*	*Gray*	*Peano*
2*2	1.00	1.00	1.50
4*4	2.00	2.75	2.75
8*8	3.28	5.00	4.84
16*16	4.89	8.52	7.91

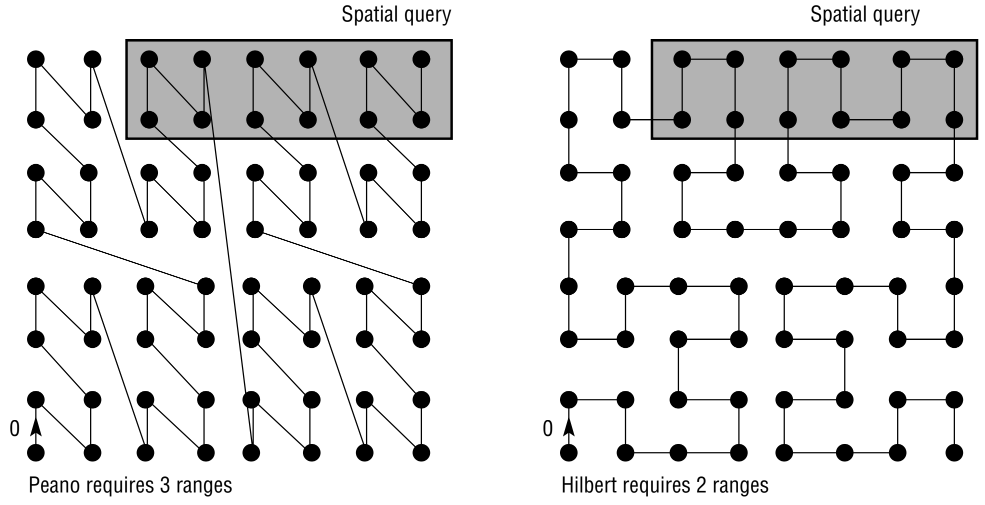


Fig 10. Number of clusters for a given range query.

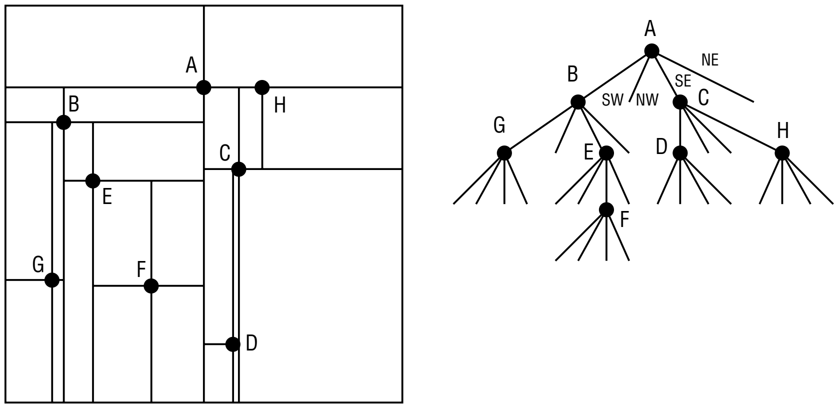


Fig 11. The point quadtree.

Table 2 Average number of clusters for all possible range queries. (After Faloutsos and Roseman 1989.)

grid	*Hilbert*	*Gray*	*Peano*
2*2	1.11	1.11	1.22
4*4	1.64	1.92	2.16
8*8	2.93	4.02	4.41
16*16	5.60	8.71	9.29

4 THE QUADTREE FAMILY

The quadtree is a generic name for all kinds of trees that are built by recursive division of space into four quadrants. Several different variants have been described in the literature, of which the following will be presented here: point quadtree, PR (point region) quadtree, region quadtree, and PM (polygonal map) quadtree. Samet (1984, 1989) gives an excellent overview.

4.1 Point quadtree and PR quadtree

The point quadtree resembles the KD-tree described in section 2. The difference is that the space is divided into four rectangles instead of two; see Figure 11. The input points are stored in the internal nodes of the tree. The four different rectangles are typically referred to as SW (southwest), NW (northwest), SE (southeast), and NE (northeast).

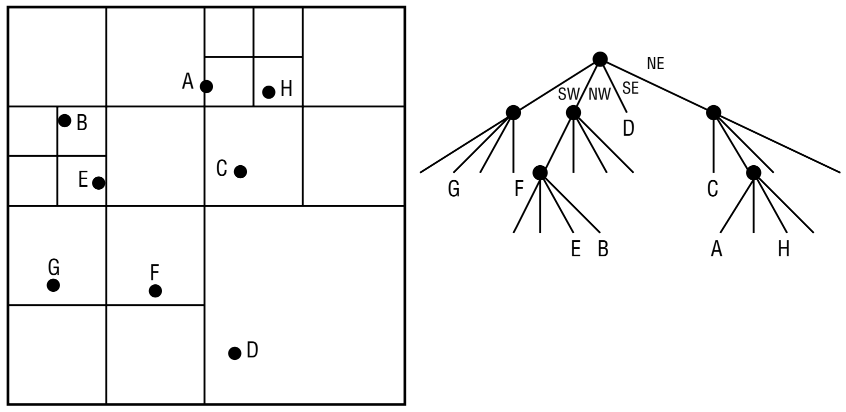


Fig 12. The PR quadtree.

Searching in the quadtree is very similar to the KD-tree: whenever a point is included in the search range it is reported and whenever a subtree overlaps with the search range it is traversed.

A minor variant of the point quadtree is the PR quadtree, which does not use the points of the data set to divide the space. Every time it divides the space, a square, into four equal subsquares, until each contains no more than the given bucket size (e.g. one object). Note that dense data regions require more partitions and therefore the quadtree will not be balanced in this situation; see Figure 12.

4.2 Region quadtree

A very well known quadtree is the region quadtree, which is used to store a rasterised approximation of a polygon. First, the area of interest is enclosed by a square. A square is repeatedly divided into four squares of equal size until it is completely inside (a black leaf) or outside (a white leaf) the polygon or until the maximum depth of the tree is reached (dominant colour is assigned to the leaf); see Figure 13. The main drawback is that it does not contain an exact representation of the polygon. The same applies if the

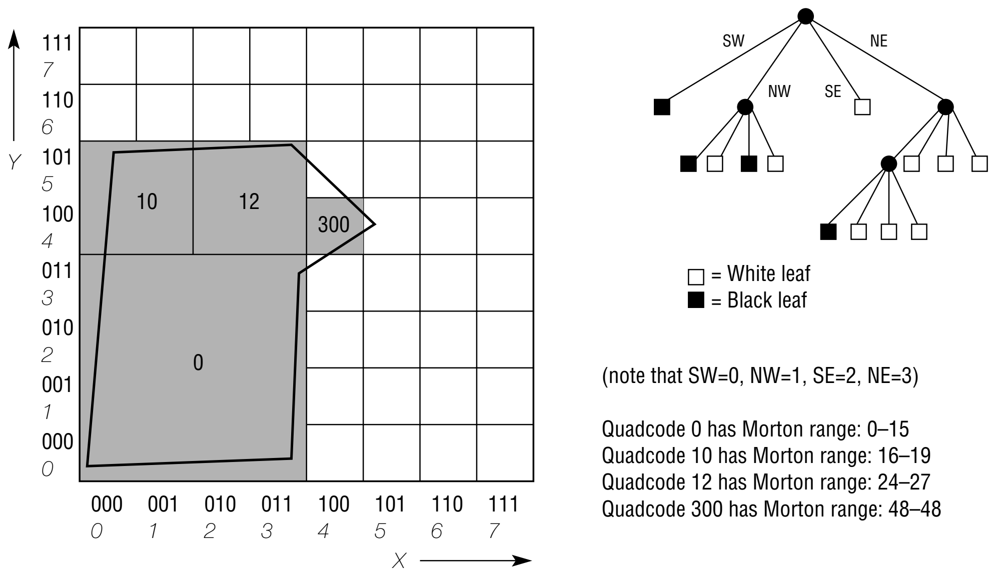

111
7
110
6
Y
101
5
100
4
011
3
010
2
001
1
000
0
10
12
300
0
000 001 010 011 100 101 110 111
0 1 2 3 4 5 6 7
X
SW
NE
NW
SE
□ = White leaf
■ = Black leaf
(note that SW=0, NW=1, SE=2, NE=3)
Quadcode 0 has Morton range: 0–15
Quadcode 10 has Morton range: 16–19
Quadcode 12 has Morton range: 24–27
Quadcode 300 has Morton range: 48–48

Fig 13. The region quadtree.

region quadtree is used to store points and polylines. This kind of quadtree is useful for storing raster data.

4.3 PM quadtree

A polygonal map, a collection of polygons, can be represented by the PM quadtree. The vertices are stored in the tree in the same way as in the PR quadtree. The edges are segmented into q-edges which completely fall within the squares of the leaves. There are seven classes of q-edges. The first are those that intersect one boundary of the square and meet at a vertex within that square. The other six classes intersect two boundaries and are named after the boundaries they intersect: NW, NS, NE, EW, SW, and SE. For each non-empty class, the q-edges are stored in a balanced binary tree. The first class is ordered by an angular measure and the other six classes are ordered by their intercepts along the perimeter. Figure 14 shows a polygonal map and the corresponding PM quadtree. The PM quadtree provides a reasonably efficient data structure for performing various operations: inserting an edge, point-in-polygon testing, overlaying two maps, range searching and windowing.

The extensive research efforts on quadtrees in the last decade resulted in more variants and algorithms to manipulate these quadtrees efficiently. For example, the CIF quadtree is particularly suited for rectangles. Another interesting example is the linear quadtree: in this representation there is no explicit quadtree, but only an enumeration of the quadcodes belonging to the object; e.g. 0, 10, 12, and 300 in Figure 13. Rosenberg (1985) and Samet (1984) describe these variants.

5 GRID-BASED METHODS

The intuitively attractive approach of organising space by imposing a regular grid has been refined in several different ways, in order to avoid the problems of dealing with irregular distributed data. In this section two different approaches are described: the grid file and the field-tree.

5.1 The grid file

The principle of a grid file is the division of a space into rectangles (regular tiles, grids, squares, cells) that can be identified by two indices, one for the *x*-direction and the other for the *y*-direction. The grid file is a non-hierarchical structure. The geometric primitives are stored in the grids, which are not necessarily of equal size. There are several variants of this technique. In this subsection the file structure as defined by Nievergelt et al (1984) is described.

The advantage of the grid file as defined by Nievergelt et al is that, unlike most other grid files, it adjusts itself to the density of the data: however, it is also more complicated. The cell division lines need not be equidistant; for each of *x* and *y* there is a 1-dimensional array (in main memory) giving the actual sizes of the cells. Neighbouring cells may be joined into one bucket if the resulting area is a rectangle. The buckets have a fixed size and are stored on a disk page. The grid directory is a 2-dimensional array, with a pointer for each cell to the correct bucket. Figure 15 shows a grid file with linear scales and grid directory. The grid file has good dynamic properties. If a bucket is too full to store a new primitive and it is used for more than one cell, then the bucket may be divided into two buckets. This is a

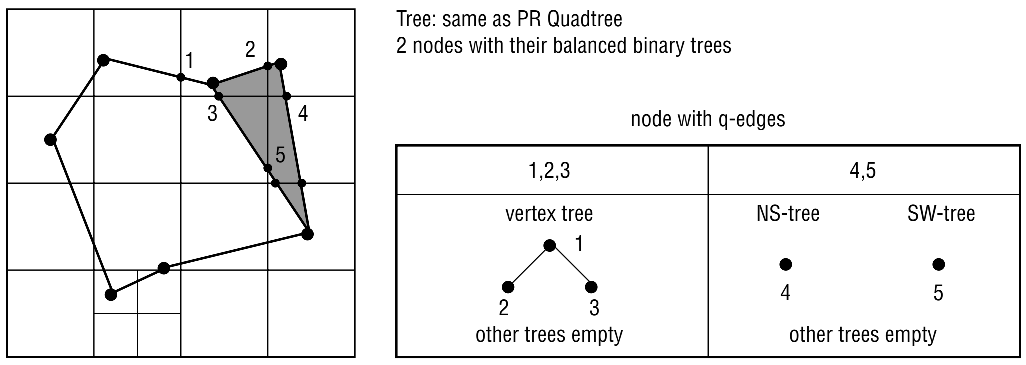


Fig 14. The PM quadtree.

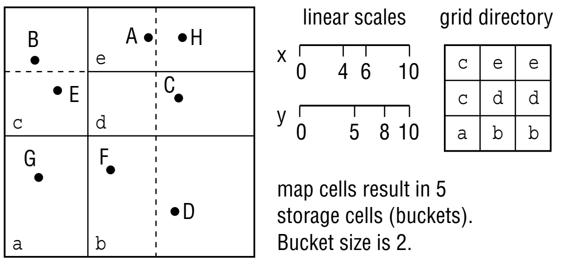


Fig 15. The grid file.

minor operation. If the bucket is used for only one cell, then a division line is added to one of the linear scales. This is a little more complex but is still a minor operation. In the case of a deletion of primitives, the merging process is performed in a manner analogous to the splitting process for insertion.

5.2 The field-tree

The field-tree is suited to store points, polylines, and polygons in a non-fragmented manner. Several variants of the field-tree have been published (Frank 1983; Frank and Barrera 1989; Kleiner and Brassel 1986). In this subsection attention will be focused on the partition field-tree. Conceptually, the field-tree consists of several levels of grids, each with a different resolution and a different displacement/origin; see Figure 16. A grid cell is called a field. The field-tree is not, in fact, a hierarchical tree, but a directed acyclic graph, as each field can have one, two, or four ancestors. At one level the fields form a partition and therefore never overlap. In another variant, the cover field-tree (Frank and Barrera 1989), the fields may overlap. It is not necessary at each level that the entire grid be explicitly present as fields.

A newly inserted object is stored in the smallest field in which it completely fits (unless its importance requires it to be stored at a higher level). As a result of the different displacements and grid resolutions, an object never has to be stored more than three levels above the field size that corresponds to the object size. Note that this is not the case in a quad-tree-like structure, because here the edges at different levels are collinear. The insertion of a new object may cause a field to become too full. In this case an attempt is made to create one or more new descendants and to reorganise the field by moving objects down. This is not always possible. A drawback of the field-tree is that an overflow page is sometimes required, as it is not possible to move relatively large or important objects from an over-full field to a lower level field.

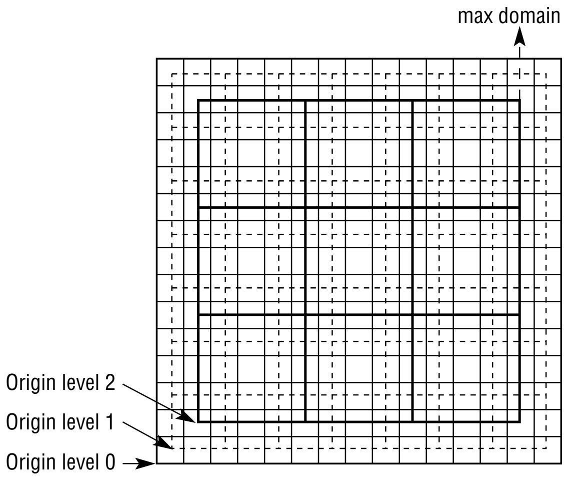


Fig 16. The positioning of geometric objects in the field-tree.

6 THE R-TREE FAMILY

Instead of dividing space in some manner, it is also possible to group the objects in some hierarchical organisation based on (a rectangular approximation of) their location. This is the approach of the R-tree and in this section several variants are also described: the R+-tree, R*-tree, Hilbert R-tree and sphere-tree.

6.1 The R-tree

The R-tree is an index structure that was defined by Guttman in 1984. The leaf nodes of this multiway tree contain entries of the form (I, object-id), where object-id is a pointer to a data object and I is a bounding box (or an axes-parallel minimal bounding rectangle, MBR). The data object can be of any type: point, polyline, or polygon. The internal nodes contain entries of the form (I, child-pointer), where child-pointer is a pointer to a child and I is the MBR of that child. The maximum number of entries in each node is called the branching factor M and is chosen to suit paging and disk I/O buffering. The Insert and Delete algorithms assure that the number of entries in each node remains between m and M, where $m \leq \lceil M/2 \rceil$ is the minimum number of entries per node. An advantage of the R-tree is that pointers to complete objects (e.g. polygons) are stored, so the objects are never fragmented.

When inserting a new object, the tree is traversed from the root to a leaf choosing each time the child which needs the least enlargement to enclose the object. If there is still space, then the object is stored in that leaf. Otherwise, the leaf is split into two leaves. The entries are distributed among the two leaves in order to try to minimise the total area of the two leaves. A new leaf may cause the parent to become over-full, so it has to be split also. This process may be repeated up to the root. During the reverse operation, delete, a node may become under-full. In this situation the node is removed (all other entries are saved and reinserted at the proper level later on). Again, this may cause the parent to become under-full, and the same technique is applied at the next level. This process may have to be repeated up to the root. Of course, the MBRs of all affected nodes have to be updated during an insert or a delete operation.

Figure 17 shows an R-tree with two levels and $M = 4$. The lowest level contains three leaf nodes and the highest level contains one node with pointers and MBRs of the leaf nodes. Coverage is defined as the total area of all the MBRs of all leaf R-tree nodes, and overlap is the total area contained within two or more leaf MBRs (Faloutsos et al 1987). In Figure 17 the coverage is $A \cup B \cup C$ and the overlap is $A \cap B$. It is clear that efficient searching demands both low coverage and low overlap.

6.2 Some R-tree variants

Roussopoulos and Leifker (1985) describe the Pack algorithm which creates an initial R-tree that is more efficient than the R-tree created by the Insert algorithm. The Pack algorithm requires all data to be known *a priori*. The R+-tree (Faloutsos et al 1987), a modification of the R-tree, avoids overlap at the expense of more nodes and multiple references to some objects; see Figure 18. Therefore, point queries always correspond to a single-path tree traversal. A drawback of the R+-tree is that no minimum space utilisation per node can be given. Analytical results indicate that R+-trees allow more efficient searching, in the case of relatively large objects.

The R*-tree (Beckmann et al 1990) is based on the same structure as the R-tree, but it applies a different Insert algorithm. When a node overflows, it is not split right away, but first an attempt is made to remove *p* entries and reinsert these in the tree. The parameter *p* can vary. In the original paper it is suggested that *p* be set to be 30 per cent of the maximum number of entries per node. In some cases this will solve the node overflow problem without splitting the node. In general this will result in a fuller tree. However, this reinsert technique will not always solve the problem and sometimes a real node split is required. Instead of only minimising the total area, an attempt is also made to minimise overlap between the nodes, and to make the nodes as square as possible.

The Hilbert R-tree uses the centre point Hilbert value of the MBR to organise the objects (Kamel and Faloutsos 1994). When grouping objects (based on their Hilbert value), they form an entry in their parent node which contains both the union of all MBRs of the objects and the largest Hilbert value of the objects. Again, this is repeated on the higher levels until a single root is obtained. Inserting and deleting is basically done using the (largest) Hilbert value and applying B-tree (Bayer and McCreight 1973) techniques. Searching is done using the MBR and applying R-tree techniques. The B-tree technique makes it possible to get fuller nodes, because '2-to-3' or '3-to-4' (or higher) split policies can be used. This results in a more compact tree,

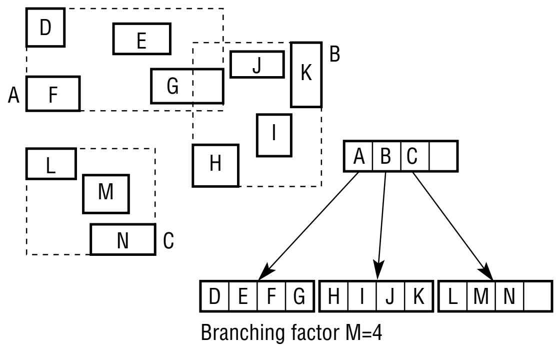


Fig 17. The R-tree.

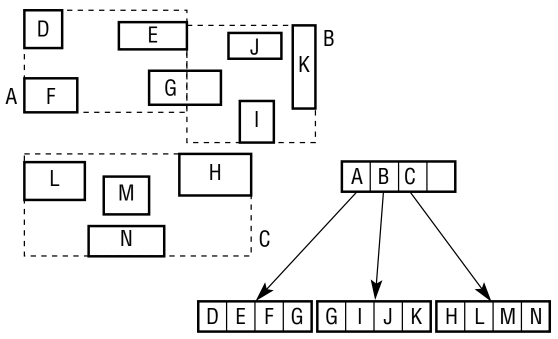


Fig 18. The R+-tree.

which is again beneficial for performance. The drawback of the Hilbert R-tree is that the real spatial aspects of the objects are not used to organise them, but instead their Hilbert values. It is possible that two objects which are very close in reality have Hilbert values which are very different. Therefore, these two objects will not end up in the same node in spite of the fact that they are very close; instead each of them may be grouped with other objects (further away in reality, but with closer Hilbert value). This will result in larger MBRs and therefore reduced performance.

The sphere-tree (Oosterom and Claassen 1990) is very similar to the R-tree (Guttman 1984), with the exception that it uses minimal bounding circles (or spheres in higher dimensions, MBSs) instead of MBRs; see Figure 19. Besides being orientation-insensitive, the sphere-tree has the advantage over the R-tree in that it requires less storage space. The operations on the sphere-tree are very similar to those on the R-tree, with the exception of the computation of the minimal bounding circle, which is more difficult (Elzinga and Hearn 1972; Megiddo 1983; Sylvester 1857).

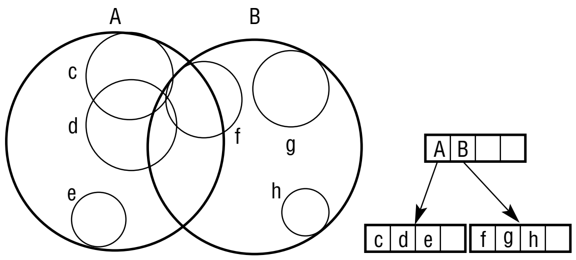


Fig 19. The sphere-tree.

7 MULTISCALE SPATIAL ACCESS METHODS

Interactive GIS applications can be supported even better if importance (resolution, scale) is taken into account in addition to spatial location when designing access methods (see also Weibel and Dutton, Chapter 10). Think of a user who is panning and zooming in a certain dataset. Just enlarging the objects when the user zooms in will result in a poor map. Not only must the objects be enlarged, but they must be displayed with more detail (because of the higher resolution), and less significant objects must also be displayed. A simple solution is to store the map at different scales (or levels of detail). This would introduce redundancy with all the related drawbacks of possible inconsistency and increased memory usage. Therefore, geographical data should be stored in an integrated manner without redundancy and, if required, be supported by a special data structure. Detail levels are closely related to cartographic map generalisation techniques. Besides being suited for map generalisation, these multiscale data structures must also provide spatial properties; e.g. it must be possible to find all objects within a specified region efficiently. The name of these types of data structures is reactive data structures (Oosterom 1989, 1991, 1994).

The simplification part of the generalisation process is supported by the binary line generalisation tree (Oosterom and Bos 1989) based on the Douglas–Peucker algorithm (Douglas and Peucker 1973); see Figure 20. The reactive-tree (Oosterom 1991) is a spatial index structure that also takes care of the selection part of generalisation. The reactive-

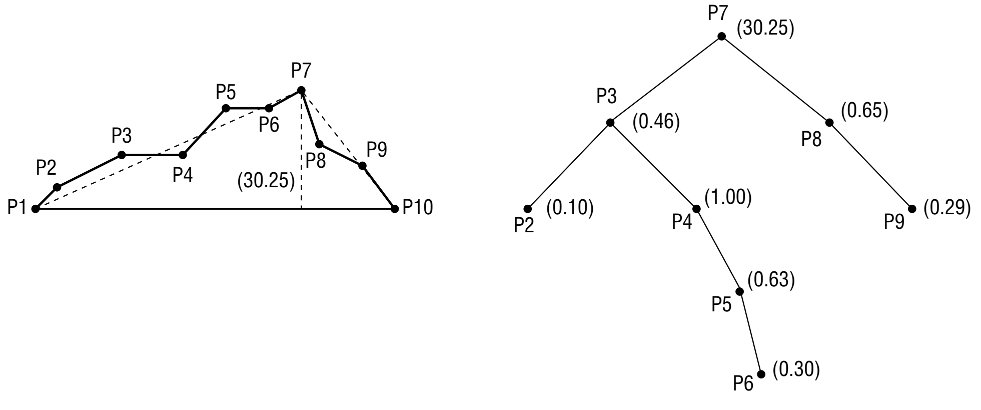


Fig 20. The binary line generalisation tree.

tree is based on the R-tree (Guttman 1984) with the difference that important objects are not stored at leaf level, but are stored at higher levels according to their importance; see Figures 21 and 22. The further one zooms in, the more tree levels must be addressed. Roughly stated, during map generation based on a selection from the reactive-tree, one should try to choose the required importance value such that a constant number of objects will be selected. This means that if the required region is large only the more important objects should be selected, and if the required region is small then the less important objects must also be selected. When using the reactive-tree and the binary line generalisation (BLG)-tree for the generalisation of an area partitioning, some problems are

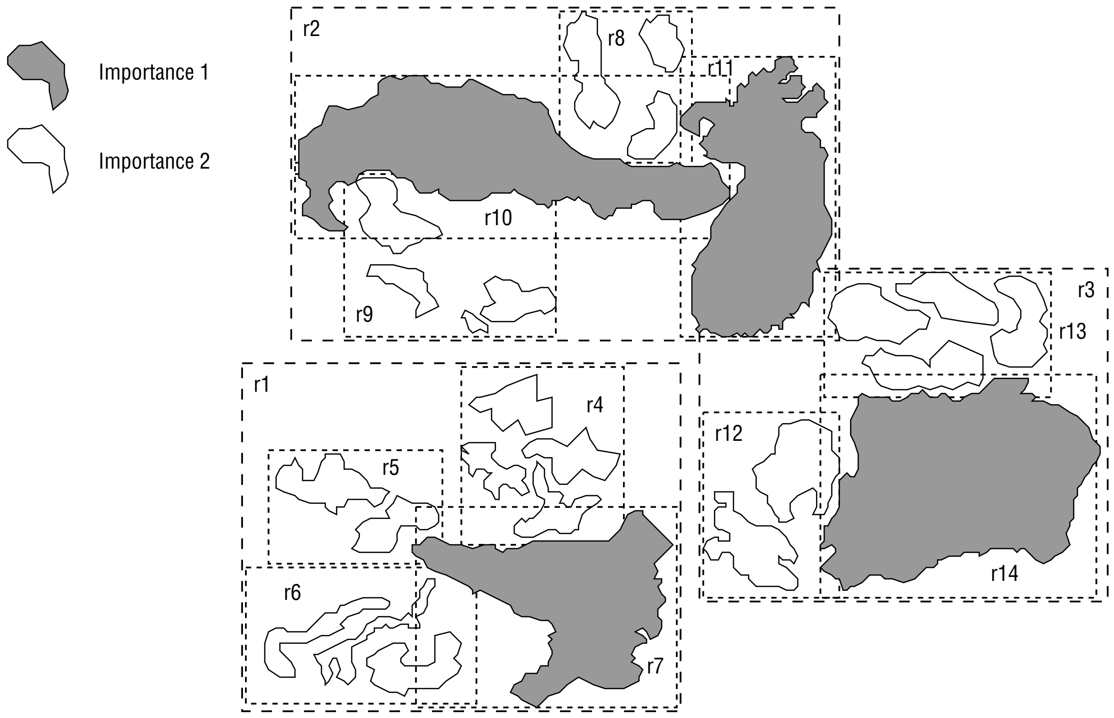


Fig 21. An example of reactive-tree rectangles.

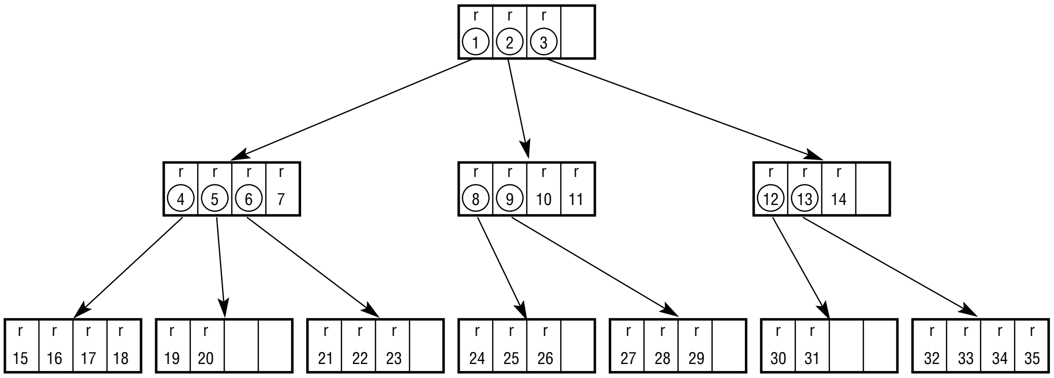
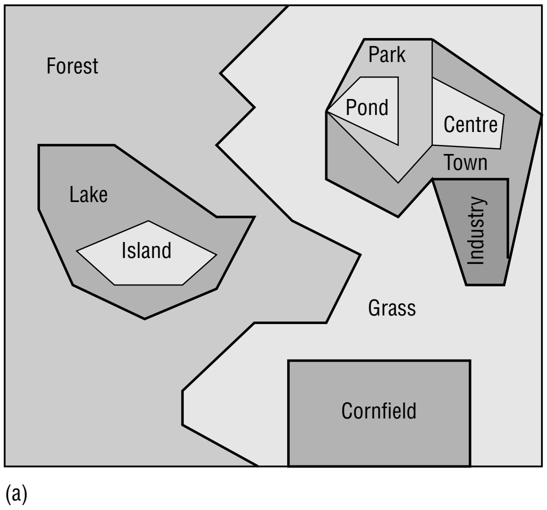


Fig 22. The reactive-tree.



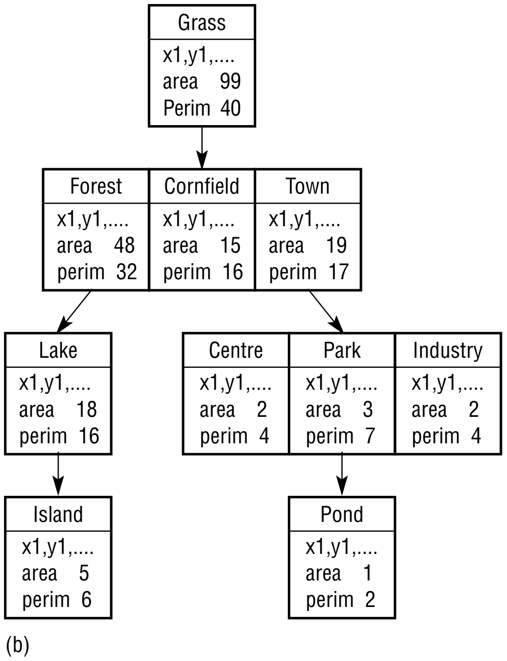


Fig 23. (a) The scene and (b) the associated GAP-tree.

encountered: gaps may be introduced by omitting small area features and mismatches may occur as a result of independent simplification of common boundaries. These problems can be solved by additionally using the generalised area partitioning (GAP)-tree. Using the reactive-tree, BLG-tree, and the GAP-tree, it is possible to browse interactively through large geographical data sets at very different scales (Oosterom and Schenkelaars 1995).

8 CONCLUSION

Though many spatial access methods have been described in the literature, the sad situation is that only a few have been implemented within the kernel of any (commercial) database and are ready to be used: exceptions are the use of the R-tree in Illustra (Informix) and the use of the Hilbert R-tree in CA-OpenIngres. At the moment several layered 'middleware' solutions are provided; for example, the spatial data engine (SDE) of ESRI (Environmental Systems Research Corporation). The drawback of the layered approaches is that the database query optimiser does not know anything about the spatial data, so it cannot generate an optimal query plan. Further, all access should be through the layer, yet already many database applications do exist which have direct access to the database. In this situation consistency may become a serious problem. Thus, in practice, possibilities are quite limited and users have to develop their own solutions. An approach for this is the Spatial Location Code (SLC: Oosterom and Vijlbrief 1996), which has been designed to enable efficient storage and retrieval of spatial data in a standard (relational) DBMS. It is used for indexing and clustering geographical objects in a database and it combines the strong aspects of several known spatial access methods (quadtree, field-tree, and Morton code) into one SLC value per object. The unique aspect of the SLC is that both the location and the extent of possibly non-zero-sized objects are approximated by this single value. The SLC is quite general and can be applied in higher dimensions.

References

Abel D J, Mark D M 1990 A comparative analysis of some 2-dimensional orderings. *International Journal of Geographical Information Systems* 4: 21–31

Bayer R, McCreight E 1973 Organization and maintenance of large ordered indexes. *Acta Informatica* 1: 173–89

Beckmann N, Kriegel H-P, Schneider R, Seeger B 1990 The R-tree: an efficient and robust access method for points and rectangles. *Proceedings ACM/SIGMOD, Atlantic City*. New York, ACM: 322-31

Bentley J L 1975 Multi-dimensional binary search trees used for associative searching. *Communications of the ACM* 18: 509–17

Bentley J L, Friedman J H 1979 Data structures for range searching. *Computing Surveys* 11: 397–409

Douglas D H, Peucker T K 1973 Algorithms for the reduction of points required to represent a digitized line or its caricature. *Canadian Cartographer* 10: 112–22

Elzinga J, Hearn D W 1972 Geometrical solutions for some minimax location problems. *Transportation Science* 6: 379–94

Faloutsos C 1988 Gray codes for partial match and range queries. *IEEE Transactions on Software Engineering* SE-14: 1381–93

Faloutsos C, Roseman S 1989 Fractals for secondary key retrieval. *Eighth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*: 247–52

Faloutsos C, Sellis T, Roussopoulos N 1987 Analysis of object oriented spatial access methods. *ACM SIGMOD* 16: 426–39

Frank A U 1983 Storage methods for space-related data: the field-tree. Tech. rep. Bericht no. 71. Zurich, Eidgenössische Technische Hochschule

Frank A U, Barrera R 1989 The Field-tree: a data structure for geographic information systems. *Symposium on the Design and Implementation of Large Spatial Databases, Santa Barbara, California*. Berlin, Springer: 29–44

Fuchs H, Abram G D, Grant E D 1983 Near real-time shaded display of rigid objects. *ACM Computer Graphics* 17: 65–72

Fuchs H, Kedem Z M, Naylor B F 1980 On visible surface generation by a priori tree structures. *ACM Computer Graphics* 14: 124–33

Goodchild M F 1989 Tiling large geographical databases. *Symposium on the Design and Implementation of Large Spatial Databases, Santa Barbara, California*. Berlin, Springer: 137–46

Goodchild M F, Grandfield A W 1983 Optimizing raster storage: an examination of four alternatives. *Proceedings AutoCarto 6*: 400–7

Guttman A 1984 R-trees: a dynamic index structure for spatial searching. *ACM SIGMOD* 13: 47–57

Hutflesz A, Six H-W, Widmayer P 1990 The R-file: an efficient access structure for proximity queries. *Proceedings IEEE Sixth International Conference on Data Engineering, Los Angeles, California*. Los Alamitos, IEEE Computer Society Press: 372–9

Jagadish H V 1990 Linear clustering of objects with multiple attributes. *ACM/SIGMOD, Atlantic City*. New York, ACM: 332–42

Kamel I, Faloutsos C 1994 Hilbert R-tree: an improved R-tree using fractals. *VLDB Conference*

Kleiner A, Brassel K E 1986 Hierarchical grid structures for static geographic databases. In Blakemore M (ed.) *AutoCarto London*. London, The Royal Institution of Chartered Surveyors: 485–96

Matsuyama T, Hao L V, Nagao M 1984 A file organization for geographic information systems based on spatial proximity. *Computer Vision, Graphics, and Image Processing* 26: 303–18

Megiddo N 1983 Linear-time algorithms for linear programming in R3 and related problems. *SIAM Journal on Computing* 12: 759–76

Nievergelt J, Hinterberger H, Sevcik K C 1984 The grid file: an adaptable, symmetric multikey file structure. *ACM Transactions on Database Systems* 9: 38–71

Nulty W G, Barholdi J J III 1994 Robust multi-dimensional searching with space-filling curves. In Waugh T C, Healey R G (eds) *Proceedings, Sixth International Symposium on Spatial Data Handling, Edinburgh, Scotland*. London, Taylor and Francis: 805–18

Oosterom P van 1989 A reactive data structure for geographic information systems. *Auto-Carto 9*, April 1989. American Society for Photogrammetry and Remote Sensing: 665–74

Oosterom P van 1990 A modified binary space partitioning tree for geographic information systems. *International Journal of Geographical Information Systems* 4: 133–46

Oosterom P van 1991 The Reactive-tree: a storage structure for a seamless, scaleless geographic database. In Mark D M, White D (eds) *AutoCarto 10*. American Congress on Surveying and Mapping: 393–407

Oosterom P van 1994 *Reactive data structures for geographic information systems*. Oxford, Oxford University Press

Oosterom P van, Claassen E 1990 Orientation insensitive indexing methods for geometric objects. *Fourth International Symposium on Spatial Data Handling, Zürich, Switzerland. Colombus,* International Geographic Union: 1016-29

Oosterom P van, Schenkelaars V 1995 The development of an interactive multi-scale GIS. *International Journal of Geographical Information Systems* 9: 489–507

Oosterom P van, Bos J van den 1989 An object-oriented approach to the design of geographic information systems. *Computers & Graphics* 13: 409–18

Oosterom P van, Vijlbrief T 1996 The spatial location code. In Kraak M-J, Molenaar M (eds) *Proceedings, Seventh International Symposium on Spatial Data Handling, Delft, The Netherlands*. London, Taylor and Francis

Orenstein J A, Manola F A 1988 PROBE spatial data modeling and query processing in an image database application. *IEEE Transactions on Software Engineering* 14: 611–29

Paterson M S, Yao F F 1989 Binary partitions with applications to hidden-surface removal and solid modeling. *Proceedings, Fifth ACM Symposium on Computational Geometry.* New York, ACM: 23–32

Robinson J T 1981 The K-D-B-tree: a search structure for large multidimensional dynamic indexes. *ACM SIGMOD* 10: 10–18

Rosenberg J B 1985 Geographical data structures compared: a study of data structures supporting region queries. *IEEE Transactions on Computer Aided Design* CAD-4: 53–67

Roussopoulos N, Leifker D 1985 Direct spatial search on pictorial databases using packed R-trees. *ACM SIGMOD* 14: 17–31

Samet H 1984 The quadtree and related hierarchical data structures. *Computing Surveys* 16: 187–260

Samet H 1989 *The design and analysis of spatial data structures*. Reading (USA), Addison-Wesley

Sylvester J J 1857 A question in the geometry of situation. *Quarterly Journal of Mathematics* 1: 79

Tamminen M 1984 Comment on quad- and octrees. *Communications of the ACM* 27: 248–9