TITLE:

Sorting in Space: Multidimensional, Spatial, and Metric Data Structures for Computer Graphics Applications

Hanan Samet
Computer Science Department
Center for Automation Research
Institute for Advanced Computer Studies
University of Maryland
College Park, Maryland 20742
e-mail: `hjs@cs.umd.edu`
url: `http://www.cs.umd.edu/~hjs`

SUMMARY STATEMENT:

We show how to represent spatial data using techniques that sort the data with respect to the space that it occupies. These techniques include quadtrees, octrees, and bounding volume hierarchies and are useful for speeding-up operations involving search in all computer graphics applications including games, ray tracing, and solid modeling.

COURSE ABSTRACT:

The representation of spatial data is an important issue in game programming, computer graphics, visualization, solid modeling, and related areas including computer vision and geographic information systems (GIS). A wide number of representations is currently in use. Recently, there has been much interest in hierarchical data structures such as quadtrees, octrees, and pyramids which are based on image hierarchies, as well methods that make use of bounding boxes which are based on object hierarchies. The key advantage of these representations is that they provide a way to index into space. In fact, they are little more than multidimensional sorts. They are compact and depending on the nature of the spatial data they save space as well as time and also facilitate operations such as search.

In this course we provide a brief overview of hierarchical spatial data structures and related algorithms that make use of them. We describe hierarchical representations of points, lines, collections of small rectangles, regions, surfaces, and volumes. For region data, we point out the dimension-reduction property of the region quadtree and octree, as how to navigate between nodes in the same tree, thereby leading to the popularity of these representations in ray tracing applications. We also demonstrate how to use these representations for both raster and vector data. In the case of nonregion data, we show how these data structures can be used to compute nearest objects in an incremental fashion so that the number of objects need not be known in advance. We also review a number of different tessellations and show why hierarchical decomposition into squares instead of triangles or hexagons is preferred. In addition a demonstration of the SAND spatial browser based on the SAND spatial database system and of the VASCO JAVA applet illustrating these methods (found at `http://www.cs.umd.edu/~hjs/quadtree/index.html` is presented.

PREREQUISITE:

A familiarity with computer terminology and some programming experience.

COURSE LEVEL:

Beginner

INTENDED AUDIENCE:

Practitioners working in computer graphics will be given a different perspective on data structures found to be useful in most applications Game developers and technical managers will appreciate the presentation and methods described herein.

COURSE SYLLABUS:

1. Introduction
    a. Sample queries
    b. Spatial Indexing
    c. Sorting approach
    d. Minimum bounding rectangles (e.g., R-tree)
    e. Disjoint cells (e.g., R+-tree, k-d-B-tree)
    f. Uniform grid
    g. Location-based queries vs: feature-based queries
    h. Region quadtree
    i. Pyramid
    j. Region quadtrees vs: pyramids
    k. Space ordering methods
2. Points
    a. point quadtree
    b. MX quadtree
    c. PR quadtree
    d. k-d tree
3. Lines
    a. Strip tree
    b. PM1 quadtree
    c. PM2 quadtree
    d. PM3 quadtree
    e. PMR quadtree

4. Rectangles
    a. MX-CIF quadtree
    b. Loose quadtree
    c. R-tree
5. Regions
    a. Region quadtree
    b. Dimension reduction
    c. Tessellations
    d. Bintree
    e. BSP tree
6. Surfaces and Volumes
    a. Restricted quadtree
    b. Region octree
    c. PM octree
7. Metric Data
    a. vp-tree
    b. gh-tree
    c. mb-tree
8. Operations
    a. Incremental nearest object location
    b. Boolean set operations
    c. Nearest neighbor operations
9. Example system
    a. SAND internet browser
    b. JAVA spatial data applets

COURSE MATERIALS

Participants receive a copy of the slides. In addition, there is a web site at `http://www.cs.umd.edu/~hjs/quadtree/index.html` where applets demonstrating much of the material in the course are available. Participants are referred to the text: H.Samet, *Foundations of Multidimensional and Metric Data Structures*, Morgan-Kaufmann, San Francisco, 2006. Participants also have the opportunity to obtain the book at a discount of 20% as reflected at `http://www.cs.umd.edu/~hjs/multidimensional-book-flyer.pdf`

SPEAKER BIOGRAPHY

Hanan Samet received the B.S. degree in engineering from the University of California, Los Angeles, and the M.S. Degree in operations research and the M.S. and Ph.D. degrees in computer science from Stanford University, Stanford, CA. He is a Fellow of the IEEE, ACM, and IAPR (International Association for Pattern Recognition).

In 1975 he joined the Computer Science Department at the University of Maryland, College Park, where he is now a Professor. He is a member of the Computer Vision Laboratory of the Center for Automation Research and also has an appointment in the University of Maryland Institute for Advanced Computer Studies. At the Computer Vision Laboratory he leads a number of research projects on the use of hierarchical data structures for geographic information systems. His research group has developed the QUILT system which is a GIS based on hierarchical spatial data structures such as quadtrees and octrees, the SAND system which integrates spatial and non-spatial data, the SAND Spatial Browser which enables browsing through a spatial database using a graphical user interface, the VASCO spatial indexing applet (found at `http://www.cs.umd.edu/~hjs/quadtree/index.html`), and a symbolic image database system.

His research interests are data structures, computer graphics, geographic information systems, computer vision, robotics, and database management systems. He is the author of the recent book titled *Foundations of Multidimensional and Metric Data Structures* (`http://www.mkp.com/multidimensional`) published by Morgan-Kaufmann, an imprint of Elsevier, in 2006, and of the first two books on spatial data structures titled *Design and Analysis of Spatial Data Structures*, and *Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS*, both published by Addison-Wesley in 1990. He is an Area Editor of *Graphical Models and Image Processing* and on the Editorial Board of *Image Understanding*, *Journal of Visual Languages*, *GeoInformatica*, and *Journal of Spatial Cognition and Computation*.

Sorting in Space

Multidimensional, Spatial, and Metric Data Structures for Computer Graphics Applications

Hanan Samet
hjs@cs.umd.edu

http://www.cs.umd.edu/~hjs

Department of Computer Science

University of Maryland

College Park, MD 20742, USA

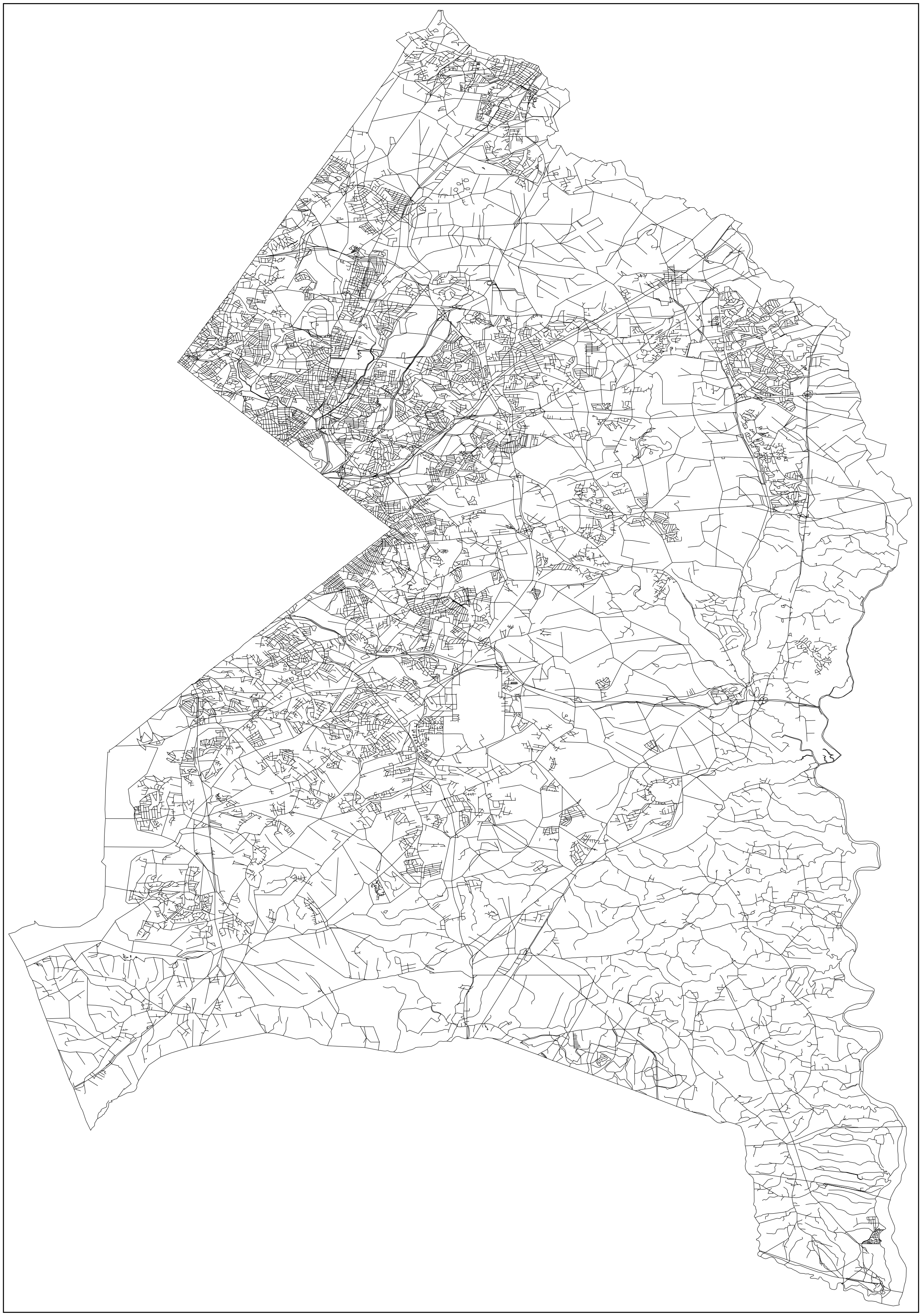
Unless explicitly stated otherwise, the upper-left corner of each slide indicates the page numbers in Foundations of Multidimensional and Metric Data Structures by H. Samet, Morgan-Kaufmann, San Francisco, 2006, where more details on the topic can be found

Outline

1. Introduction
2. Points
3. Lines
4. Rectangles
5. Regions
6. Surfaces and Volumes
7. Metric Data
8. Operations
9. Example system

hi28

PRINCE GEORGES COUNTY

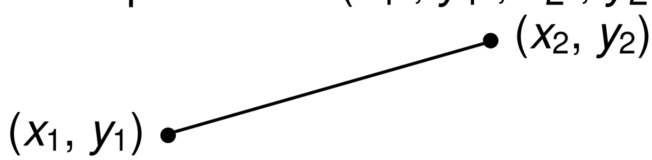


EXAMPLE QUERIES ON LINE SEGMENT DATABASES

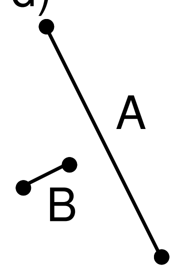
- Queries about line segments
  1. All segments that intersect a given point or set of points
  2. All segments that have a given set of endpoints
  3. All segments that intersect a given line segment
  4. All segments that are coincident with a given line segment
- Proximity queries
  1. The nearest line segment to a given point
  2. All segments within a given distance from a given point (also known as a range or window query)
- Queries involving attributes of line segments
  1. Given a point, find the closest line segment of a particular type
  2. Given a point, find the minimum enclosing polygon whose constituent line segments are all of a given type
  3. Given a point, find all the polygons that are incident on it

WHAT MAKES CONTINUOUS SPATIAL DATA DIFFERENT

1. Spatial extent of the objects is the key to the difference
2. A record in a DBMS may be considered as a point in a multidimensional space
   - a line can be transformed (i.e., represented) as a point in 4-d space with (x_1, y_1, x_2, y_2)



   - good for queries about the line segments
   - not good for proximity queries since points outside the object are not mapped into the higher dimensional space
   - representative points of two objects that are physically close to each other in the original space (e.g., 2-d for lines) may be very far from each other in the higher dimensional space (e.g., 4-d)
   - Ex:



   - problem is that the transformation only transforms the space occupied by the objects and not the rest of the space (e.g., the query point)
   - can overcome by projecting back to original space
3. Use an index that sorts based upon spatial occupancy (i.e., extent of the objects)

SPATIAL INDEXING REQUIREMENTS

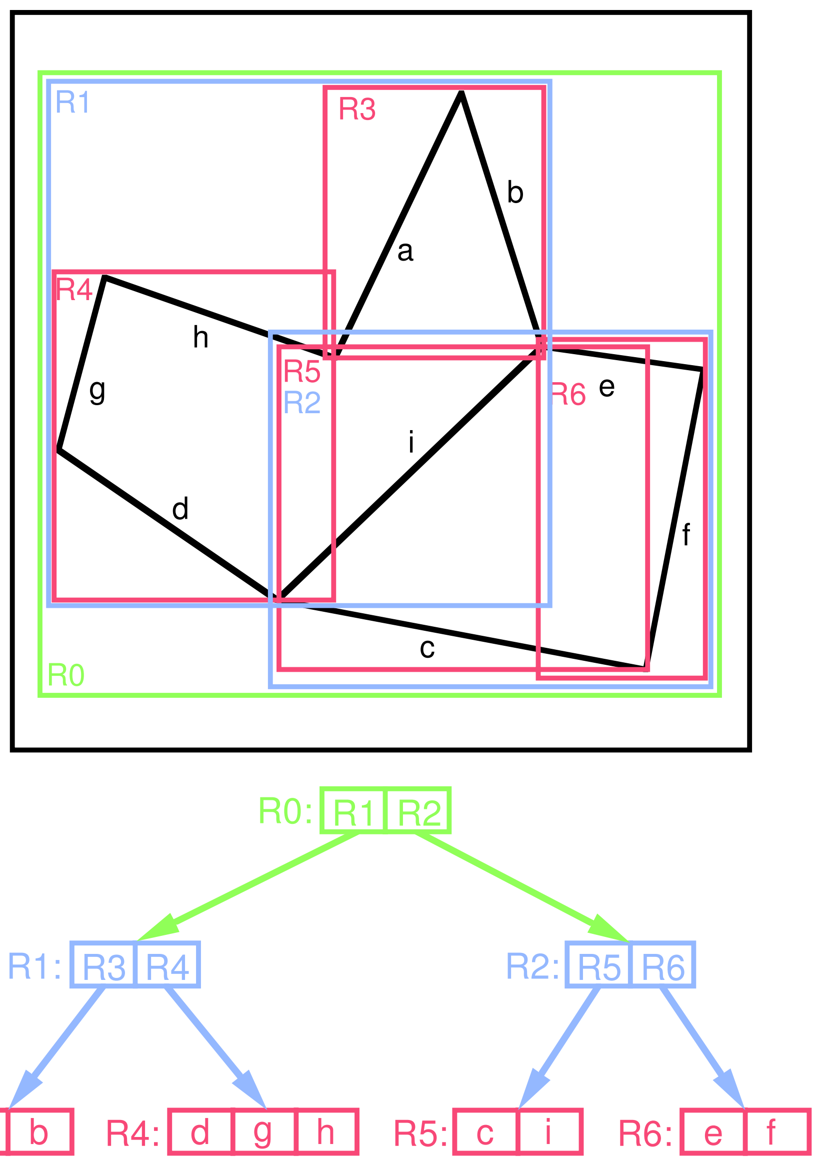
1. Compatibility with the data being stored
2. Choose an appropriate zero or reference point
3. Need an implicit rather than an explicit index
   - impossible to foresee all possible queries in advance
   - cannot have an attribute for every possible spatial relationship
     a. derive adjacency relations
     b. 2-d strings capture a subset of adjacencies
        - all rows
        - all columns
   - implicit index is better as an explicit index which, for example, sorts two-dimensional data on the basis of distance from a given point is impractical as it is inapplicable to other points
   - implicit means that don't have to resort the data for queries other than updates

SORTING ON THE BASIS OF SPATIAL OCCUPANCY

- Decompose the space from which the data is drawn into regions called *buckets* (like hashing but preserves order)
- Interested in methods that are designed specifically for the spatial data type being stored
- Basic approaches to decomposing space
  1. minimum bounding rectangles
     - e.g., R-tree
     - good at distinguishing empty and non-empty space
     - drawbacks:
       a. non-disjoint decomposition of space
          - may need to search entire space
       b. inability to correlate occupied and unoccupied space in two maps
  2. disjoint cells
     - drawback: objects may be reported more than once
     - uniform grid
       a. all cells the same size
       b. drawback: possibility of many sparse cells
     - adaptive grid — quadtree variants
       a. regular decomposition
       b. all cells of width power of 2
     - partitions at arbitrary positions
       a. drawback: not a regular decomposition
       b. e.g., R^{+}-tree
- Can use as approximations in filter/refine query processing strategy

MINIMUM BOUNDING RECTANGLES

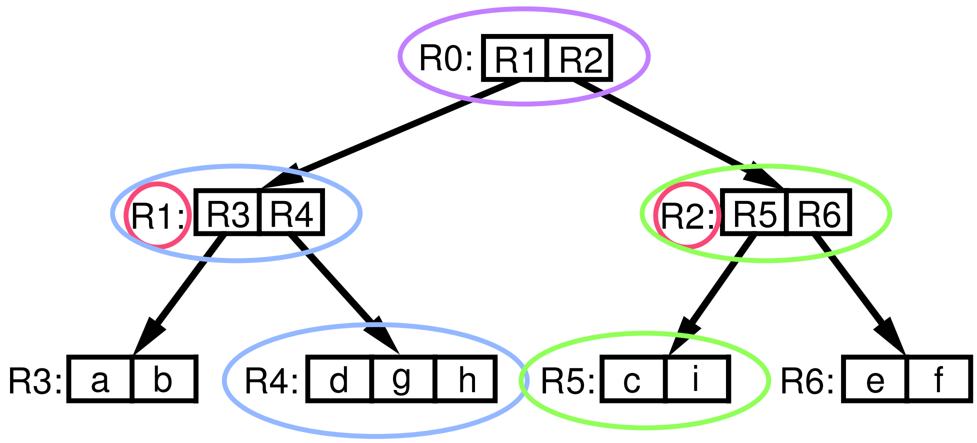
- Objects grouped into hierarchies, stored in a structure similar to a B-tree
- Drawback: not a disjoint decomposition of space
- Object has single bounding rectangle, yet area that it spans may be included in several bounding rectangles
- Examples include the R-tree and the R*-tree
- Order (m,M) R-tree
  1. between $m \leqslant \lceil M/2 \rceil$ and M entries in each node except root
  2. at least 2 entries in root unless a leaf node

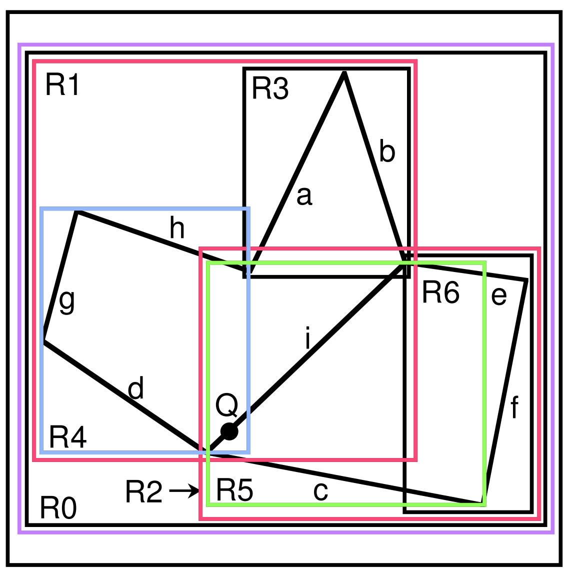


SEARCHING FOR A POINT OR LINE SEGMENT IN AN R-TREE

- Drawback is that may have to examine many nodes since a line segment can be contained in the covering rectangles of many nodes yet its record is contained in only one leaf node (e.g., i in R2, R3, R4, and R5)

Ex: Search for a line segment containing point Q

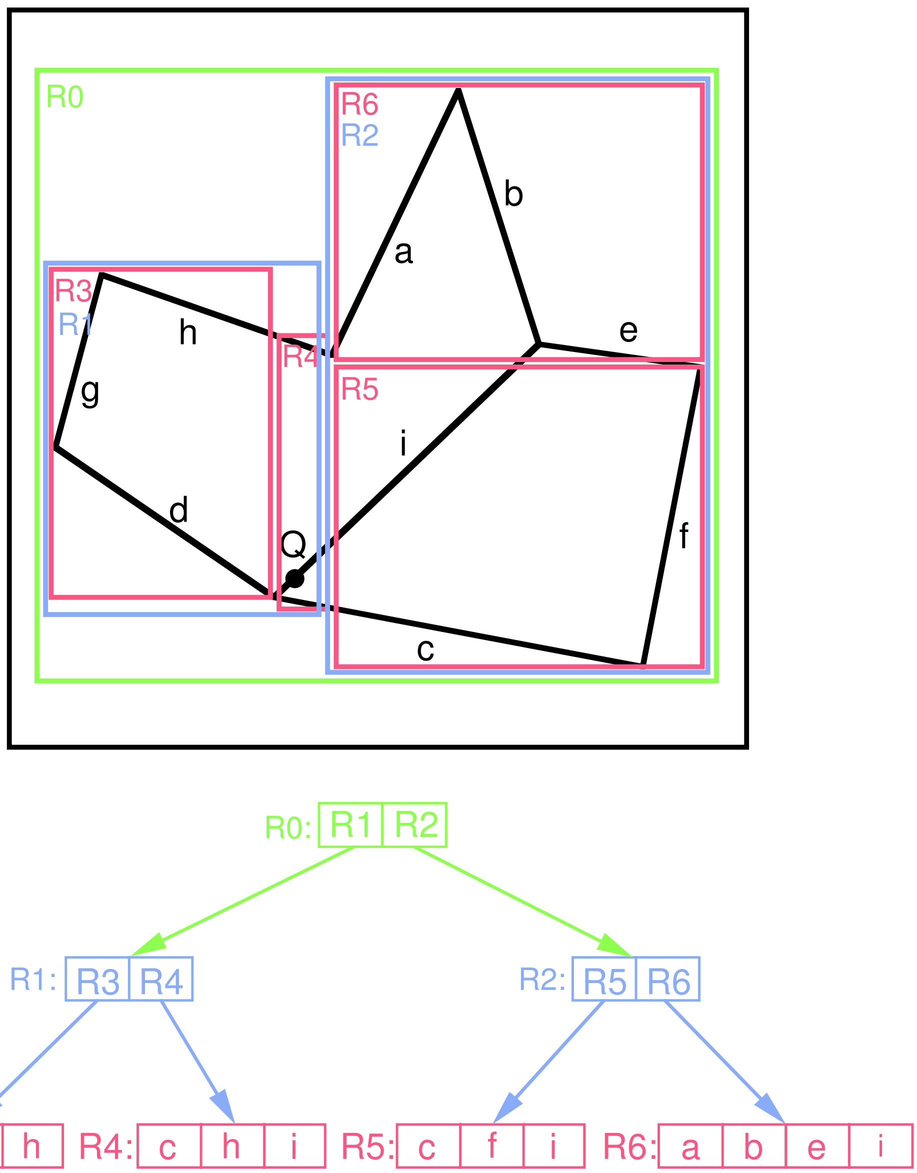




- Q is in R0
- Q can be in both R1 and R2
- Searching R1 first means that R4 is searched but this leads to failure even though Q is part of i which is in R4
- Searching R2 finds that Q can only be in R5

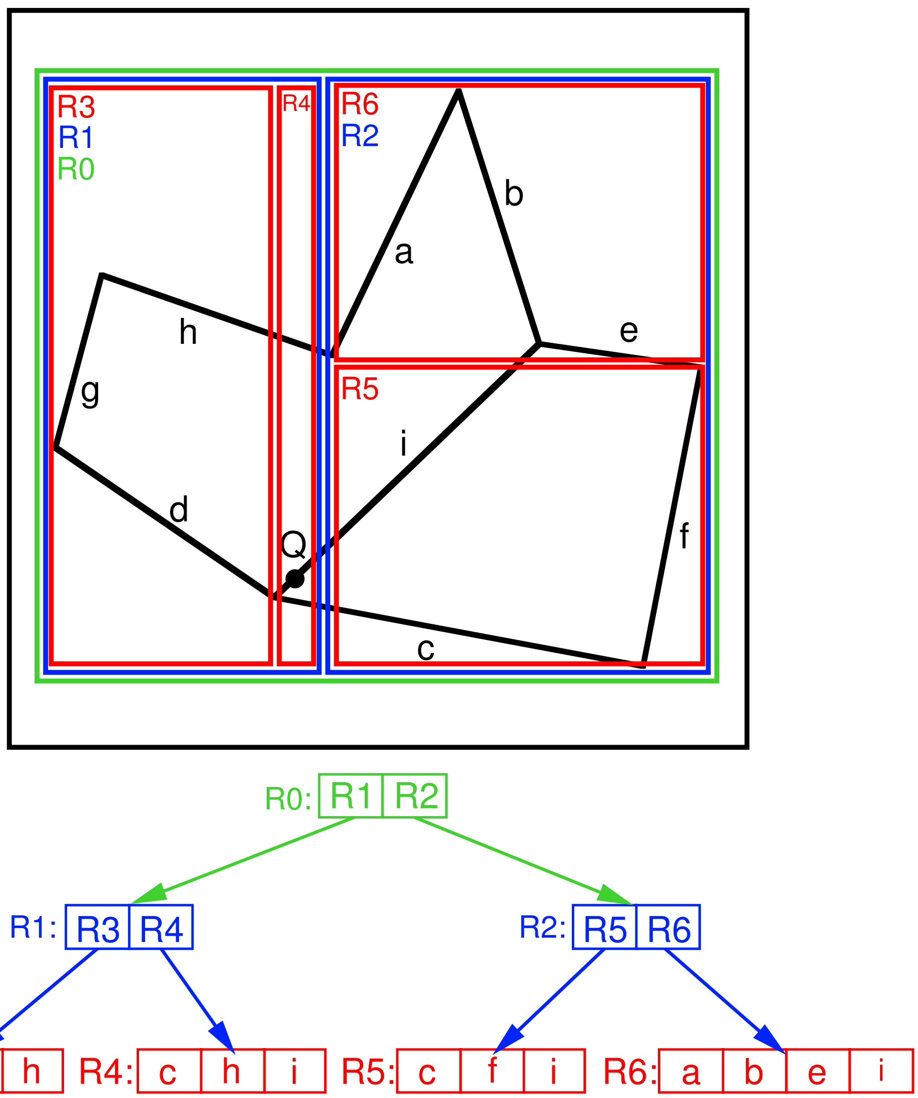
DISJOINT CELLS

- Objects decomposed into disjoint subobjects; each subobject in different cell
- Techniques differ in degree of regularity
- Drawback: in order to determine area covered by object, must retrieve all cells that it occupies
- R+-tree (also k-d-B-tree) and cell tree are examples of this technique



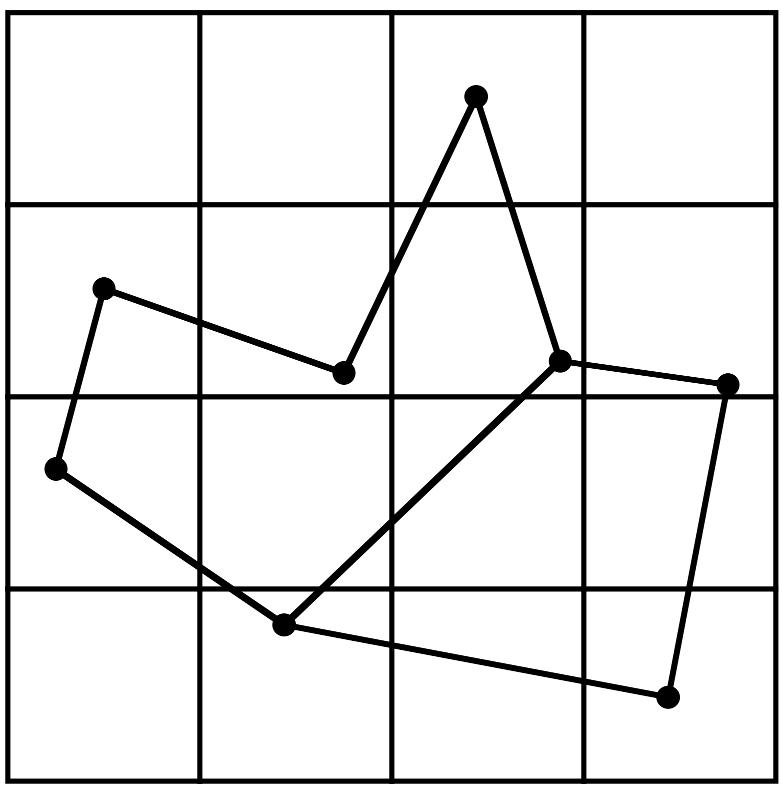
K-D-B-TREES

- Rectangular embedding space is hierarchically decomposed into disjoint rectangular regions
- No dead space in the sense that at any level of the tree, entire embedding space is covered by one of the nodes
- Blocks of k-d tree partition of space are aggregated into nodes of a finite capacity
- When a node overflows, it is split along one of the axes
- Originally developed to store points but may be extended to non-point objects represented by their minimum bounding boxes
- Drawback: in order to determine area covered by object, must retrieve all cells that it occupies



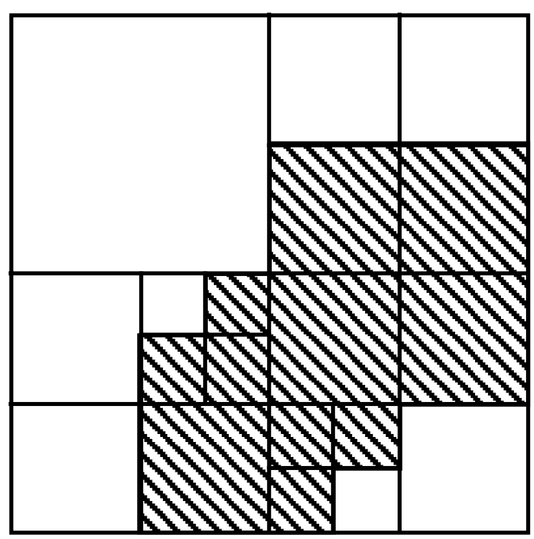
UNIFORM GRID

- Ideal for uniformly distributed data
- Supports set-theoretic operations
- Spatial data (e.g., line segment data) is rarely uniformly distributed



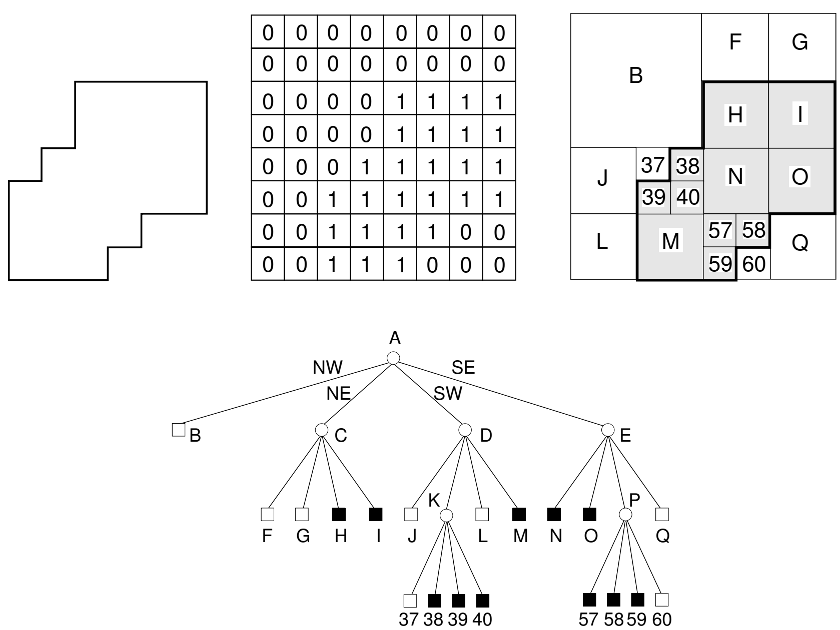
QUADTREES

- Hierarchical variable resolution data structure based on regular decomposition
- Many different decomposition schemes and applicable to different data types:
  1. points
  2. lines
  3. regions
  4. rectangles
  5. surfaces
  6. volumes
  7. higher dimensions including time
     - changes meaning of nearest
       a. nearest in time, OR
       b. nearest in distance
- Can handle both raster and vector data as just a spatial index
- Shape is usually independent of order of inserting data
- Ex: region quadtree
- A decomposition into blocks — not necessarily a tree!



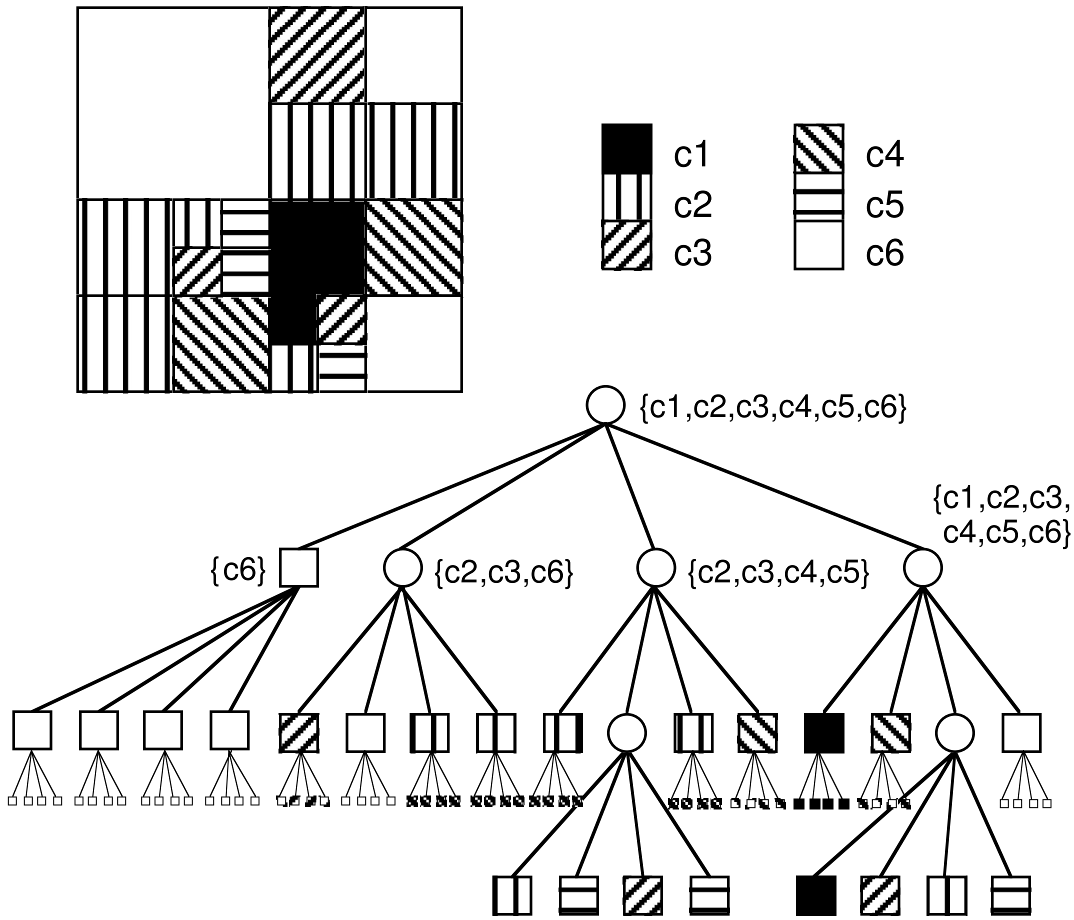
REGION QUADTREE

- Repeatedly subdivide until obtain homogeneous region
- For a binary image (BLACK ≡ 1 and WHITE ≡ 0)
- Can also use for multicolored data (e.g., a landuse class map associating colors with crops)
- Can also define data structure for grayscale images
- A collection of maximal blocks of size power of two and placed at predetermined positions
  1. could implement as a list of blocks each of which has a unique pair of numbers:
     - concatenate sequence of 2 bit codes corresponding to the path from the root to the block's node
     - the level of the block's node
  2. does not have to be implemented as a tree
     - tree good for logarithmic access
- A variable resolution data structure in contrast to a pyramid (i.e., a complete quadtree) which is a multiresolution data structure



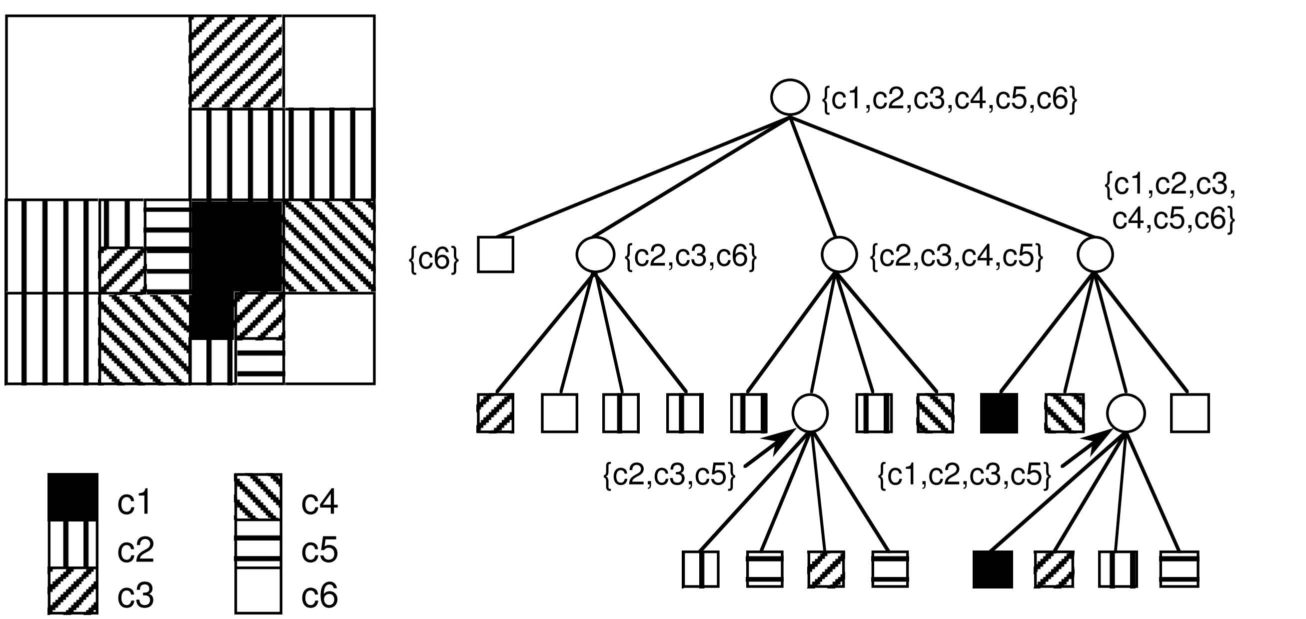
PYRAMID

- Internal nodes contain summary of information in nodes below them
- Useful for avoiding inspecting nodes where there could be no relevant information



QUADTREES VS. PYRAMIDS

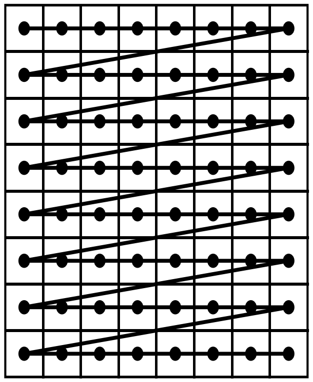
- Quadtrees are good for location-based queries
  1. e.g., what is at location *x*?
  2. not good if looking for a particular feature as have to examine every block or location asking “are you the one I am looking for?”
- Pyramid is good for feature-based queries — e.g.,
  1. does wheat exist in region *x*?
     - if wheat does not appear at the root node, then impossible to find it in the rest of the structure and the search can cease
  2. report all crops in region *x* — just look at the root
  3. select all locations where wheat is grown
     - only descend node if there is possibility that wheat is in one of its four sons — implies little wasted work
- Ex: truncated pyramid where 4 identically-colored sons are merged

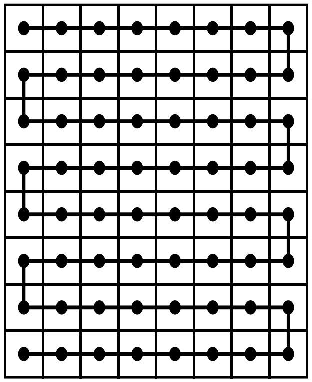


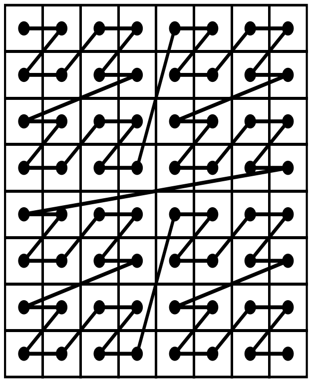
- Can represent as a list of leaf and nonleaf blocks (e.g., as a linear quadtree)

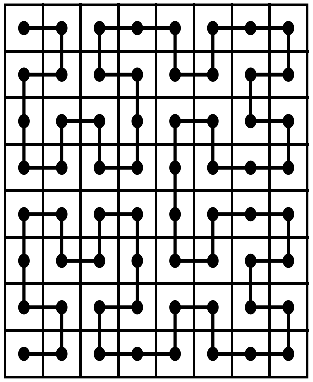
Ordering Space

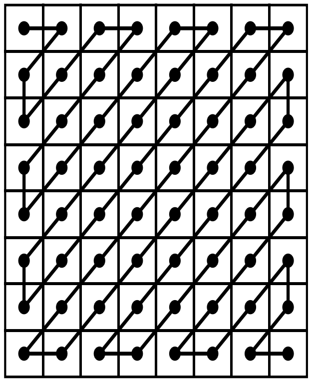
- Many ways of laying out the addresses corresponding to the locations in space of the cells each having its own mapping function
- Can use one of many possible space-filling curves
- Important to distinguish between *address* and *location* or *cell*
- *Address of a location or cell* $\equiv$ physical location (e.g., in memory, on disk, etc.), if any, where some of the information associated with the *location* or *cell* is stored

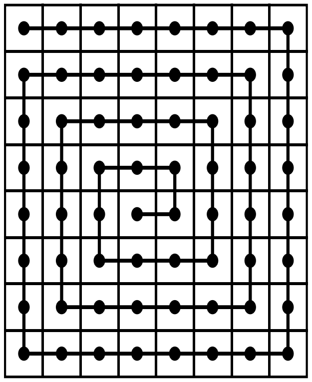

row order

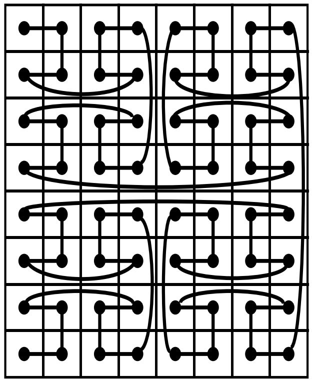

row-prime order

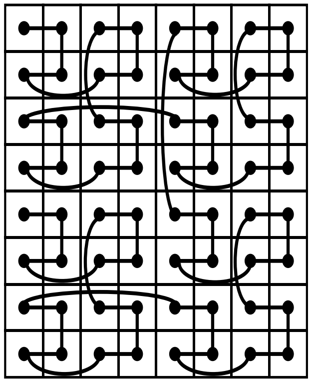

morton order

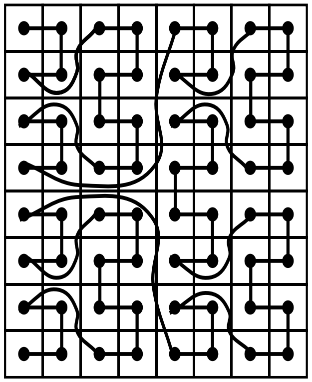

peano-hilbert order


cantor-diagonal order


spiral order

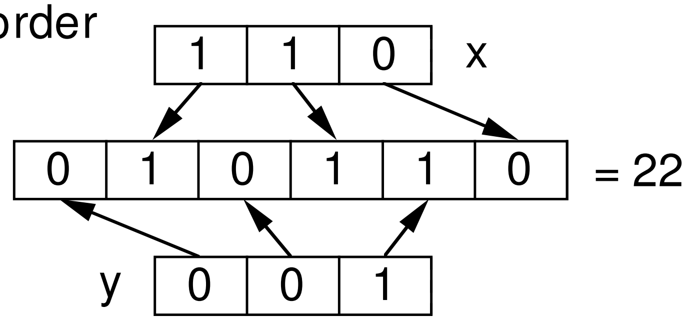

gray code


double gray order


u order

CONVERTING BETWEEN POINTS AND CURVES

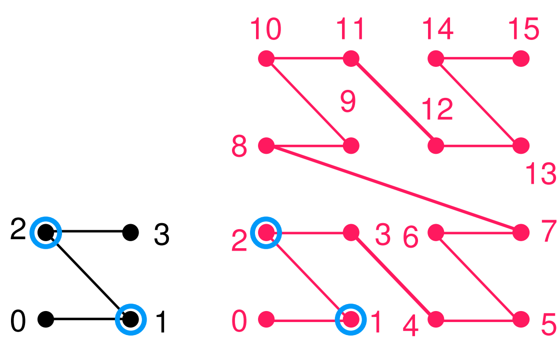
- Need to know size of image for all but the Morton order
- Relatively easy for all but the Peano-Hilbert order which is difficult (although possible) to decode and encode to obtain the corresponding *x* and *y* coordinate values
- Morton order
  1. use bit interleaving of binary representation of the *x* and *y* coordinates of the point
  2. also known as Z-order
  3. Ex: `Atlanta` (6,1)



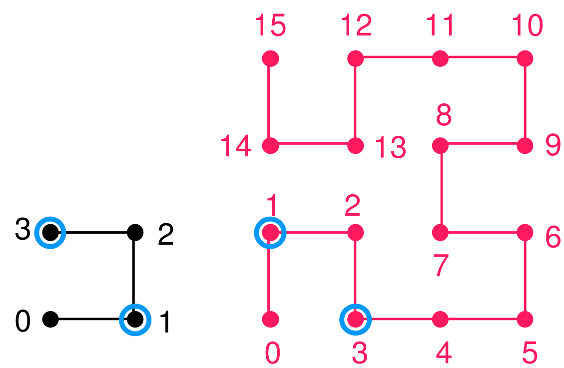
STABILITY OF SPACE ORDERING METHODS

- An order is *stable* if the relative order of the individual pixels is maintained when the resolution (i.e., the size of the space in which the cells are embedded) is doubled or halved
- Morton order is stable while the Peano-Hilbert order is not
- Ex:

Morton:



Peano-Hilbert:



- Result of doubling the resolution (i.e., the coverage) in which case the circled points do not maintain the same relative order in the Peano-Hilbert order while they do in the Morton order

DESIRABLE PROPERTIES OF SPACE FILLING CURVES

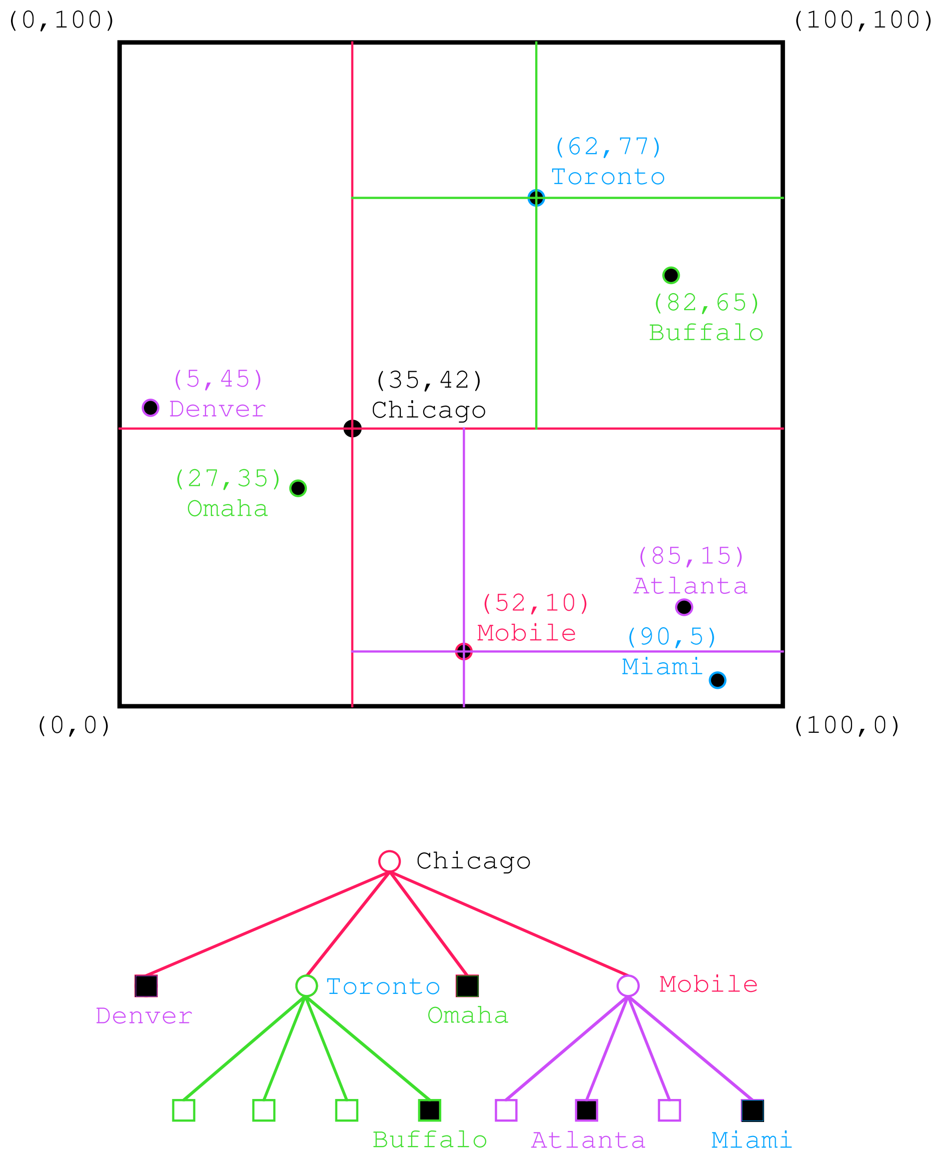
1. Pass through each point in the space once and only once
2. Two points that are neighbors in space are neighbors along the curve and vice versa
   - impossible to satisfy for all points at all resolutions
3. Easy to retrieve neighbors of a point
4. Curve should be stable as the space grows and contracts by powers of two with the same origin
   - yes for Morton and Cantor orders
   - no for row, row-prime, Peano-Hilbert, and spiral orders
5. Curve should be admissible
   - at each step at least one horizontal and one vertical neighbor must have already been encountered
   - used by active border algorithms - e.g., connected component labeling algorithm
   - row and Morton orders are admissible
   - Peano-Hilbert order is not admissible
   - row-prime, Cantor, and spiral orders are admissible if permit the direction of the horizontal and vertical neighbors to vary from point to point
6. Easy to convert between two-dimensional data and the curve and vice-versa
   - easy for Morton order
   - difficult for Peano-Hilbert order
   - relatively easy for row, row-prime, Cantor, and spiral orders

Outline

1. Introduction
2. Points
3. Lines
4. Rectangles
5. Regions
6. Surfaces and Volumes
7. Metric Data
8. Operations
9. Example system

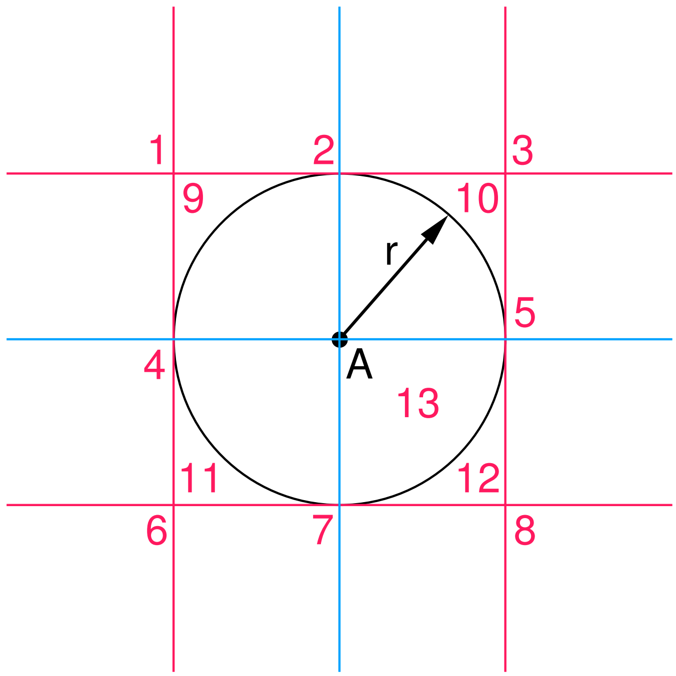
POINT QUADTREE (Finkel/Bentley)

- Marriage between a uniform grid and a binary search tree



REGION SEARCH

- Ex: Find all points within radius *r* of point A

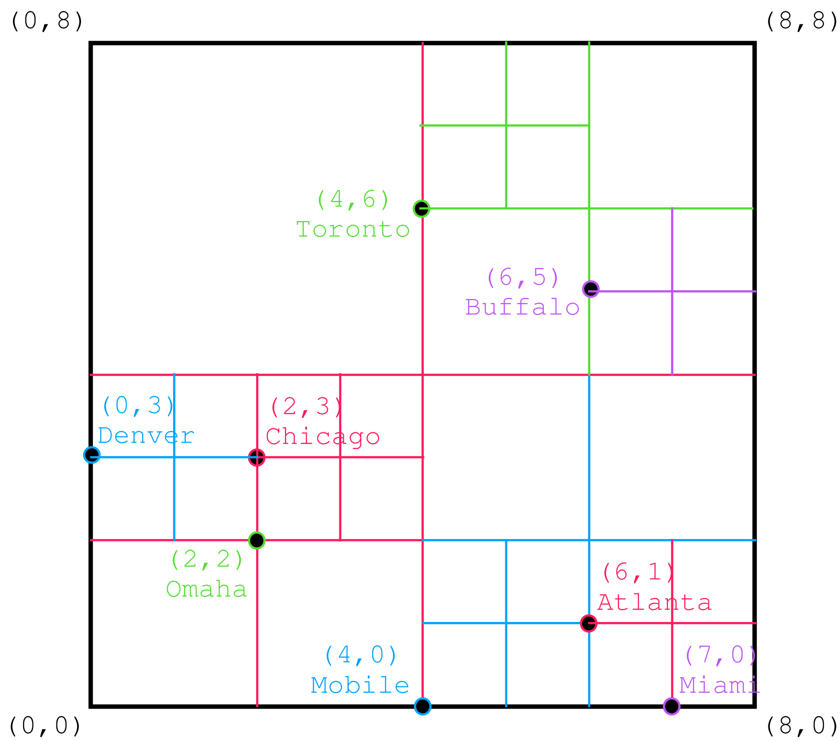


- Use of quadtree results in pruning the search space
- If a quadrant subdivision point *p* lies in a region *l*, then search the quadrants of *p* specified by *l*

1. SE
2. SE, SW
3. SW
4. SE, NE
5. SW, NW
6. NE
7. NE, NW
8. NW
9. All but NW
10. All but NE
11. All but SW
12. All but SE
13. All

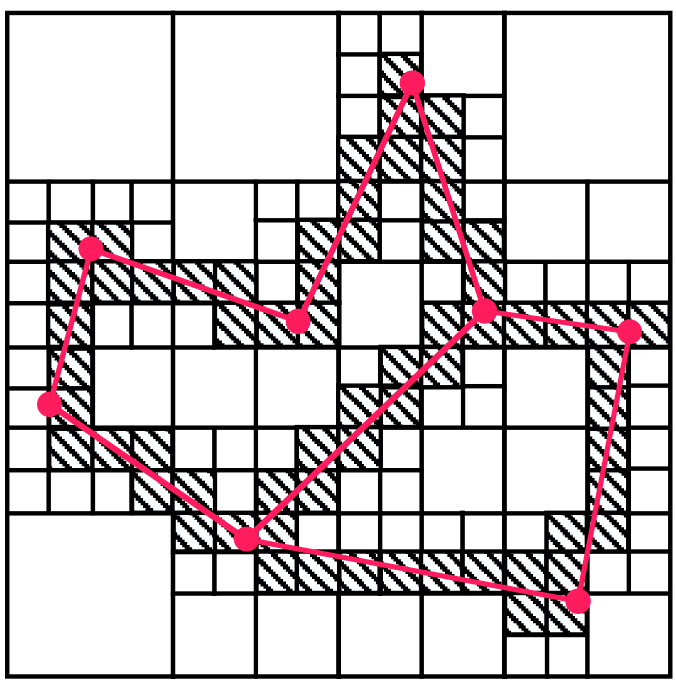
MX QUADTREE (Hunter)

- Points are like BLACK pixels in a region quadtree
- Useful for raster to vector conversion
- Empty cells are merged to form larger empty cells
- Only good for discrete data
- Good for sparse matrix applications
- Assume that the point is associated with the lower left corner of each cell
- Ex: assume an 8 x 8 array
  divide coordinate values by 12.5



APPLICATION OF THE MX QUADTREE (Hunter)

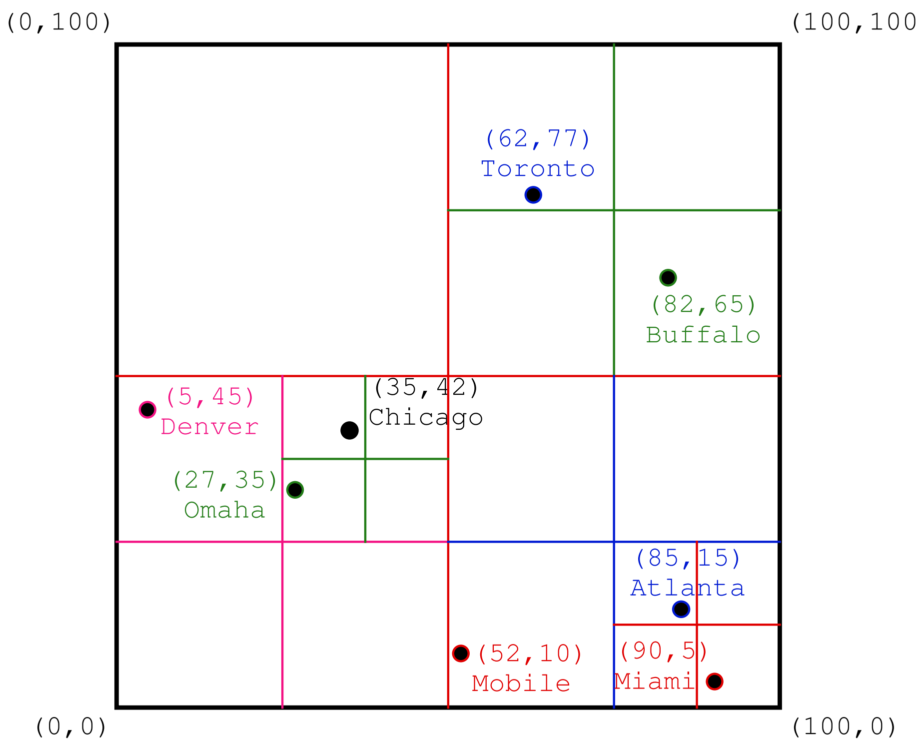
- Represent the boundary as a sequence of BLACK pixels in a region quadtree
- Useful for a simple digitized polygon (i.e., non-intersecting edges)
- Three types of nodes
  1. interior - treat like WHITE nodes
  2. exterior - treat like WHITE nodes
  3. boundary - the edge of the polygon passes through them and treated like BLACK nodes
- Disadvantages
  1. a thickness is associated with the line segments
  2. no more than 4 lines can meet at a point



PR QUADTREE (Orenstein)

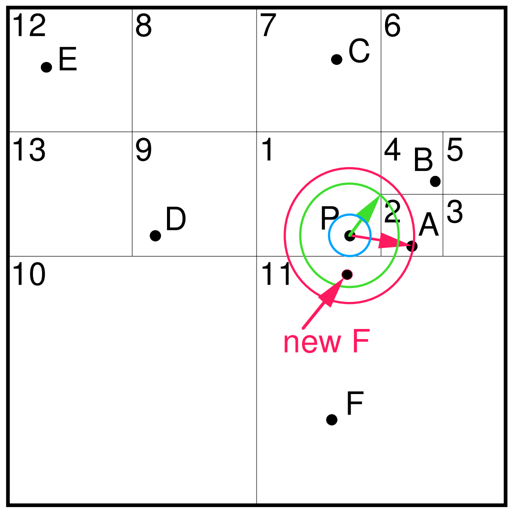
1. Regular decomposition point representation
2. Decomposition occurs whenever a block contains more than one point
3. Useful when the domain of data points is not discrete but finite
4. Maximum level of decomposition depends on the minimum separation between two points
   - if two points are very close, then decomposition can be very deep
   - can be overcome by viewing blocks as buckets with capacity *c* and only decomposing the block when it contains more than *c* points

Ex: *c* = 1



FINDING THE NEAREST OBJECT

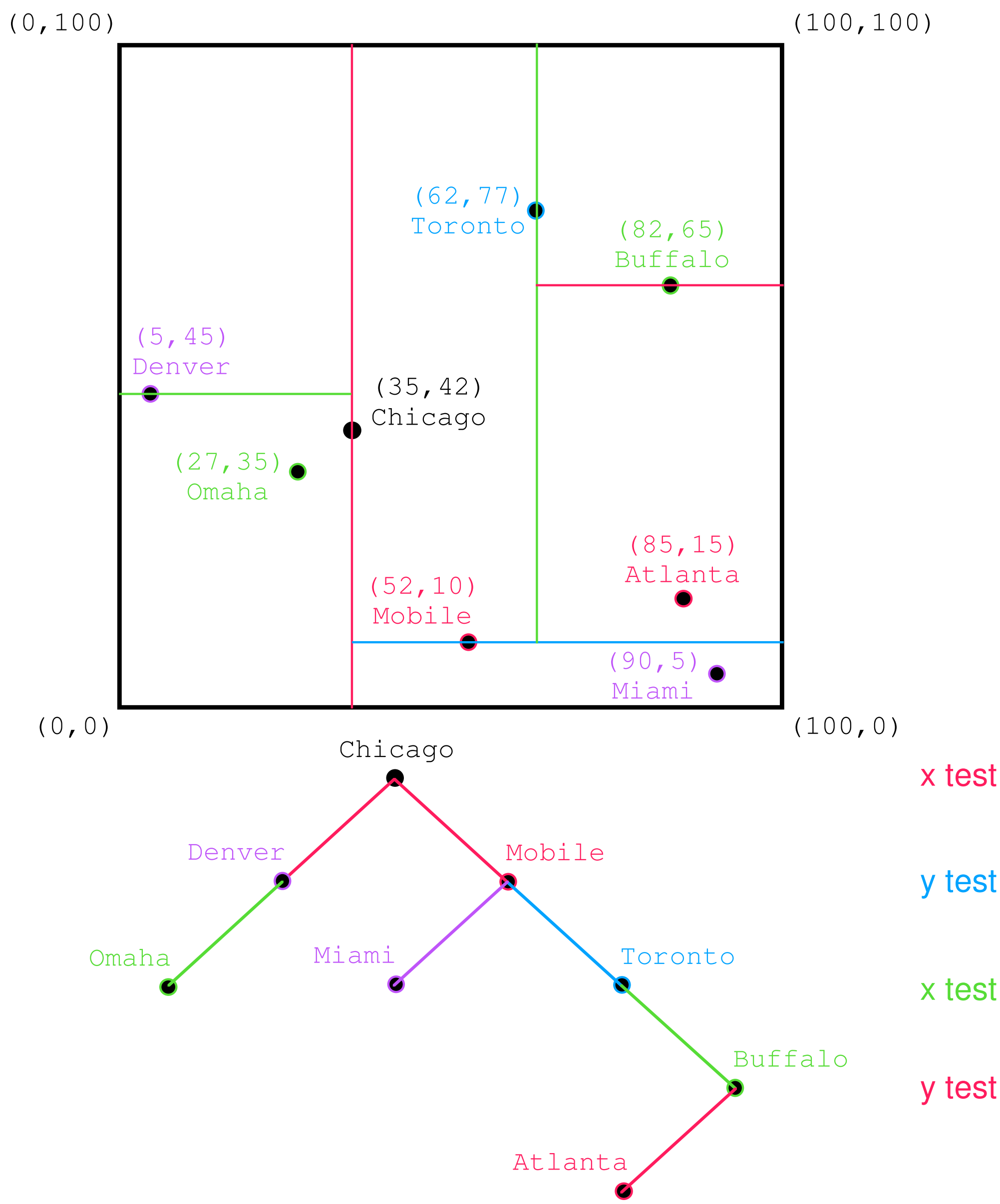
- Ex: find the nearest object to P



- Assume PR quadtree for points (i.e., at most one point per block)
- Search neighbors of block 1 in counterclockwise order
- Points are sorted with respect to the space they occupy which enables pruning the search space
- Algorithm:
  1. start at block 2 and compute distance to P from A
  2. ignore block 3 whether or not it is empty as A is closer to P than any point in 3
  3. examine block 4 as distance to SW corner is shorter than the distance from P to A; however, reject B as it is further from P than A
  4. ignore blocks 6, 7, 8, 9, and 10 as the minimum distance to them from P is greater than the distance from P to A
  5. examine block 11 as the distance from P to the southern border of 1 is shorter than the distance from P to A; however, reject F as it is further from P than A
- If F was moved, a better order would have started with block 11, the southern neighbor of 1, as it is closest

K-D TREE (Bentley)

- Test one attribute at a time instead of all simultaneously as in the point quadtree
- Usually cycle through all the attributes
- Shape of the tree depends on the order in which the data is encountered

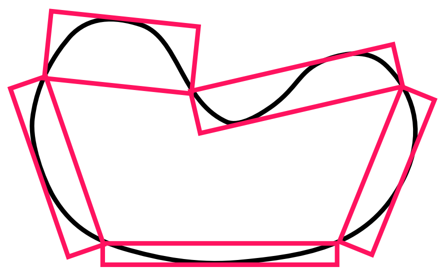


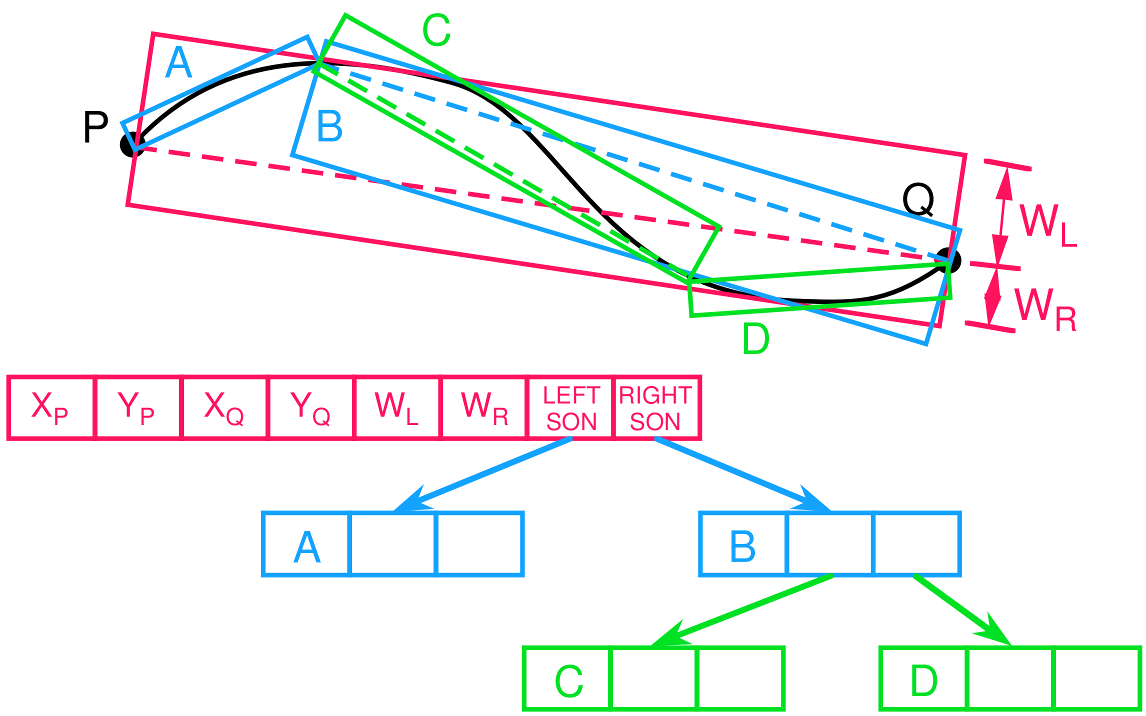
Outline

1. Introduction
2. Points
3. Lines
4. Rectangles
5. Regions
6. Surfaces and Volumes
7. Metric Data
8. Operations
9. Example system

STRIP TREE (Ballard, Peucker)

- Top-down hierarchical curve approximation
- Rectangle strips of arbitrary orientation
- Assume curve is continuous
- Ex:

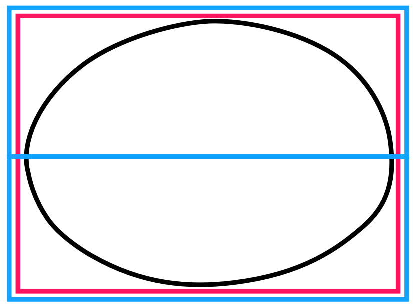




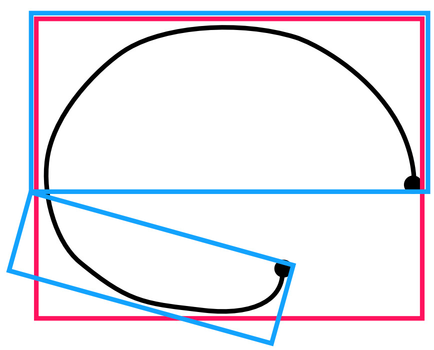
- *Contact points* = where the curve touches the box
  1. not tangent points
  2. curve need not be differentiable - just continuous
- Terminate when all rectangles are of width $\leqslant$ W

SPECIAL CASES

1. Closed curve



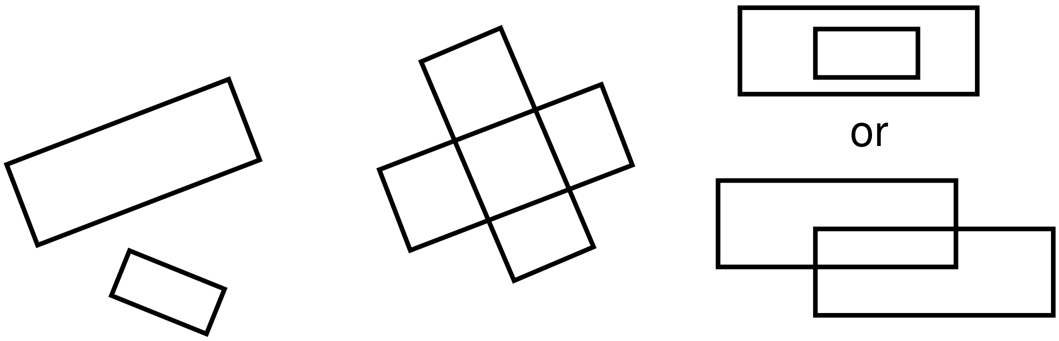
2. Curve extends beyond its endpoints



- enclosed by a rectangle
- split into two rectangular strips

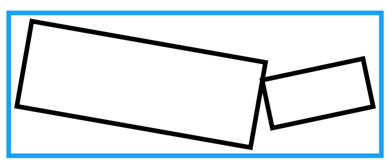
APPLICATIONS

1. Curve intersection



NULL CLEAR POSSIBLE

2. Union of two curves

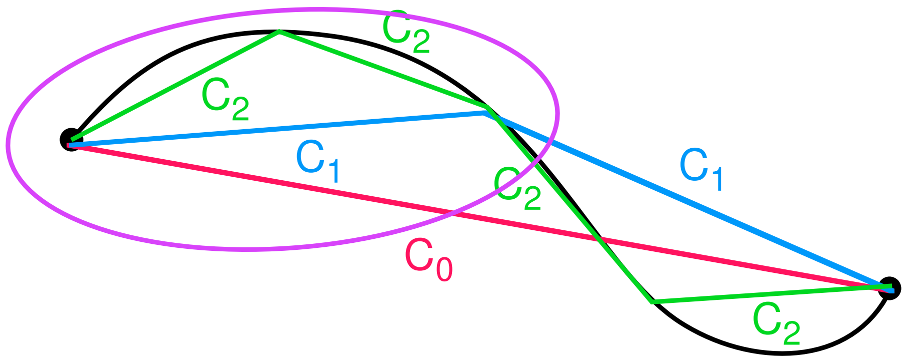


- not possible as the result may fail to be continuous

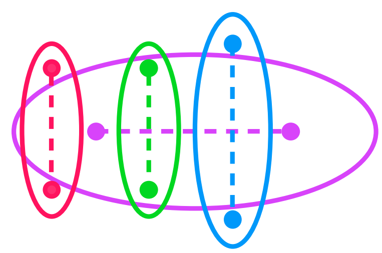
3. Others
   - length
   - area of a closed curve
   - intersection of curves with areas
   - etc.

ARC TREE (Günther)

- Recursive decomposition based on arc length
- Complete binary tree
- n^{th} level approximation has 2^n elements
- Decomposition process stops when straight line approximation of each subarc is within a given tolerance
- Drawback: computing arc length requires evaluating an elliptical integral
- Ex:



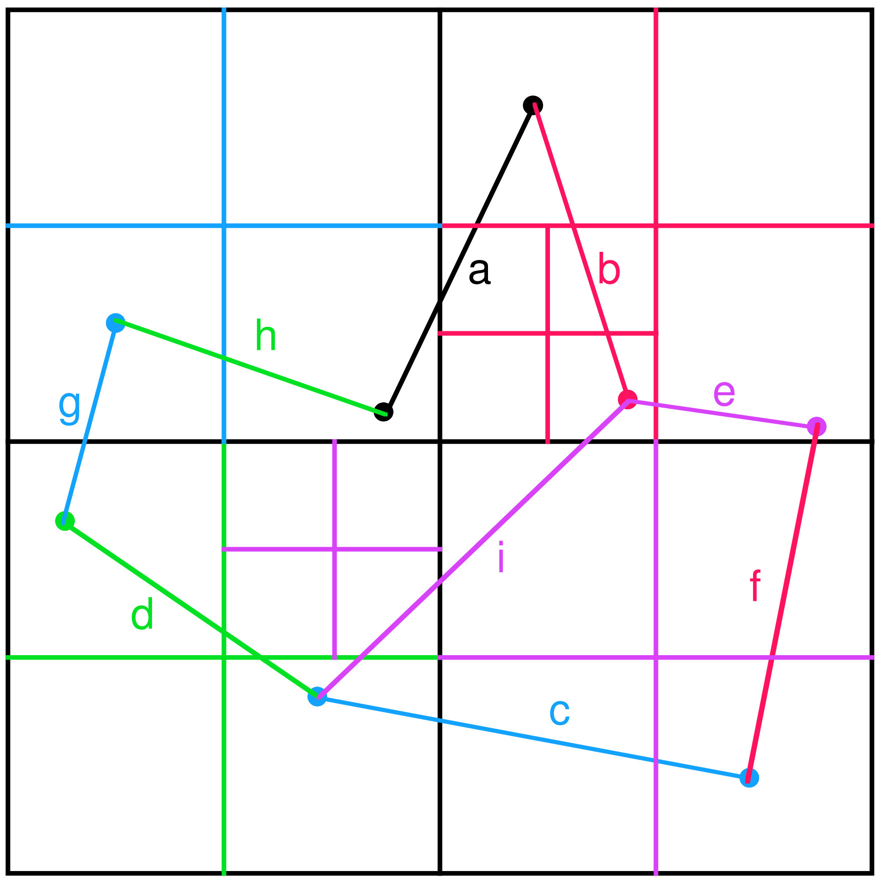
- Use ellipses as bounding boxes instead of rectangles
  1. place the foci at the endpoints of each subarc
  2. the principal axis is the length of the subarc
  3. advantage over the strip tree as no need for special provision for closed curves or curves that extend past their endpoints
- Two curves are guaranteed to intersect if the principal axes of their two bounding ellipses intersect, and the two foci of each bounding ellipse are external to the two foci of the other bounding ellipse



- Note that intersection of the principal axes of the bounding ellipses is not enough to guarantee intersection

PM1 QUADTREE

- Vertex-based (one vertex per block)



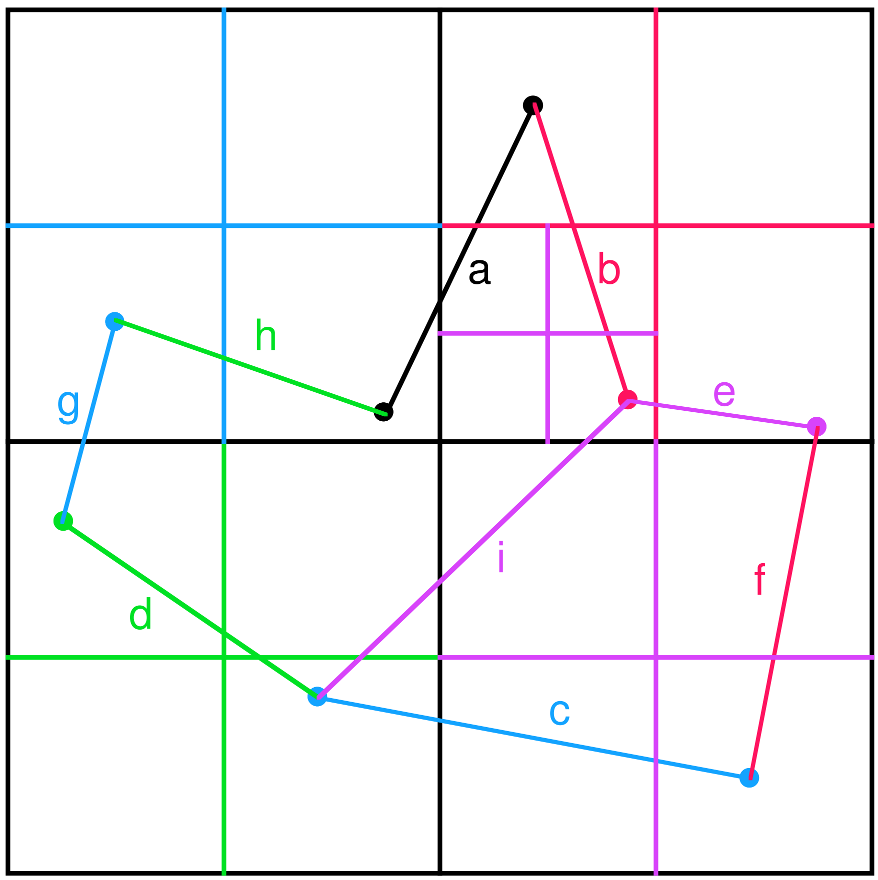
DECOMPOSITION RULE:

Partitioning occurs when a block contains more than one segment unless all the segments are incident at the same vertex which is also in the same block

- Shape independent of order of insertion

PM2 QUADTREE

- Vertex-based (one vertex per block)



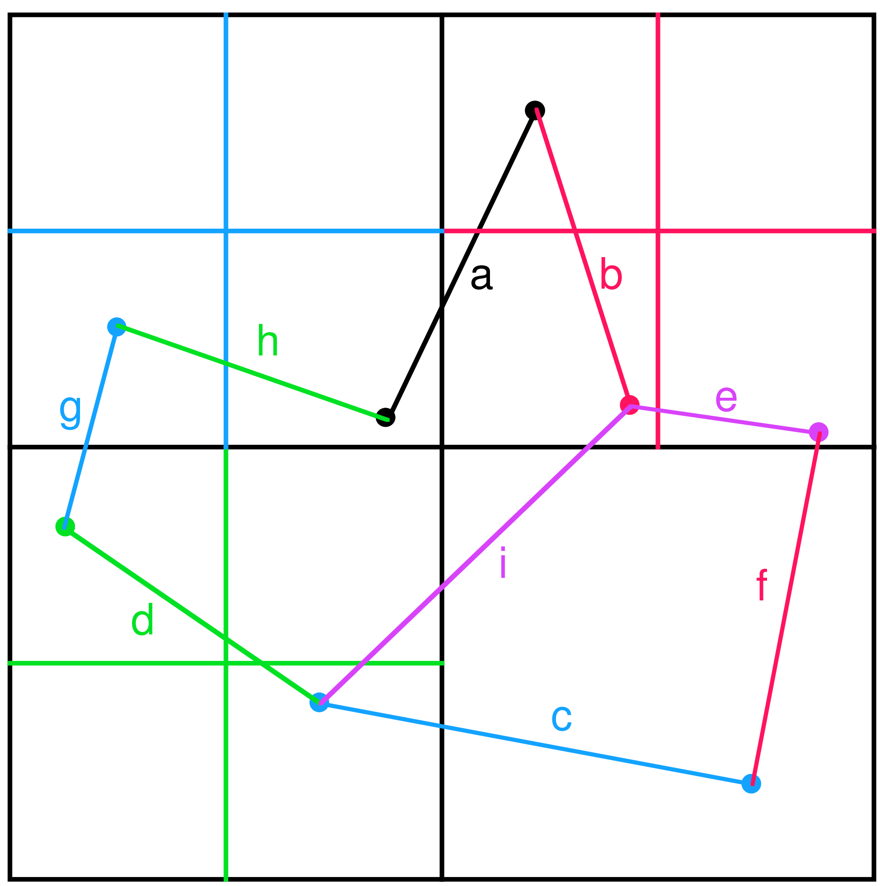
DECOMPOSITION RULE:

Partitioning occurs when a block contains more than one line segment unless all the segments are incident at the same vertex (the vertex can be in another block!)

- Shape independent of order of insertion

PM3 QUADTREE

- Vertex-based (one vertex per block)



DECOMPOSITION RULE:

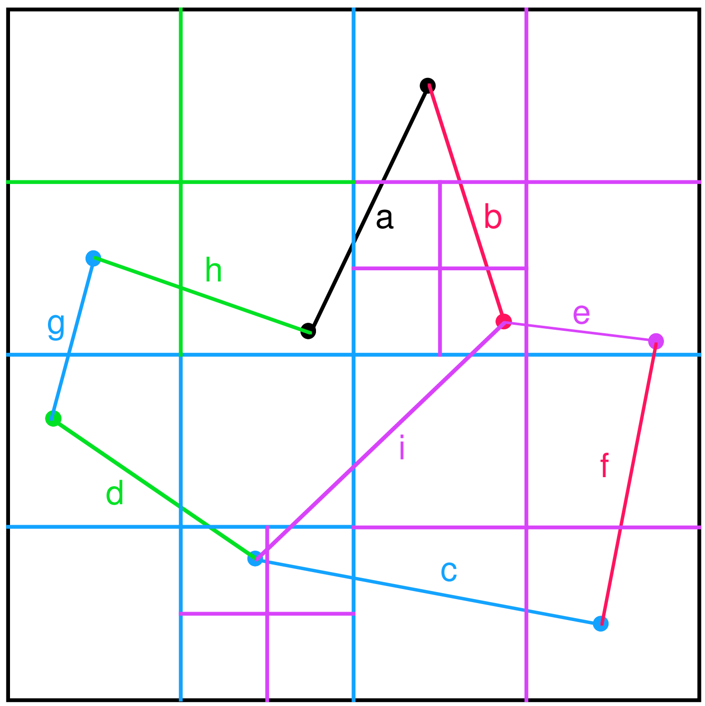
Partitioning occurs when a block contains more than one vertex (i.e., a PR quadtree with edges)

- Shape independent of order of insertion

PMR QUADTREE

- Edge-based
- Avoids having to split many times when two vertices or lines are very close as in PM1 quadtree
- Probabilistic splitting and merging rules
- Uses a splitting threshold value — say *N*

Ex: $N = 2$



DECOMPOSITION RULE:

Split a block *once* if upon insertion the number of segments intersecting a block exceeds *N*

Merge a block with its siblings if the total number of line segments intersecting them is less than *N*

- Merges can be performed more than once
- Does not guarantee that each block will contain at most *N* line segments
- Splitting threshold is not the same as bucket capacity
- Shape depends on order of insertion

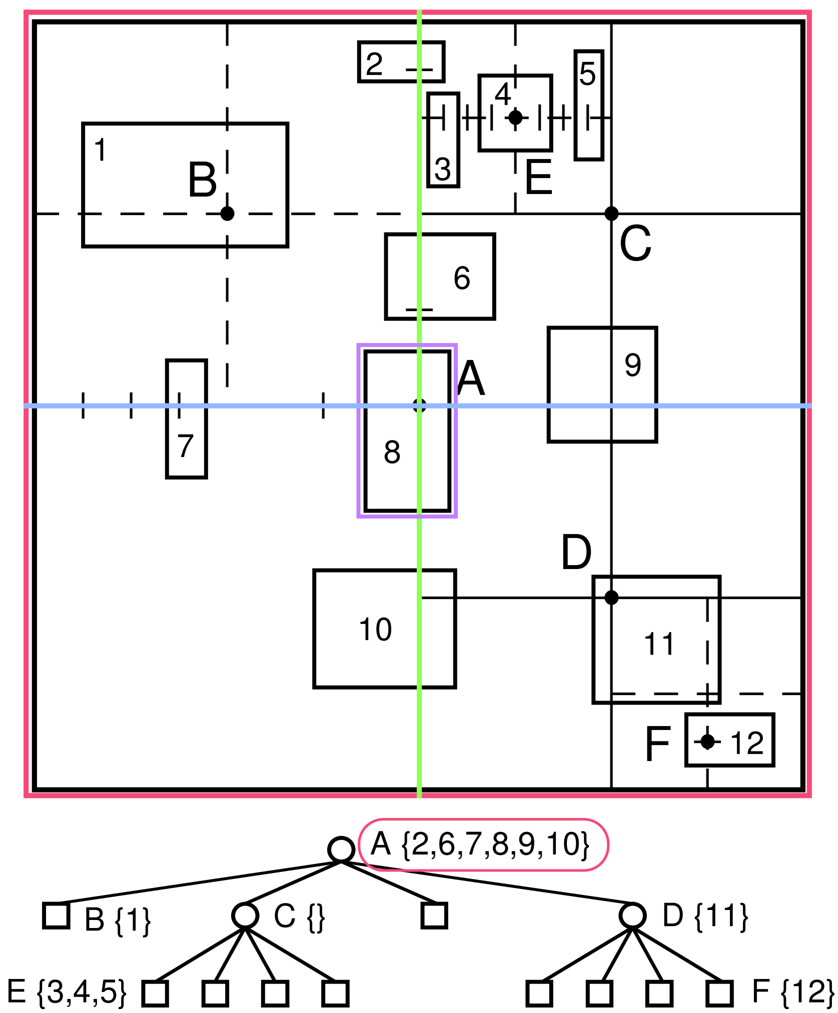
Outline

MX-CIF QUADTREE (Kedem)

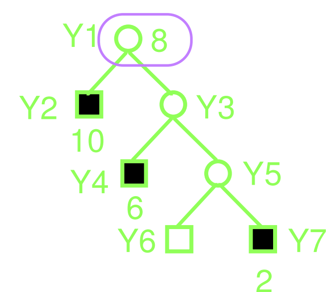
5 4 3 2 1
z v g r b

hp14

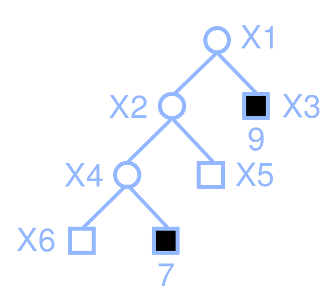
1. Collections of small rectangles for VLSI applications
2. Each rectangle is associated with its minimum enclosing quadtree block
3. Like hashing: quadtree blocks serve as hash buckets
4. Collision = more than one rectangle in a block
   - resolve by using two one-dimensional MX-CIF trees to store the rectangle intersecting the lines passing through each subdivision point
   - one for y-axis
   - if a rectangle intersects both *x* and *y* axes, then associate it with the *y* axis
   - one for x-axis



Binary tree for *y*-axis through A



Binary tree for *x*-axis through A



Loosen Quadtree Restrictions

- Conventional quadtree definition:
  - each cell contains only one object
  - objects cannot overlap
- Loosen this restriction
  - cell need not be entirely covered by an object
  - allow several objects to occupy a single cell
  - allow objects to overlap
  - allow arbitrary shapes

Halting a Quadtree Decomposition

1. Arbitrary shape may be decomposed infinitely when placed in certain locations
   - use object's minimum bounding box (MBB) to simplify representation
2. Coverage-based
   - restrict number of blocks that can cover an object
   - this will be the focus of our discussion
3. Density-based
   - restrict number of objects that can be covered by a block

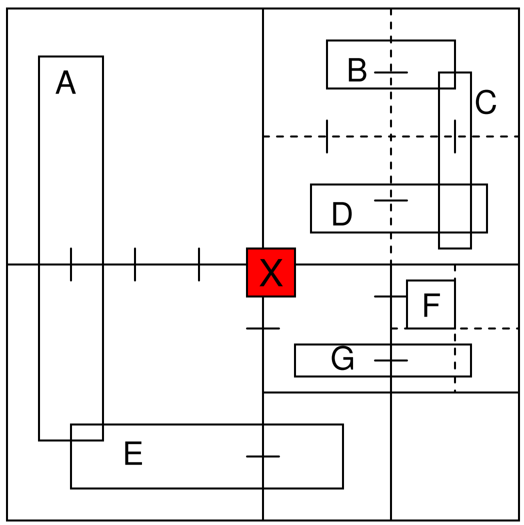
Using Quadtrees for Point Location

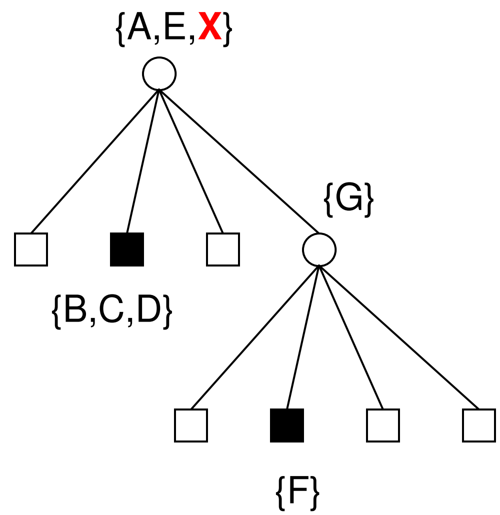
Associate MBB of object with quadtree block that contains it

1. May be non-leaf node
2. Limit to one block to avoid unnecessary searching
3. Use smallest containing block (i.e. the block at the deepest level) to avoid false hits

MX-CIF Quadtree

- Associate object with its min enclosing quadtree block
- Resolve collisions using two 1- X-CIF structures
- Drawback: small objects may be associated with much larger blocks
- Ex: every query will include x in its results
- Goal: Limit size of associated block



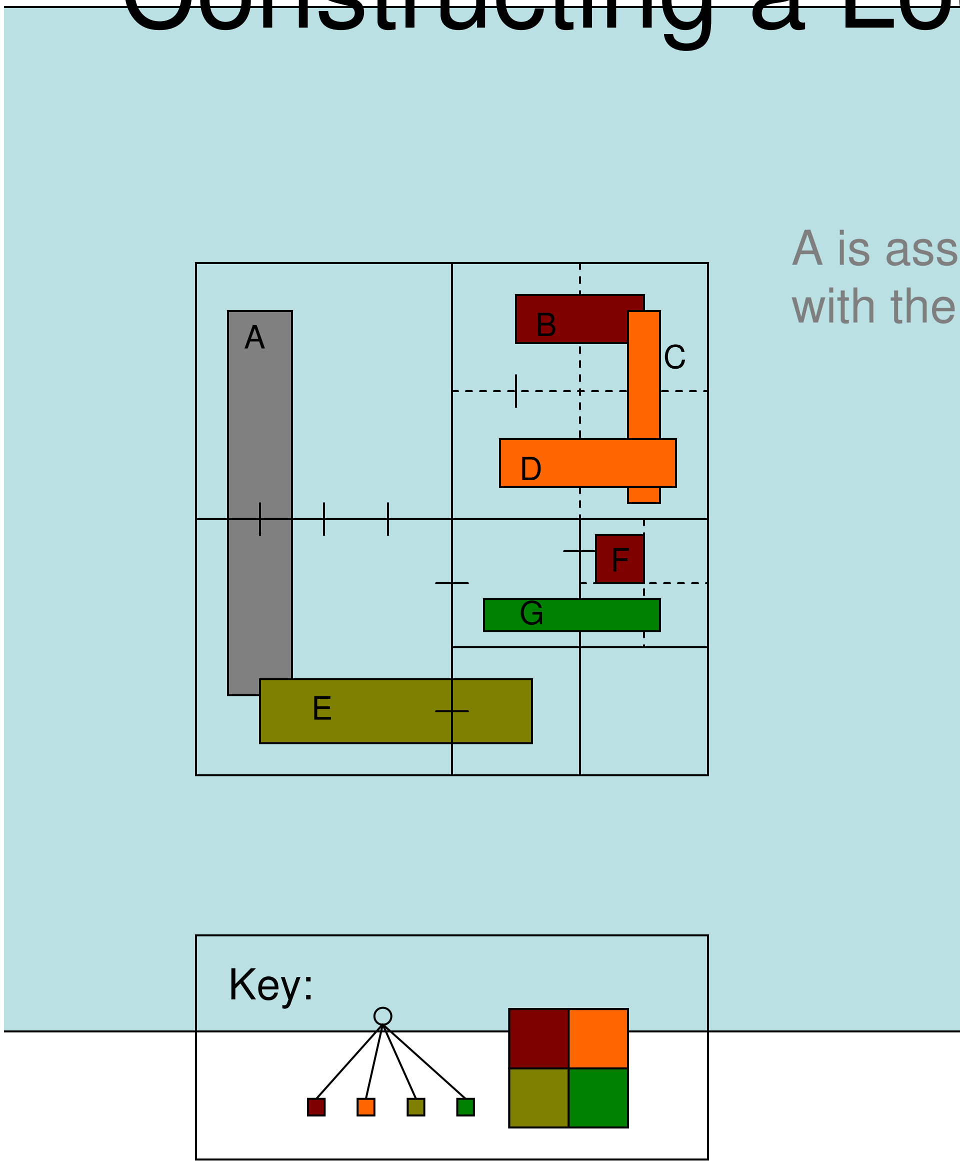


Loose Quadtree/Cover Fieldtree

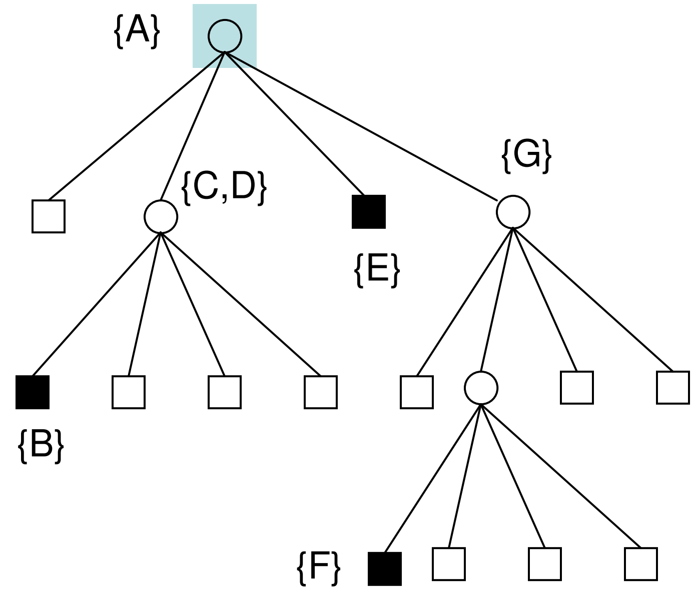
1. Problem: MBB of object is unrelated to size of its associated quadtree block
2. Uniformly expand space spanned by each block of width by a positive factor
   - object associated with its minimum enclosing expanded block
   - expanded width of is (1+p)•w
   - can show that the radius of the MBB of an object in is larger than
3. Each object associated with only one block
   - similar to QMAT (quadtree medial axis transform)

ft7o

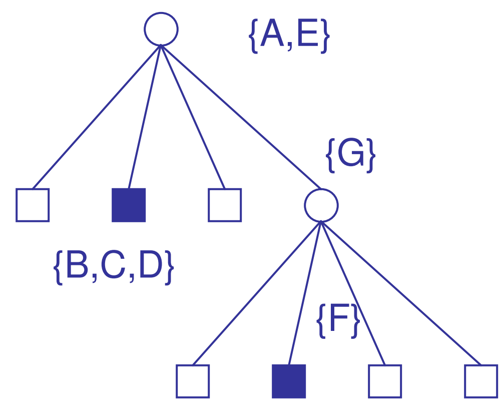
Constructing a Loose Quadtree



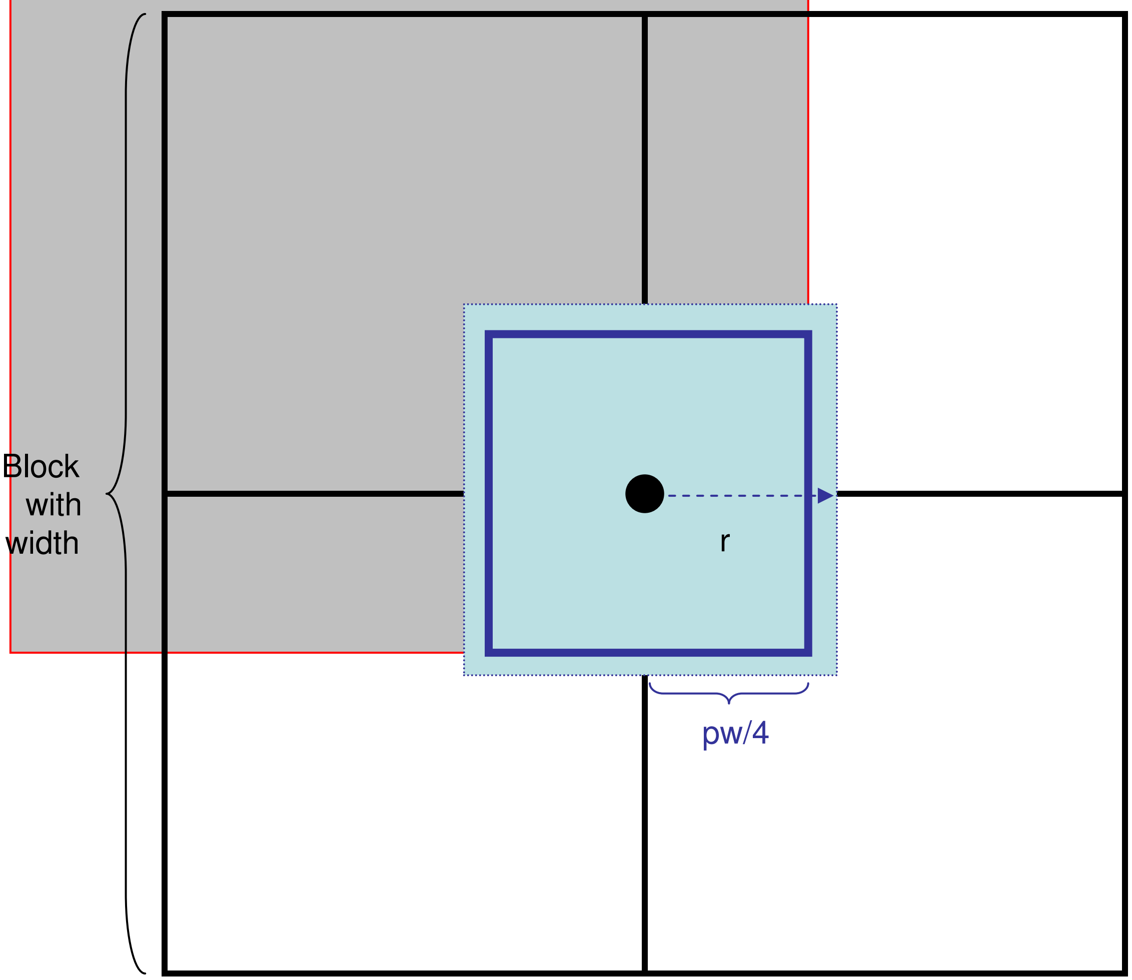
A is associated with the root node



Compare with MX-CIF Quadtree



Limits on Associated Object Radii



Block has width center
and expanded width

Claim: pw/4 = minimum radius of object associated with a block or width

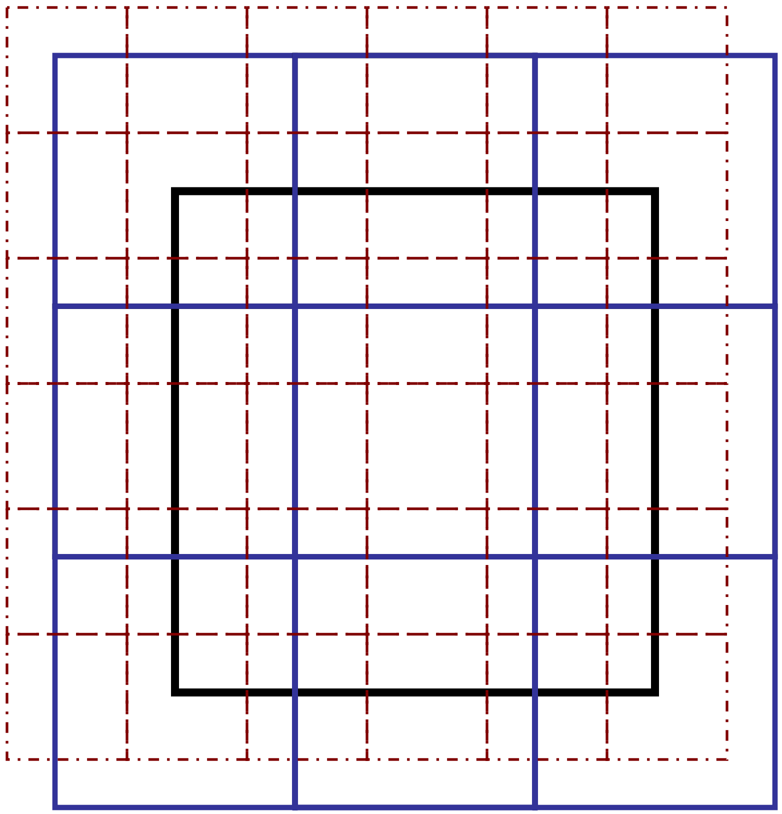
Proof: ’s children, have width and expanded width

Let be an object centered at with MBB radius

If
is associated with a
(or one of its children)
else
o may be associated with

Partition Fieldtree

- Hierarchy of grids with non-coincident origins
  - =block
  - each level forms a non-overlapping partitioning of the space
  - for each block of width that is being subdivided, the origins are shifted by



Partition Fieldtree: Observations

- Boundaries of blocks at different levels will never coincide
- Grids at different levels have a different origin
- Blocks at different levels do not form a refinement of those at a previous level
- Objects never need to be stored at more than three levels above their proper size
- KEY: Cover fieldtree achieves ratio bound by expanding the size of its blocks, while partition fieldtree achieves it by shifting the blocks

Ratio of MBB Size/ Block Size

1. Partition fieldtree
   - always bounded by at most three levels (8x object size)
2. Loose quadtree/Cover fieldtree
   - radius of MBB must be larger than p•w/4
   - as p decreases, the minimum radius decreases
   - p = 1/2, ratio is at most 4
   - p = 1/4, ratio is at most 8
   - p = 1/8, ratio is at most 16

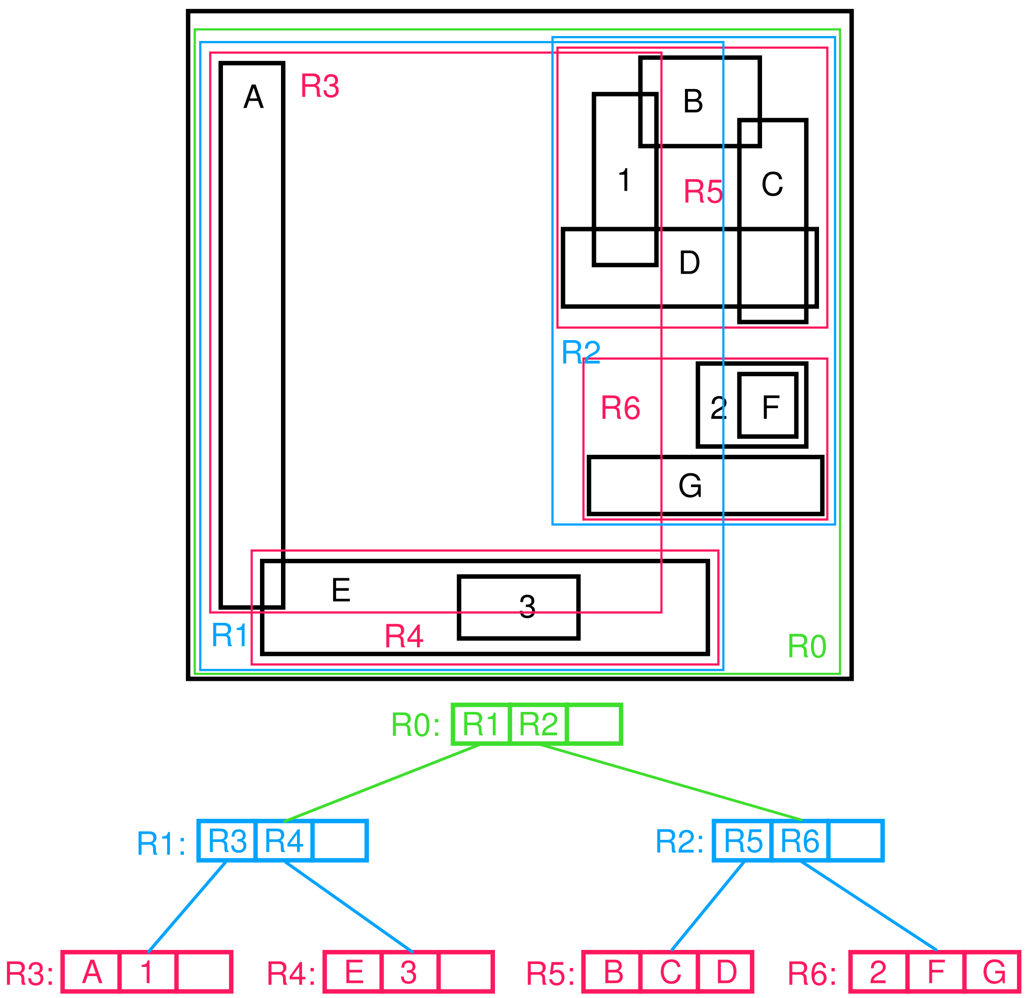
Partition fieldtree is superior when

Comparison

1. Partition fieldtree
   - number of partially covering quadtree subblocks is increased to at most 9 (27) for quadtrees (octrees) by offsetting their position while retaining their size
   - boundaries of quadtree blocks at different depths are never coincident
2. Loose quadtree/Cover fieldtree
   - area spanned by subblocks is expanded
   - boundaries of quadtree blocks can be coincident for some values of
     - e.g., when is a power of 2

MINIMUM BOUNDING RECTANGLES

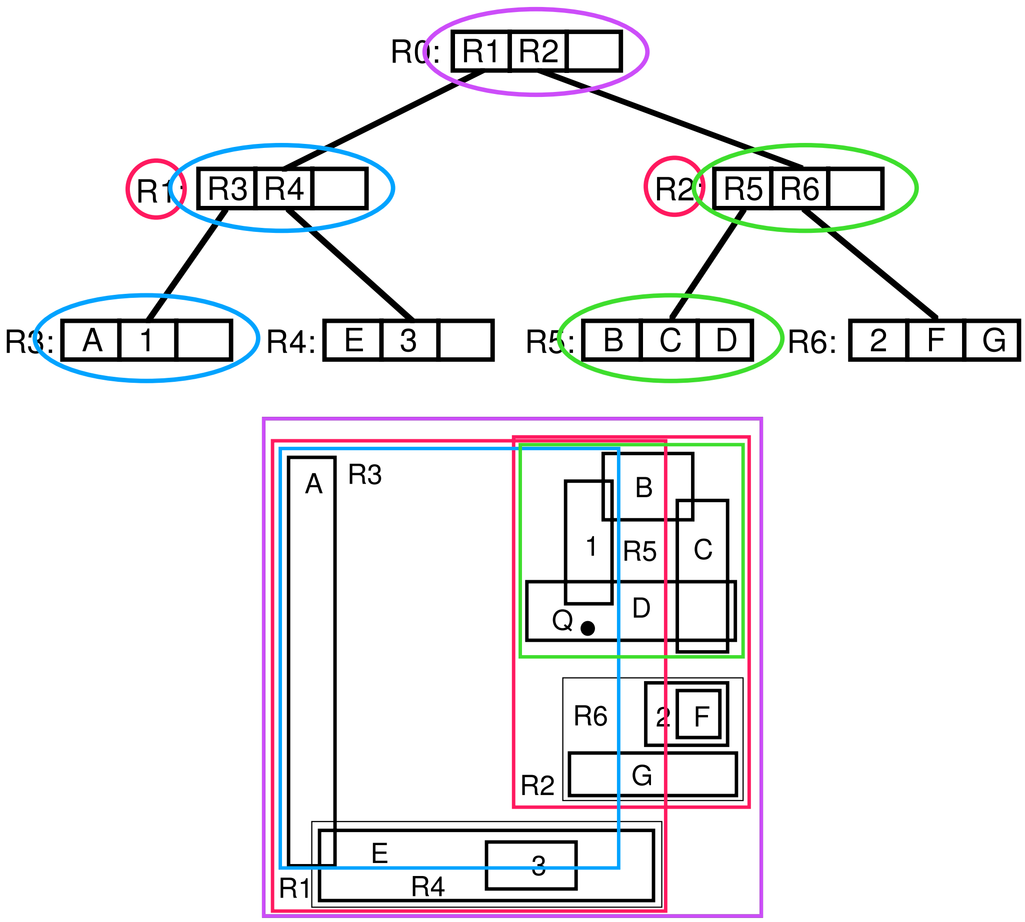
- Rectangles grouped into hierarchies, stored in another structure such as a B-tree
- Drawback: not a disjoint decomposition of space
- Rectangle has single bounding rectangle, yet area it spans may be included in several bounding rectangles
- May have to visit several rectangles to determine the presence/absence of a rectangle
- Order (*m*,*M*) R-tree
  1. between $m \leqslant \lceil M/2 \rceil$ and *M* entries in each node except root
  2. at least 2 entries in root unless a leaf node
- Ex: order (2,3) R-tree



SEARCHING FOR A RECTANGLE CONTAINING A POINT IN AN R-TREE

- Drawback is that may have to examine many nodes since a rectangle can be contained in the covering rectangles of many nodes yet its record is contained in only one leaf node (e.g., D in R0, R1, R2, R3, and R5)

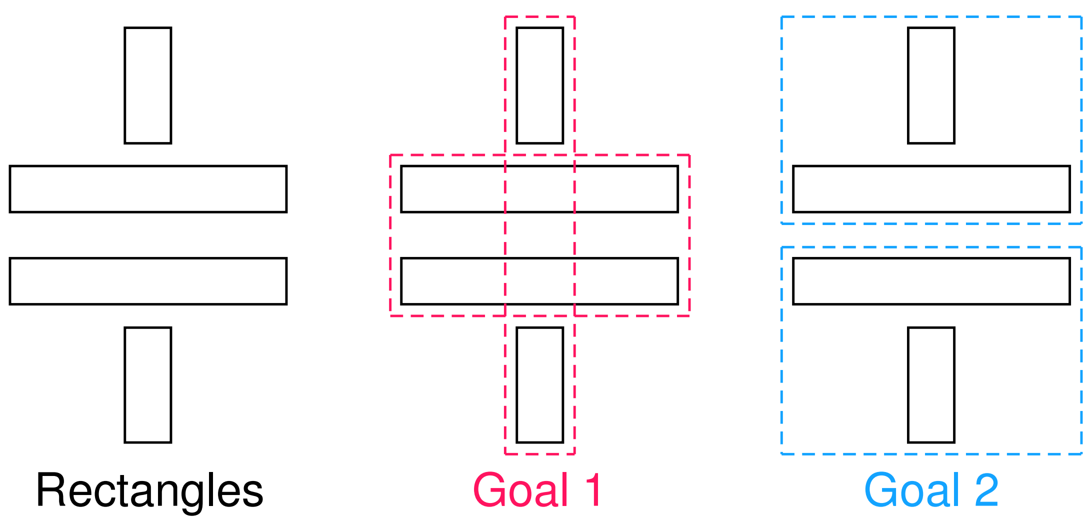
  Ex: Search for the rectangle containing point Q



- Q is in R0
- Q can be in both R1 and R2
- Searching R1 first means that R3 is searched but this leads to failure even though Q is in a part of D which is in R3
- Searching R2 finds that Q can only be in R5

DYNAMIC R-TREE CONSTRUCTION

- Differ by how to split overflowing node *p* upon insertion
- Conflicting goals:
  1. minimize number of children of *p* that must be visited by search operations
     - achieve by minimizing area common to children (overlap)
  2. reduce likelihood that each node *q* is visited by the search
     - achieve by minimizing total area spanned by bounding box of *q* (coverage)
- Ex:



EXAMPLE DYNAMIC SPLITTING METHODS

1. Methods based on reducing coverage:
   - exhaustive search
   - quadratic
   - linear
2. R*-tree
   - primary focus on reducing overlap
   - also reduces coverage by minimizing perimeter of bounding boxes of resulting nodes when effect on coverage is the same
   - when node overflows, first see if can avoid problem by reinserting a fraction of the nodes (e.g., 30%)
3. Ang/Tan: linear with focus on reduction of overlap
4. Non-packed methods that make use of an ordering
   - usually order centroids of bounding boxes of objects and build a B+-tree
     a. Hilbert packed R-tree: Peano-Hilbert order
     b. Morton packed R-tree: Morton order
   - node overflow
     a. goals of minimizing coverage of overlap are not part of the splitting process
     b. do not make use of spatial extent of bounding boxes in determining how to split a node

R-TREE OVERFLOW NODE SPLITTING POLICIES

- Could use exhaustive search to look at all possible partitions
- Usually two stages:
  1. pick a pair of bounding boxes to serve as seeds for resulting nodes (‘seed-picking’)
  2. redistribute remaining nodes with goal of minimizing the growth of the total area (‘seed-growing’)
- Different algorithms of varying time complexity
  1. quadratic:
     - find two boxes *j* and *k* that would waste the most area if they were in the same node
     - for each remaining box *i*, determine the increase in area d_{ij} and d_{ik} of the bounding boxes of *j* and *k* resulting from the addition of *i* and add the box *r* for which $|d_{rj} - d_{rk}|$ is a maximum to the node with the smallest increase in area
     - rationale: find box with most preference for one of *j*, *k*
  2. linear:
     - find two boxes with greatest normalized separation along all of the dimensions
     - add remaining boxes in arbitrary order to box whose area is increased the least by the addition
  3. linear (Ang/Tan)
     - minimizes overlap
     - for each dimension, associate each box with the closest face of the box of the overflowing node
     - pick partition that has most even distribution
       a. if a tie, minimize overlap
       b. if a tie, minimize coverage

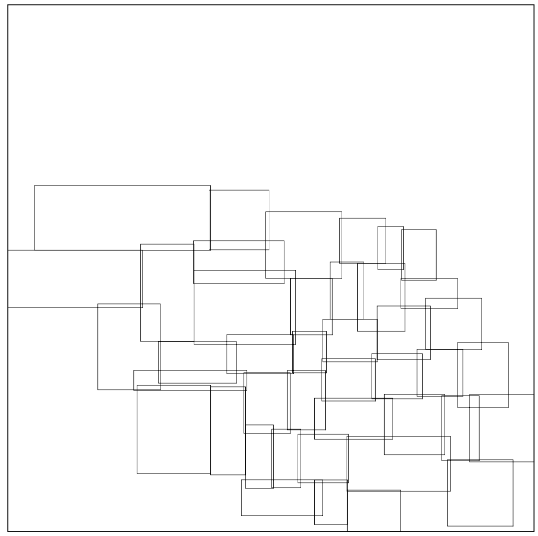
R*-TREE

- Tries to minimize overlap in case of leaf nodes and minimize increase in area for nonleaf nodes
- Changes from R-tree:
  1. insert into leaf node *p* for which the resulting bounding box has minimum increase in overlap with bounding boxes of *p*'s brothers
     - compare with R-tree where insert into leaf node for which increase in area is a minimum (minimizes coverage)
  2. in case of overflow in *p*, instead of splitting *p* as in R-tree, reinsert a fraction of objects in *p*
     - known as 'forced reinsertion' and similar to 'deferred splitting' or 'rotation' in B-trees
     - how do we pick objects to be reinserted? possibly sort by distance from center of *p* and reinsert furthest ones
  3. in case of true overflow, use a two-stage process
     - determine the axis along which the split takes place
       a. sort bounding boxes for each axis to get *d* lists
       b. choose the axis having the split value for which the sum of the perimeters of the bounding boxes of the resulting nodes is the smallest while still satisfying the capacity constraints (reduces coverage)
     - determine the position of the split
       a. position where overlap between two nodes is minimized
       b. resolve ties by minimizing total area of bounding boxes (reduces coverage)
- Works very well but takes time due to reinsertion

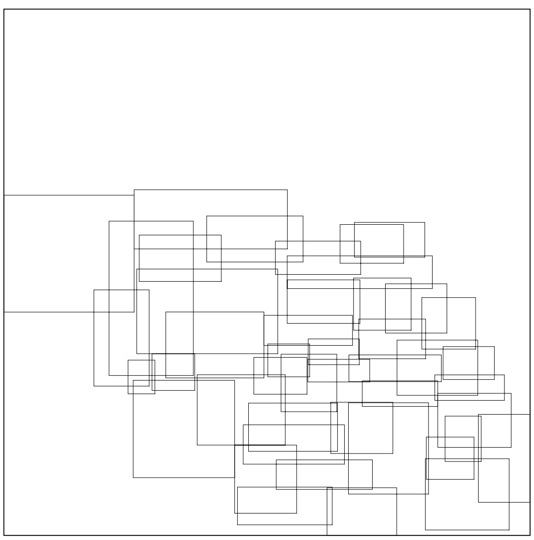
EXAMPLE OF R-TREE NODE SPLITTING POLICIES

- Sample collection of 1700 lines using m=20 and M=50

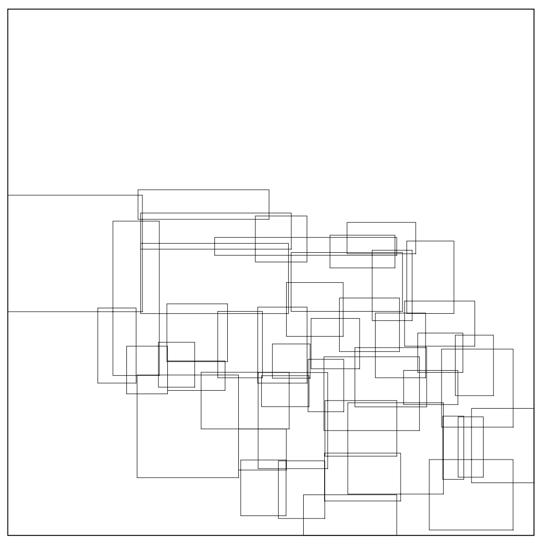
Collection of lines



R*-tree



Linear



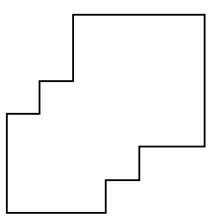
Quadratic

Outline

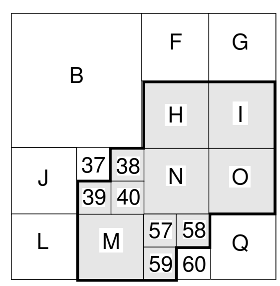
1. Introduction
2. Points
3. Lines
4. Rectangles
5. Regions
6. Surfaces and Volumes
7. Metric Data
8. Operations
9. Example system

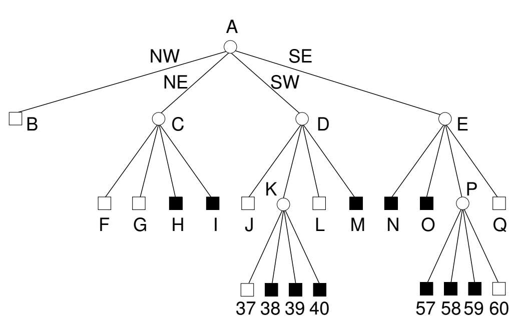
REGION QUADTREE

- Repeatedly subdivide until obtain homogeneous region
- For a binary image (BLACK ≡ 1 and WHITE ≡ 0)
- Can also use for multicolored data (e.g., a landuse class map associating colors with crops)
- Can also define data structure for grayscale images
- A collection of maximal blocks of size power of two and placed at predetermined positions
  1. could implement as a list of blocks each of which has a unique pair of numbers:
     - concatenate sequence of 2 bit codes corresponding to the path from the root to the block's node
     - the level of the block's node
  2. does not have to be implemented as a tree
     - tree good for logarithmic access
- A variable resolution data structure in contrast to a pyramid (i.e., a complete quadtree) which is a multiresolution data structure



0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	1	1	1	1
0	0	0	0	1	1	1	1
0	0	0	1	1	1	1	1
0	0	1	1	1	1	1	1
0	0	1	1	1	1	0	0
0	0	1	1	1	0	0	0



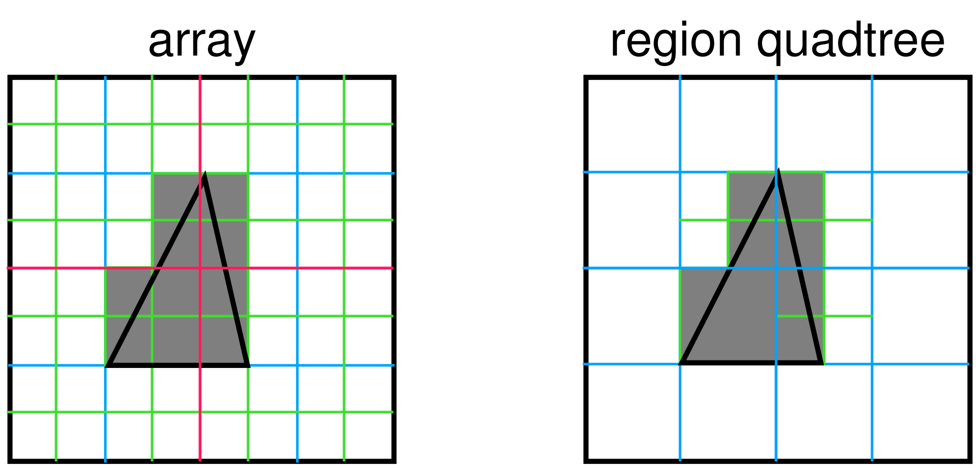


SPACE REQUIREMENTS

1. Rationale for using quadtrees/octrees is not so much for saving space but for saving execution time
2. Execution time of standard image processing algorithms that are based on traversing the entire image and performing a computation at each image element is proportional to the number of blocks in the decomposition of the image rather than their size
   - aggregation of space leads directly to execution time savings as the aggregate (i.e., block) is visited just once instead of once for each image element (i.e., pixel, voxel) in the aggregate (e.g., connected component labeling)
3. If want to save space, then, in general, statistical image compression methods are superior
   - drawback: statistical methods are not progressive as need to transmit the entire image whereas quadtrees lend themselves to progressive approximation
   - quadtrees, though, do achieve compression as a result of use of common subexpression elimination techniques
     a. e.g., checkerboard image
     b. see also vector quantization
4. Sensitive to positioning of the origin of the decomposition
   - for an n x n image, the optimal positioning requires an $O(n^2 \log_2 n)$ dynamic programming algorithm (Li, Grosky, and Jain)

DIMENSION REDUCTION

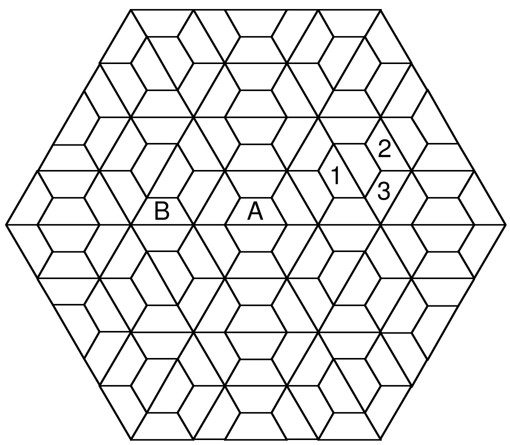
1. Number of blocks necessary to store a simple polygon as a region quadtree is proportional to its perimeter (Hunter)
   - implies that many quadtree algorithms execute in *O*(perimeter) time as they are tree traversals
   - the region quadtree is a dimension reducing device as perimeter (ignoring fractal effects) is a one-dimensional measure and we are starting with two-dimensional data
   - generalizes to higher dimensions
     a. region octree takes *O* (surface area) time and space (Meagher)
     b. *d*-dimensional data take time and space proportional to a *O* (d-1)-dimensional quantity (Walsh)
2. Alternatively, for a region quadtree, the space requirements double as the resolution doubles
   - in contrast with quadrupling in the array representation
   - for a region octree the space requirements quadruple as the resolution doubles
   - ex.

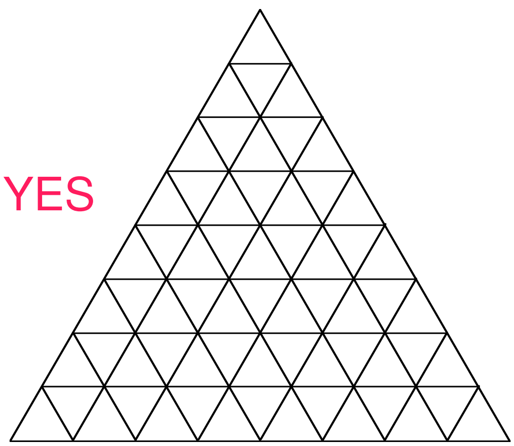


   - easy to see dependence on perimeter as decomposition only takes place on the boundary as the resolution increases

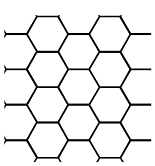
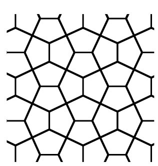
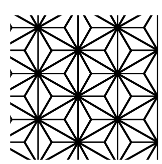
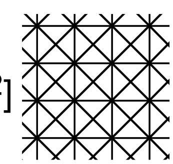
ALTERNATIVE DECOMPOSITION METHODS

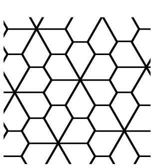
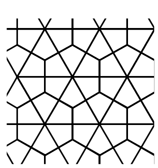
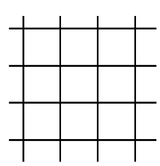
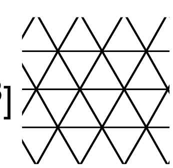
- A planar decomposition for image representation should be:
  1. infinitely repetitive
  2. infinitely decomposable into successively finer patterns
- Classification of tilings (Bell, Diaz, Holroyd, and Jackson)
  1. isohedral — all tiles are equivalent under the symmetry group of the tiling (i.e., when stand in one tile and look around, the view is independent of the tile)

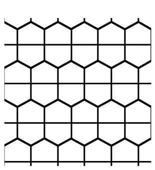
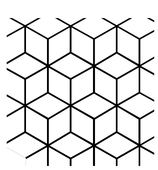
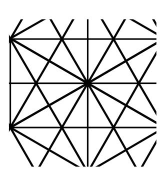
NO 

YES 

  2. regular — each tile is a regular polygon
- There are 81 types if classify by their symmetry groups
- Only 11 types if classify by their adjacency structure

$[3^6]$  $[3^2.4.3.2]$  $[3.12^2]$  $[4.8^2]$ 

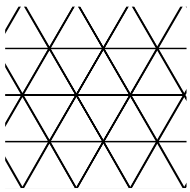
$[3^4.6]$  $[3.4.6.4]$  $[4^4]$  $[6^3]$ 

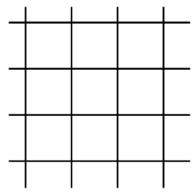
$[3^3.4^2]$  $[3.6.3.6]$  $[4.6.12]$ 

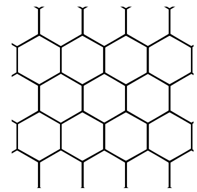
- $[3.12^2]$ means 3 edges at the first vertex of the polygonal tile followed by 12 edges at the next two vertices

PROPERTIES OF TILINGS — SIMILARITY

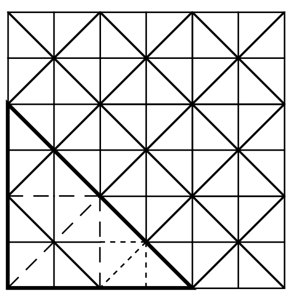
- Similarity — a tile at level *k* has the same shape as a tile at level 0 (basic tile shape)

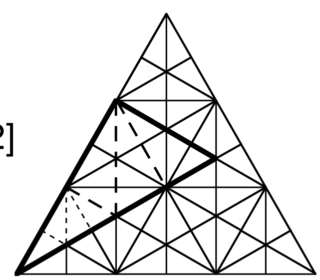
[6^3]

YES

[4^4]

YES

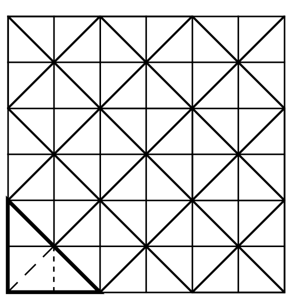
[3^6]

NO

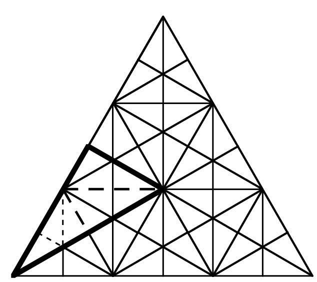
- Limited ≡ NOT similar (i.e., cannot be decomposed infinitely into smaller tiles of the same shape)
- Unlimited: each edge of each tile lies on an infinitely straight line composed entirely of edges
- Only 4 unlimited tilings [4^4], [6^3], [4.8^2], and [4.6.12]

[4.8^2]


[4.6.12]


- Two additional hierarchies:

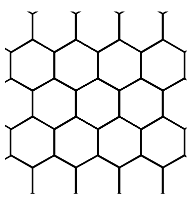

rotation of 135° between levels

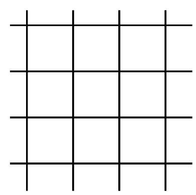

reflection between levels

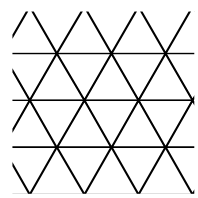
Note: [4.8^2] and [4.6.12] are not regular

PROPERTIES OF TILINGS — ADJACENCY

- Adjacency — two tiles are neighbors if they are adjacent along an edge or at a vertex
- Uniform adjacency ≡ distances between the centroid of one tile and the centroids of all its neighbors are the same
- Adjacency number of a tiling (A) ≡ number of different adjacency distances

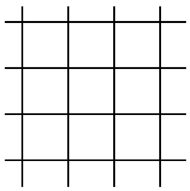
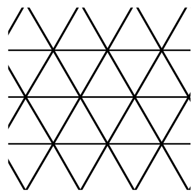
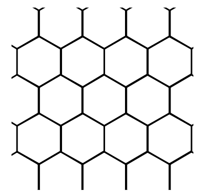
$[3^6]$

A=1

$[4^4]$

A=2

$[6^3]$

A=3

PROPERTIES OF TILINGS — UNIFORM ORIENTATION

- Uniform orientation
- All tiles with the same orientation can be mapped into each other by translations of the plane which do not involve rotation for reflection

[4^4]  [6^3]  [3^6] 

YES NO YES

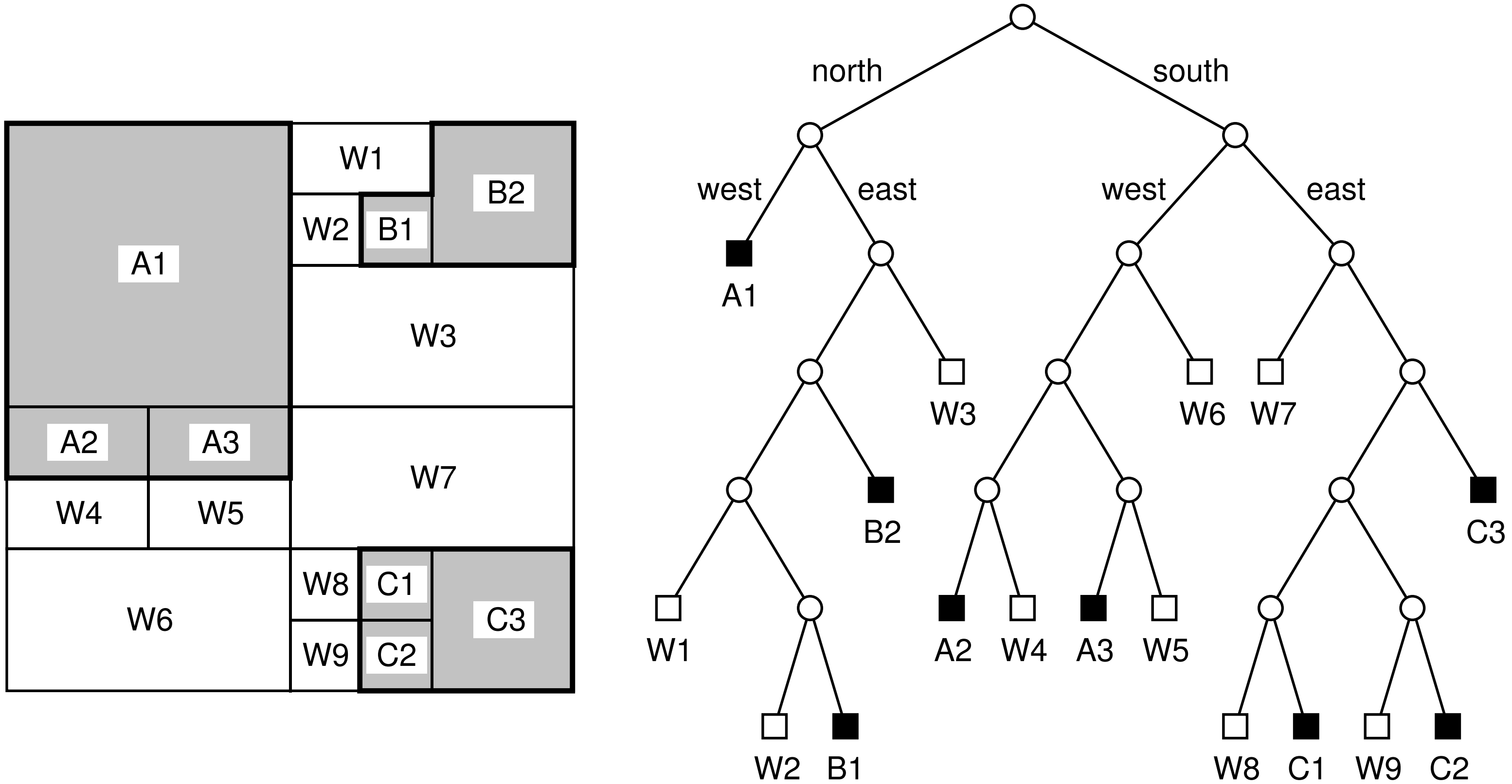
Conclusion:

- [4^4] has a lower adjacency number than [6^3]
- [4^4] has a uniform orientation while [6^3] does not
- [4^4] is unlimited while [3^6] is limited

  Use [4^4]!

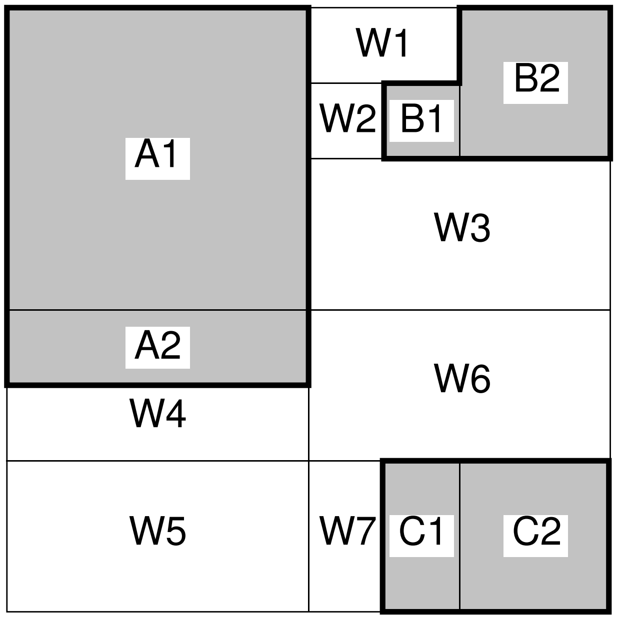
Bintree

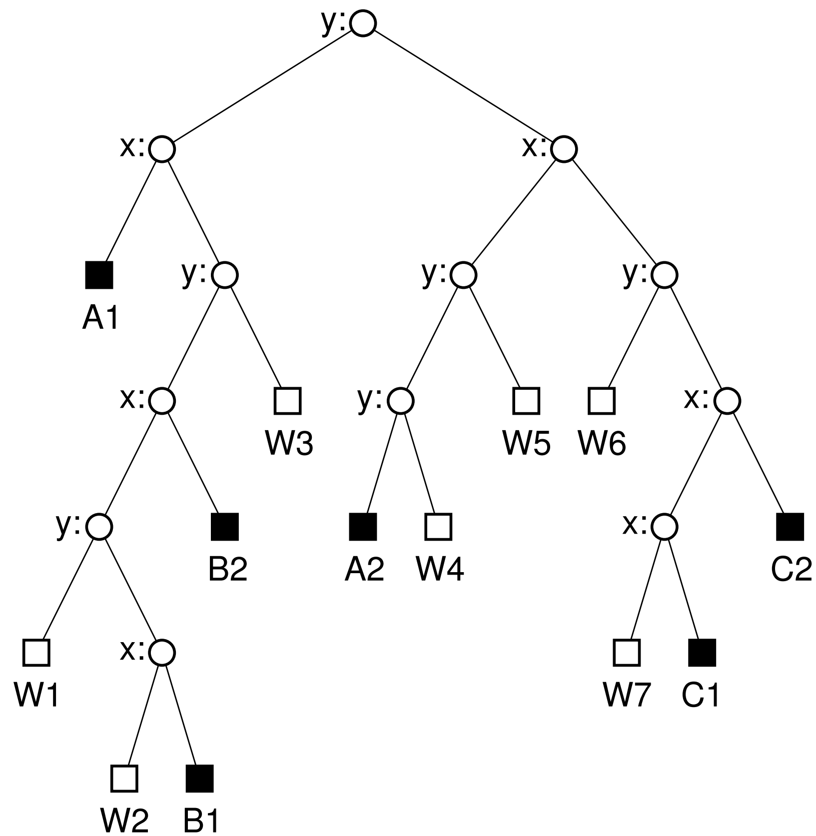
- Regular decomposition k-d tree
- Cycle through attributes



Generalized Bintree

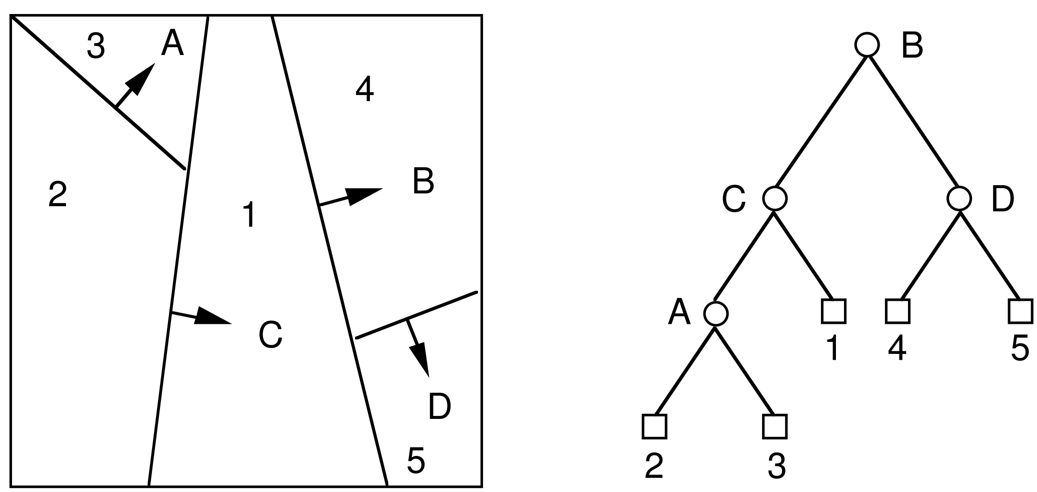
- Regular decomposition k-d tree but no need to cycle through attributes
- Need to record identity of partition axis at each nonleaf node





BSP TREES (Fuchs, Kedem, Naylor)

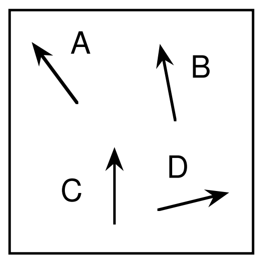
- Like a bintree except that the decomposition lines are at arbitrary orientations (i.e., they need not be parallel or orthogonal)
- For data of arbitrary dimensions
- In 2D (3D), partition along the edges (faces) of a polygon (polyhedron)
- Ex: arrows indicate direction of positive area



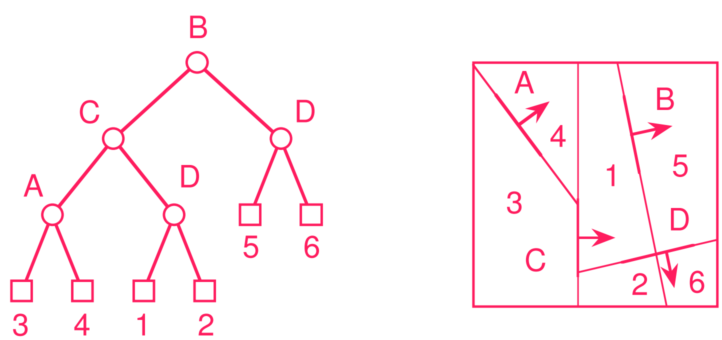
- Usually used for hidden-surface elimination
  1. domain is a set of polygons in three dimensions
  2. position of viewpoint determines the order in which the BSP tree is traversed
- A polygon's plane is extended infinitely to partition the entire space

DRAWBACKS OF BSP TREES

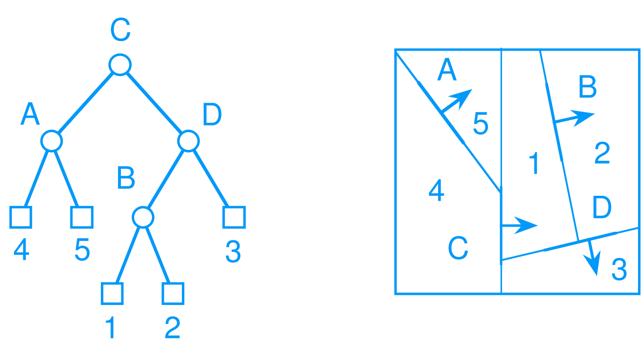
- A polygon may be included in both the left and right subtrees of node
- Same issues of duplicate reporting as in representations based on a disjoint decomposition of the underlying space
- Shape of the BSP tree depends on the order in which the polygons are processed and on the polygons chosen to serve as the partitioning plane
- Not based on a regular decomposition thereby complicating the performance of set-theoretic operations
- Ex: use line segments in two dimensions



1. partition induced by choosing B as the root



2. partition induced by choosing C as the root

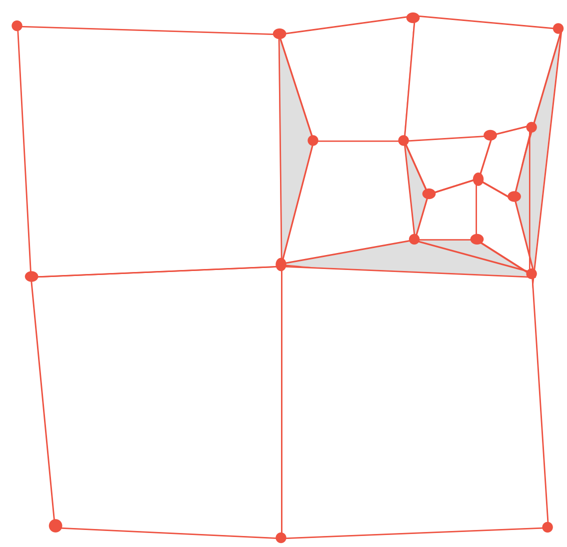


Outline

1. Introduction
2. Points
3. Lines
4. Rectangles
5. Regions
6. Surfaces and Volumes
7. Metric Data
8. Operations
9. Example system

HIERARCHICAL RECTANGULAR DECOMPOSITION

- Similar to triangular decomposition
- Good when data points are the vertices of a rectangular grid
- Drawback is absence of continuity between adjacent patches of unequal width (termed the *alignment problem*)

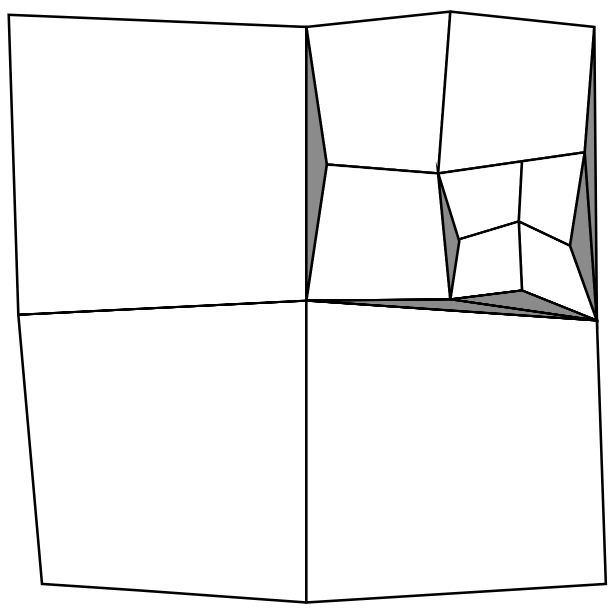


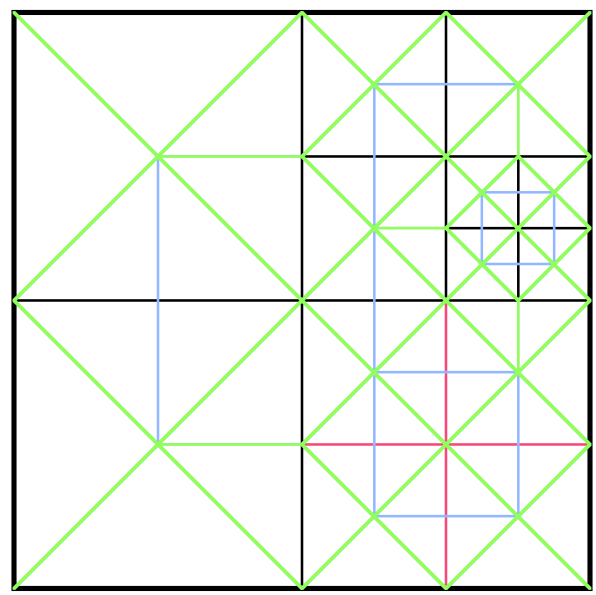
- Overcoming the presence of cracks
  1. use the interpolated point instead of the true point (Barrera and Hinjosa)
  2. triangulate the squares (Von Herzen and Barr)
     - can split into 2, 4, or 8 triangles depending on how many lines are drawn through the midpoint
     - if split into 2 triangles, then cracks still remain
     - no cracks if split into 4 or 8 triangles

RESTRICTED QUADTREE (VON HERZEN/BARR)

- All 4-adjacent blocks are either of equal size or of ratio 2:1

  Note: also used in finite element analysis to adptively refine an element as well as to achieve element compatibility (termed *h-refinement* by Kela, Perucchio, and Voelcker)



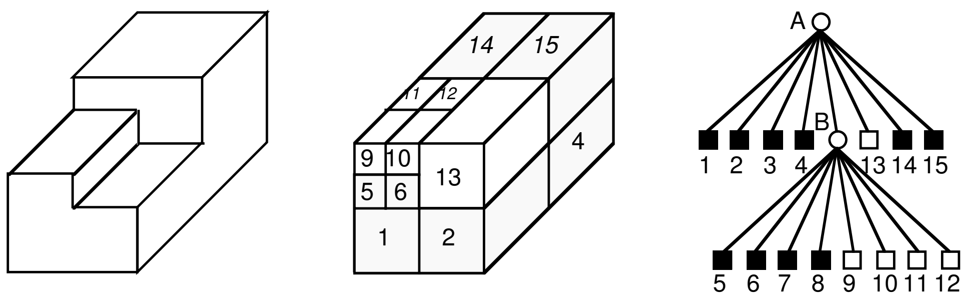


- 8-triangle decomposition rule
  1. decompose each block into 8 triangles (i.e., 2 triangles per edge)
  2. unless the edge is shared by a larger block
  3. in which case only 1 triangle is formed
- 4-triangle decomposition rule
  1. decompose each block into 4 triangles (i.e., 1 triangle per edge)
  2. unless the edge is shared by a smaller block
  3. in which case 2 triangles are formed along the edge
- Prefer 8-triangle rule as it is better for display applications (shading)

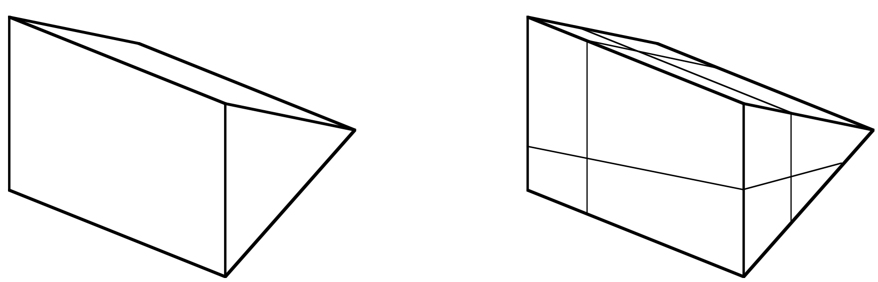
OCTREES

1. Interior (voxels)
   - analogous to region quadtree
   - approximate object by aggregating similar voxels
   - good for medical images but not for objects with planar faces

   Ex:



2. Boundary
   - adaptation of PM quadtree to three-dimensional data
   - decompose until each block contains
     a. one face
     b. more than one face but all meet at same edge
     c. more than one edge but all meet at same vertex
   - impose a spatial index on a boundary model (BRep)



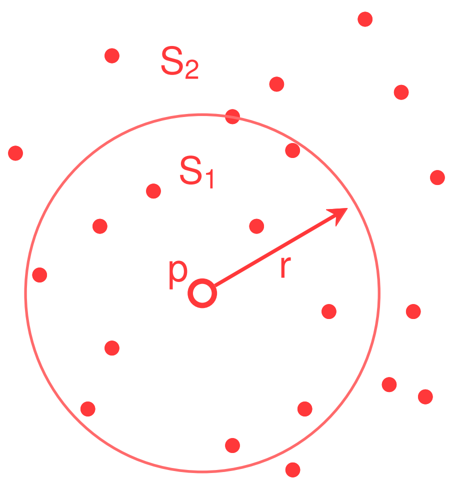
Outline

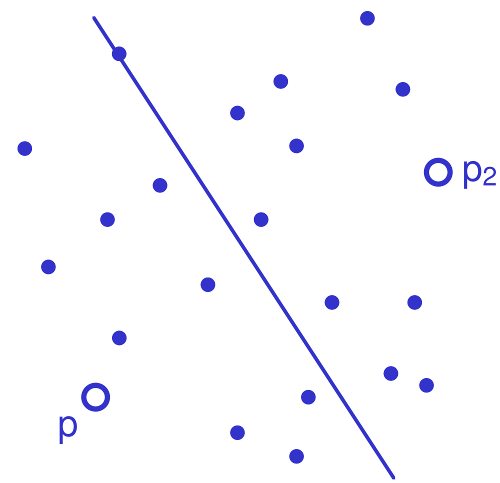
Basic Definitions

1. Often only information available is a distance function indicating degree of similarity (or dis-similarity) between all pairs of N data objects
2. Distance metric d: objects must reside in finite metric space (S, d) where for o_1, o_2, o_3 in S, d must satisfy
   - $d(o_1, o_2) = d(o_2, o_1)$ (symmetry)
   - $d(o_1, o_2) \geq 0$, $d(o_1, o_2) = 0$ iff $o_1 = o_2$ (non-negativity)
   - $d(o_1, o_3) \leq d(o_1, o_2) + d(o_2, o_3)$ (triangle inequality)
3. Triangle inequality is a key property for pruning search space
4. Non-negativity property enables ignoring negative values in derivations

Pivots

- Identify a distinguished object or subset of the objects termed <u>pivots</u> or <u>vantage points</u>
  1. sort remaining objects based on distances from pivots and build index
  2. use index to achieve pruning of other objects during search
- Given pivot $p \in S$, for all objects $o \in S' \subseteq S$, we know:
  1. exact value of $d(p, o)$,
  2. $d(p, o)$ lies within range $[r_{\text{lo}}, r_{\text{hi}}]$ of values (ball partitioning) or
     - drawback is asymmetry of partition as outer shell is usually narrow
  3. o is closer to p than to some other object $p_2 \in S$ (generalized hyperplane partitioning)
- Distances from pivots are useful in pruning the search



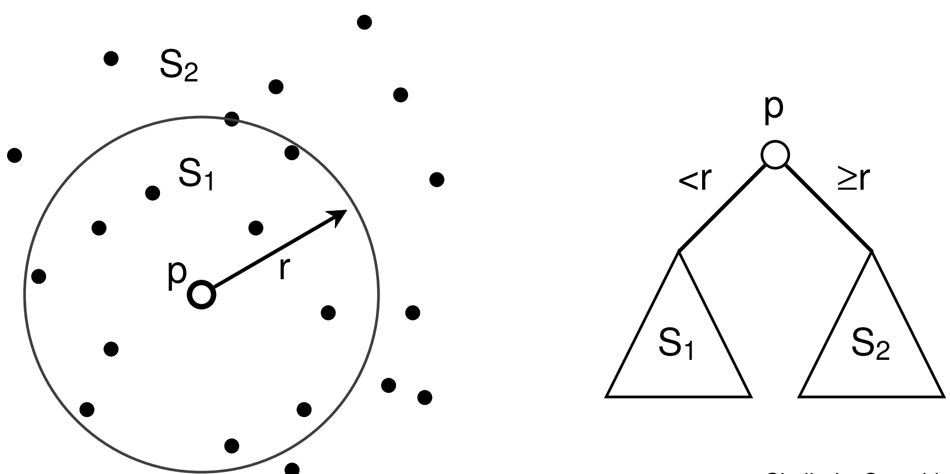


vp-Tree (Metric Tree; Uhlmann|Yianilos)

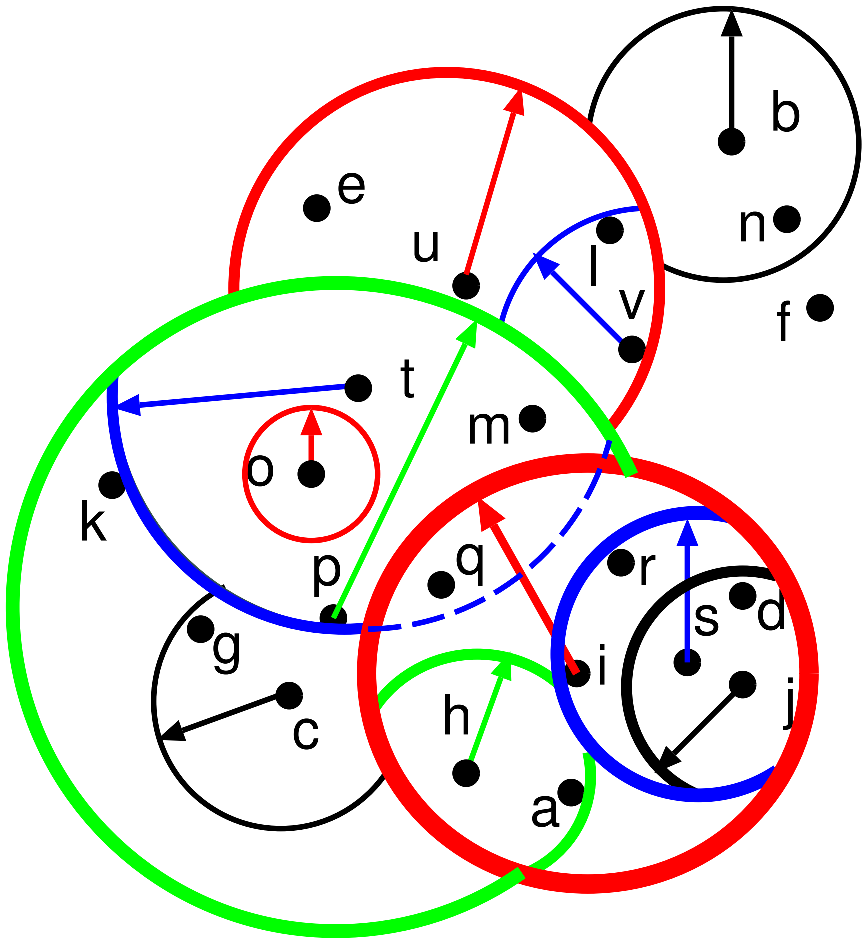
- Ball partitioning method
- Pick p from S and let r be median of distances of other objects from p
- Partition S into two sets S_1 and S_2 where:

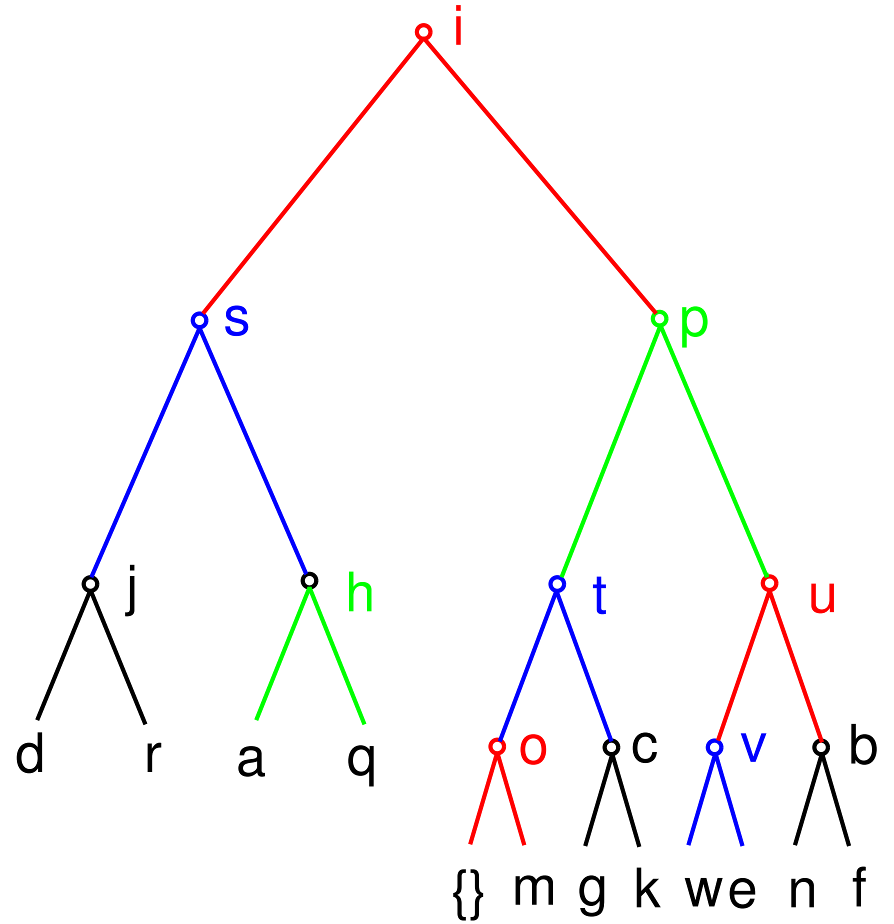
$$\begin{aligned} S_1 &= \{o \in S \setminus \{p\} \mid d(p,o) < r\} \\ S_2 &= \{o \in S \setminus \{p\} \mid d(p,o) \geq r\} \end{aligned}$$

- Apply recursively, yielding a binary tree with pivot and radius values at internal nodes
- Choosing pivots
  1. simplest is to pick at random
  2. choose a random sample and then select median



vp-Tree Example



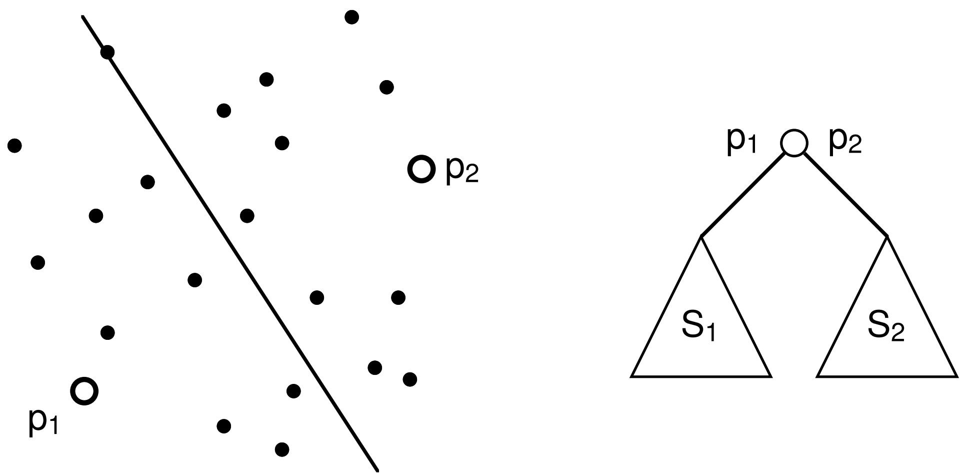


gh-Tree (Metric Tree; Uhlmann)

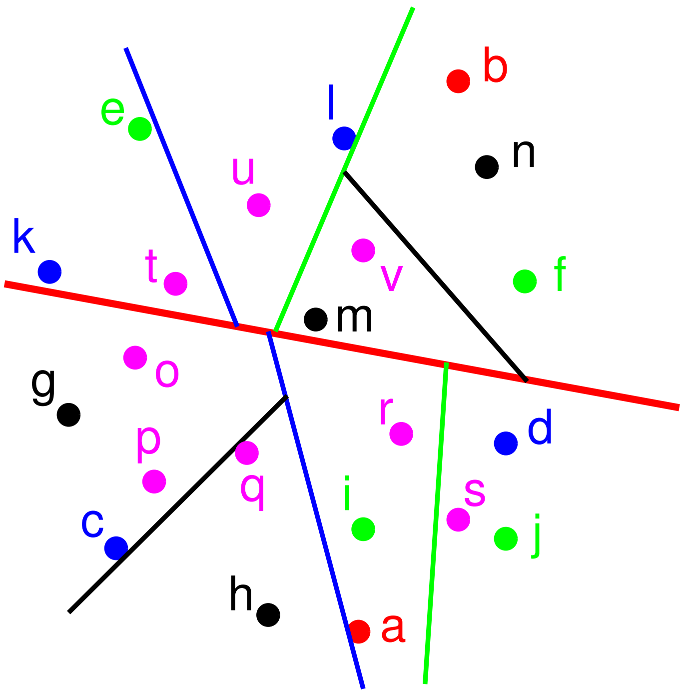
- Generalized hyperplane partitioning method
- Pick p_1 and p_2 from S and partition S into two sets S_1 and S_2 where:

$$\begin{aligned} S_1 &= \{o \in S \setminus \{p_1, p_2\} \mid d(p_1, o) \leq d(p_2, o)\} \\ S_2 &= \{o \in S \setminus \{p_1, p_2\} \mid d(p_2, o) < d(p_1, o)\} \end{aligned}$$

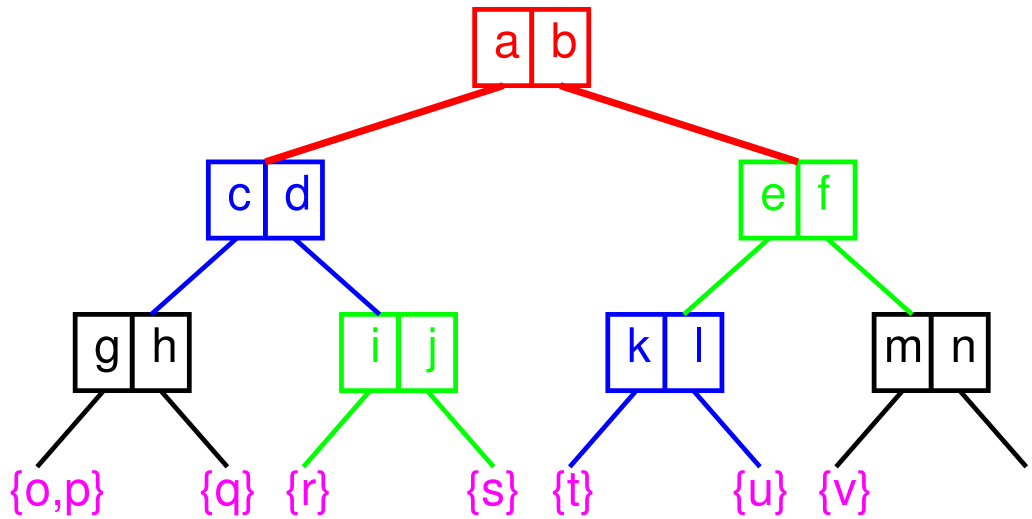
- Objects in S_1 are closer to p_1 than to p_2 (or equidistant from both), and objects in S_2 are closer to p_2 than to p_1
  - hyperplane corresponds to all points o satisfying $d(p_1, o) = d(p_2, o)$
  - can also “move” hyperplane, by using $d(p_1, o) = d(p_2, o) + m$
- Apply recursively, yielding a binary tree with two pivots at internal nodes



gh-Tree Example



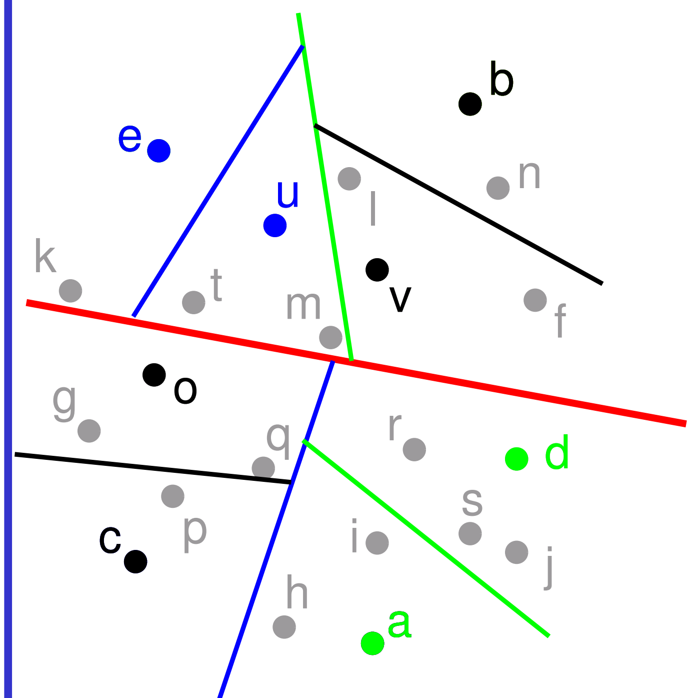
(a)



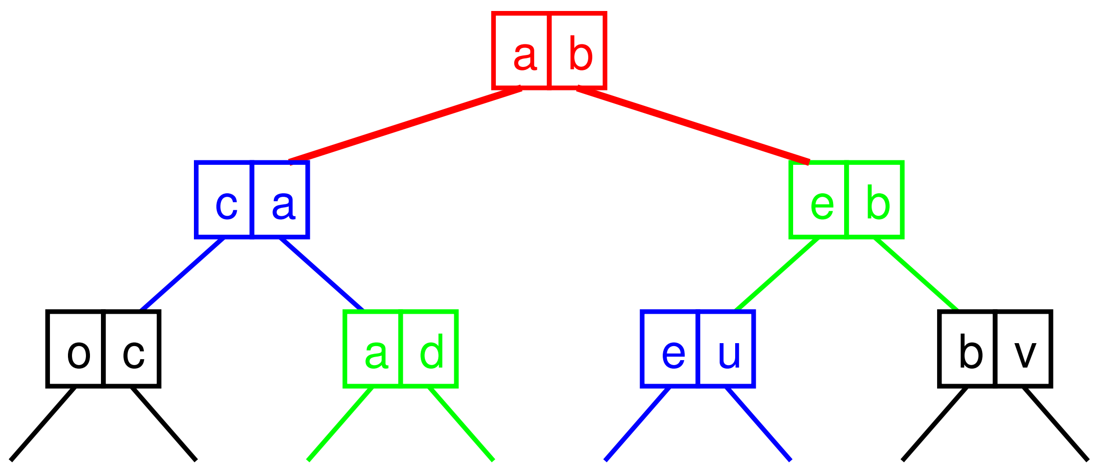
(b)

mb-Tree (Dehne/Noltemeier)

1. Inherit one pivot from ancestor node
2. Fewer pivots and fewer distance computations but perhaps deeper tree
3. Like bucket (k) PR k-d tree as split whenever region has $k > 1$ objects but region partitions are implicit (defined by pivot objects) instead of explicit



(a)



(b)

Outline

1. Introduction
2. Points
3. Lines
4. Rectangles
5. Regions
6. Surfaces and Volumes
7. Metric Data
8. Operations
9. Example system

Incremental Nearest Neighbors (Hjaltason/Samet)

- Motivation
  1. often don't know in advance how many neighbors will need
  2. e.g., want nearest city to Chicago with population > 1 million
- Several approaches
  1. guess some area range around Chicago and check populations of cities in range
     - if find a city with population > 1 million, must make sure that there are no other cities that are closer with population > 1 million
     - inefficient as have to guess size of area to search
     - problem with guessing is we may choose too small a region or too large a region
       a. if size too small, area may not contain any cities with right population and need to expand the search region
       b. if size too large, may be examining many cities needlessly
  2. sort all the cities by distance from Chicago
     - impractical as we need to re-sort them each time pose a similar query with respect to another city
     - also sorting is overkill when only need first few neighbors
  3. find k closest neighbors and check population condition

Mechanics of Incremental Nearest Neighbor Algorithm

- Make use of a search hierarchy (e.g., tree) where
  1. objects at lowest level
  2. object approximations are at next level (e.g., bounding boxes in an R-tree)
  3. nonleaf nodes in a tree-based index
- Traverse search hierarchy in a “best-first” manner similar to A*-algorithm instead of more traditional depth-first or breadth-first manners
  1. at each step, visit element with smallest distance from query object among all unvisited elements in the search hierarchy
     - i.e., all unvisited elements whose parents have been visited
  2. use a global list of elements, organized by their distance from query object
     - use a priority queue as it supports necessary insert and delete minimum operations
     - ties in distance: priority to lower type numbers
     - if still tied, priority to elements deeper in search hierarchy

Incremental Nearest Neighbor Algorithm

Algorithm:

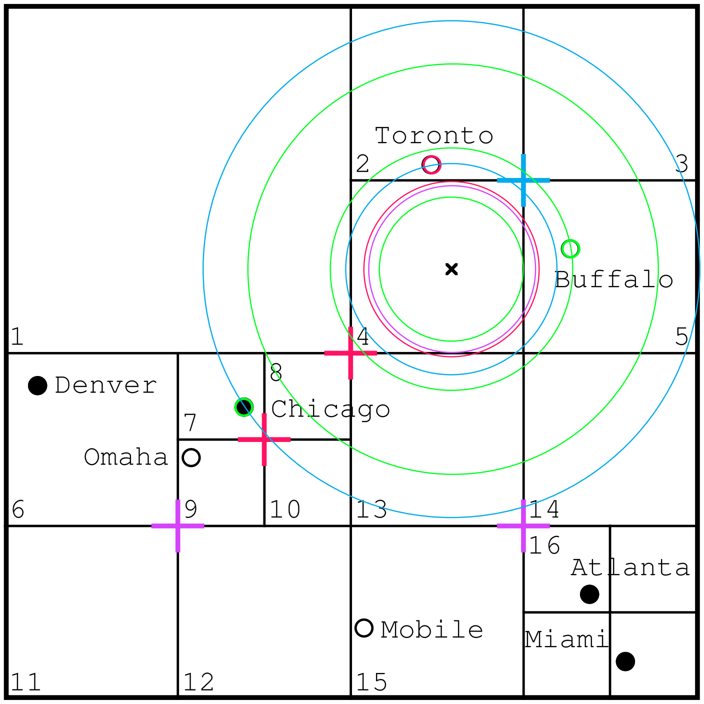
INCNEAREST(q, S, T)

1 $Q \leftarrow$ NEWPRIORITYQUEUE()
2 $e_t \leftarrow$ root of the search hierarchy induced by q, S, and T
3 ENQUEUE(Q, e_t, 0)
4 while not ISEMPTY(Q) do
5   $e_t \leftarrow$ DEQUEUE(Q)
6   if $t = 0$ then /* e_t is an object */
7   Report e_t as the next nearest object
8   else
9   for each child element $e_{t'}$ of e_t do
10   ENQUEUE(Q, $e_{t'}$, $d_{t'}(q,\ e_{t'})$)

1. Lines 1-3 initialize priority queue with root
2. In main loop take element e_t closest to q off the queue
   - report e_t as next nearest object if e_t is an object
   - otherwise, insert child elements of e_t into priority queue

EXAMPLE

- Find closest city to x=(65,62) with population ≥ 1 million



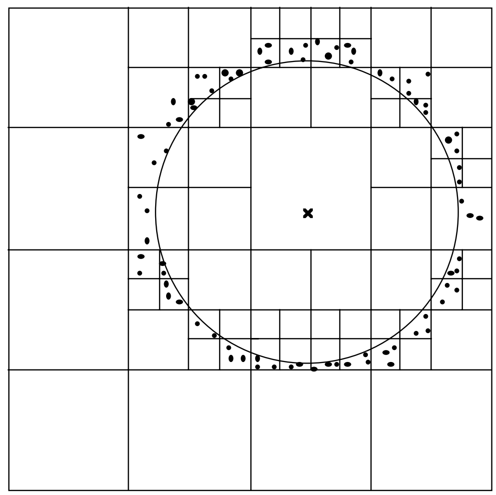
- Blocks labeled “depth/ NW-most descendant”
- Search circles correspond to block/feature being dequeued
- Legend
  - ● satisfies query
  - ○ doesn’t satisfy query

	City	Pop (1000)	Pos
	Atlanta	4,129	(85,15)
2	Buffalo	764	(82,65)
3	Chicago	6,532	(35,42)
	Denver	1,381	(5,45)
	Mobile	504	(52,10)
	Omaha	416	(27,35)
1	Toronto	904	(62,77)
	Miami	5,250	(90,5)

1. Initially, queue only contains root: [0/1]
2. Dequeue root and enqueue 4 sons: [1/2 1/13 1/1 1/6]
3. Dequeue 1/2 and enqueue 4 sons: [2/4 2/5 1/13 2/2 1/1 2/3 1/6]
4. Dequeue 2/4 (empty); dequeue 2/5 containing Buffalo; enqueue Buffalo: [1/13 2/2 1/1 2/3 Buffalo 1/6]
5. Dequeue 1/13 and enqueue 4 sons: [2/13 2/2 1/1 2/14 2/3 Buffalo 1/6 2/15 2/16]
6. Dequeue 2/13 (empty); dequeue 2/2 containing Toronto; enqueue Toronto: [1/1 Toronto 2/14 2/3 Buffalo 1/6 2/15 2/16]
7. Dequeue 1/1 which is empty; dequeue Toronto (population too small); dequeue 2/14 and 2/3 (empty): [Buffalo 1/6 2/15 2/16]
8. Dequeue Buffalo (population too small): [1/6 2/15 2/16]
9. Dequeue 1/6 and enqueue 4 sons: [2/7 2/15 2/16 2/12 2/6 2/11]
10. Dequeue 2/7 and enqueue 4 sons: [3/8 3/10 3/7 3/9 2/15 2/16 2/12 2/6 2/11]
11. Dequeue 3/8 and 3/10 (empty); dequeue 3/7 containing Chicago; enqueue Chicago: [Chicago 3/9 2/15 2/16 2/12 2/6 2/11]
12. Dequeue Chicago which satisfies the query

ANALYSIS

1. Assume constant time to calculate distance metric
   - true for point query objects but not necessarily for more complex features such as polygons
2. Assume a PMR quadtree
   - for line data, $O(N)$ blocks for N lines
3. Assume spatial index exists and thus ignore cost of building it
4. Worst case queue size arises when:

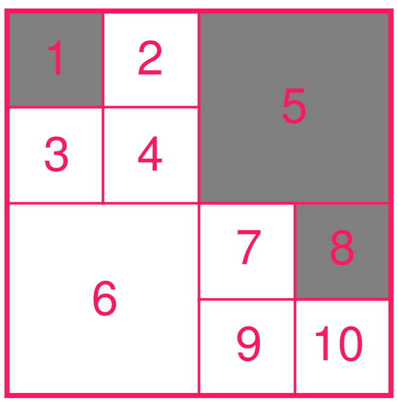


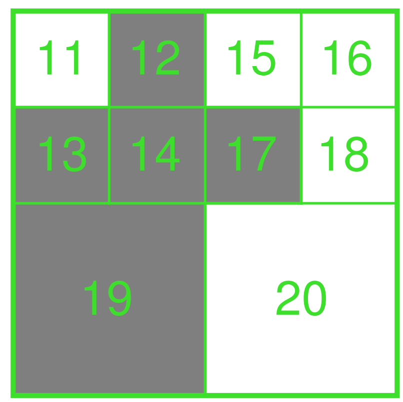
   - all features in queue are at a distance of at least d from q, and
   - all leaf blocks containing the features are at a distance less than d from q
   - implies all features are inserted into the queue before finding the nearest one — i.e., $O(N)$ space
   - if must rank all features, then $O(N \log M)$ execution time
     a. M is maximum queue size
     b. $O(N \log N)$ worst-case time which compares favorably with one-dimensional sorting
   - circular feature configuration is only bad if query object is at the center of the circle
   - highly unlikely for both the features to be in a circle and the query object to be at its center

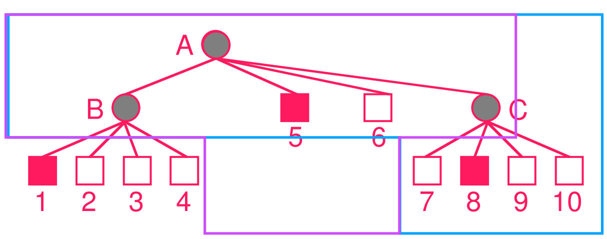
SET OPERATIONS ON QUADTREES

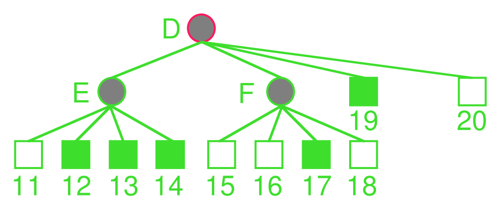
- UNION(S,T) : traverse S and T in tandem

  1. GRAY(S) ● :
     - GRAY(T) ● : recursively process subtrees and merge if all resulting sons are BLACK
     - BLACK(T) ● : result is T
     - WHITE(T) ○ : result is S
  2. BLACK(S) ● : result is S
  3. WHITE(S) ○ : result is T

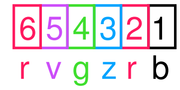






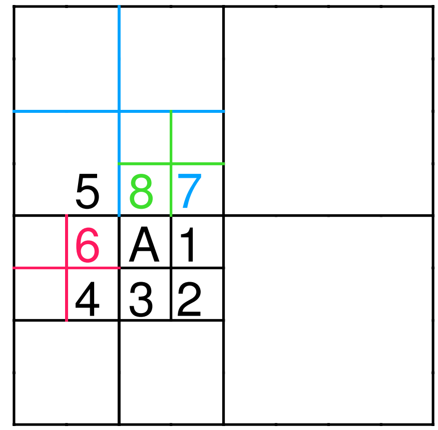


- INTERSECTION: interchange roles of BLACK and WHITE in UNION
- Execution time is bounded by sum of nodes in two input trees but may be less if don't create a new copy as really just the sum of the minimum of the number of nodes at corresponding levels of the two quadtrees
- More efficient than vectors as make use of global data
  1. vectors require a sort for efficiency
  2. region quadtree is already sorted



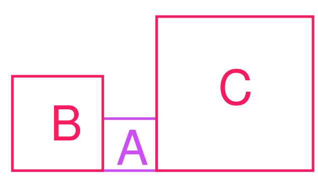
NEIGHBOR FINDING OPERATIONS USING QUADTREES

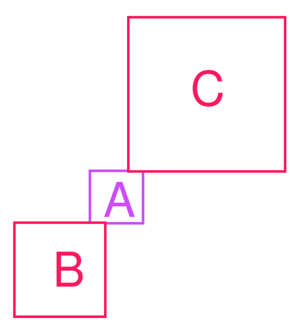
- Many image processing operations involve traversing an image and applying an operation to a pixel and some of its neighboring (i.e., adjacent) pixels
  - For quadtree/octree representations replace pixel/voxel by block
  - Neighbor is defined to be an adjacent block of greater than or equal size



A has ~~5~~ ~~6~~ ~~7~~ 8 neighbors

- Desirable to be able to locate neighbors in a manner that
  1. is position-independent
  2. is size-independent
  3. makes no use of additional links to adjacent nodes (e.g., ropes and nets a la Hunter)
  4. just uses the structure of the tree or configuration of the blocks
- Some block configurations are impossible, thereby simplifying a number of algorithms
  1. impossible for a node A to have two larger neighbors B and C on directly opposite sides or touching corners
  2. partial overlap of two blocks B and C with A is impossible since a quadtree is constructed by recursively splitting blocks into blocks that have side lengths that are powers of 2



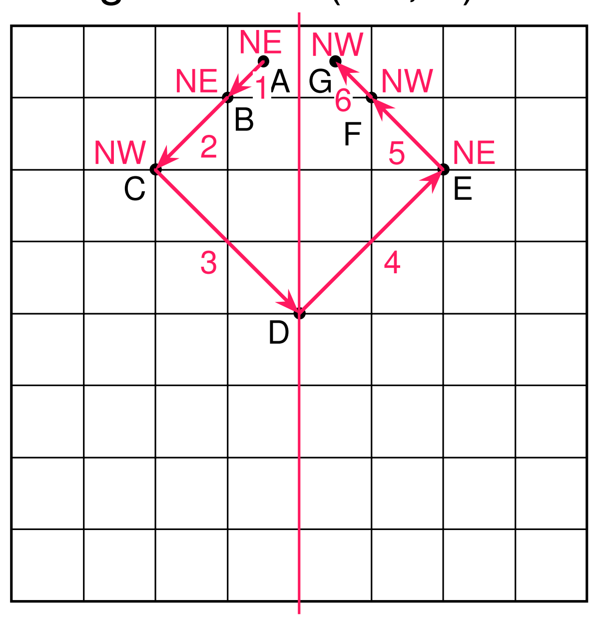


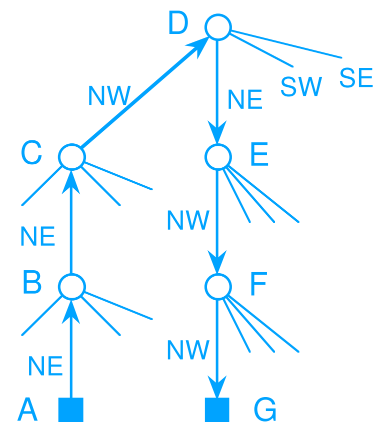
FINDING LATERAL NEIGHBORS OF EQUAL SIZE

Algorithm: based on finding the nearest common ancestor

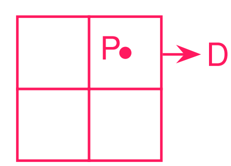
1. Ascend the tree if the node is a son of the same type as the direction of the neighbor (ADJ)
2. Otherwise, the father F is the nearest common ancestor and retrace the path starting at F making mirror image moves about the edge shared by the neighboring blocks

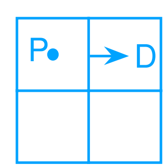
Ex: E neighbor of A (i.e., G)






```
node procedure EQUAL_LATERAL_NEIGHBOR(P,D);
/* Find = size neighbor of P in direction D */
begin
  value pointer node P;
  value direction D;
  return(SON(if ADJ(D,SONTYPE(P)) then
                EQUAL_LATERAL_NEIGHBOR(FATHER(P),D)
             else FATHER(P),
             REFLECT(D,SONTYPE(P))));
end;
```





ADJ(A,B)

A \ B	NW	NE	SW	SE
N	T	T	F	F
E	F	T	F	T
S	F	F	T	T
W	T	F	T	F

REFLECT(A,B)

A \ B	NW	NE	SW	SE
N	SW	SE	NW	NE
E	NE	NW	SE	SW
S	SW	SE	NW	NE
W	NE	NW	SE	SW

ANALYSIS OF NEIGHBOR FINDING

1. Bottom-up random image model where each pixel has an equal probability of being black or white
   - probability of the existence of a 2x2 block at a particular position is 1/8
   - OK for a checkerboard image but inappropriate for maps as it means that there is a very low probability of aggregation
   - problem is that such a model assumes independence
   - in contrast, a pixel's value is typically related to that of its neighbors
2. Top-down random image model where the probability of a node being black or white is p and $1\text{-}2p$ for being gray
   - model does not make provisions for merging
   - uses a branching process model and analysis is in terms of extinct branching processes
3. Use a model based on positions of the blocks in the decomposition
   - a block is equally likely to be at any position and depth in the tree
   - compute an average case based on all the possible positions of a block of size 1x1, 2x2, 4x4, etc.
   - 1 case at depth 0, 4 cases at depth 1, 16 cases at depth 2, etc.
   - this is not a realizable situation but in practice does model the image accurately

ANALYSIS OF FINDING LATERAL NEIGHBORS

25	9	33	1	41	17	49	
26	10	34	2	42	18	50	
27	11	35	3	43	19	51	
28	12	36	4	44	20	52	
29	13	37	5	45	21	53	
30	14	38	6	46	22	54	
31	15	39	7	47	23	55	
32	16	40	8	48	24	56	

$2^3 \cdot (2^3 - 1)$ neighbor pairs of equal sized nodes in direction E

NCA = nearest common ancestor

1–8 have NCA at level 3

9–24 have NCA at level 2

25–56 have NCA at level 1

Theorem: average number of nodes visited by `EQUAL_LATERAL_NEIGHBOR` is ≤ 4

Proof:

- Let node A be at level i (i.e., a $2^i \times 2^i$ block)
- There are $2^{n-i} \cdot (2^{n-i} - 1)$ possible positions for node A such that an equal sized neighbor exists in a given horizontal or vertical direction

  2^{n-i} rows

  $2^{n-i} - 1$ adjacencies per row

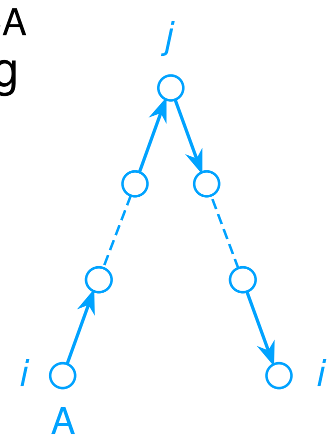
  $2^{n-i} \cdot 2^0$ have NCA at level n

  $2^{n-i} \cdot 2^1$ have NCA at level $n-1$

  …

  $2^{n-i} \cdot 2^{n-i-1}$ have NCA at level $i+1$

- For node A at level $\boxed{i}$, direction D, and the NCA at level $\boxed{j}$, $\boxed{2 \cdot (j-i)}$ nodes are visited in locating an equal-sized neighbor at level $\boxed{i}$



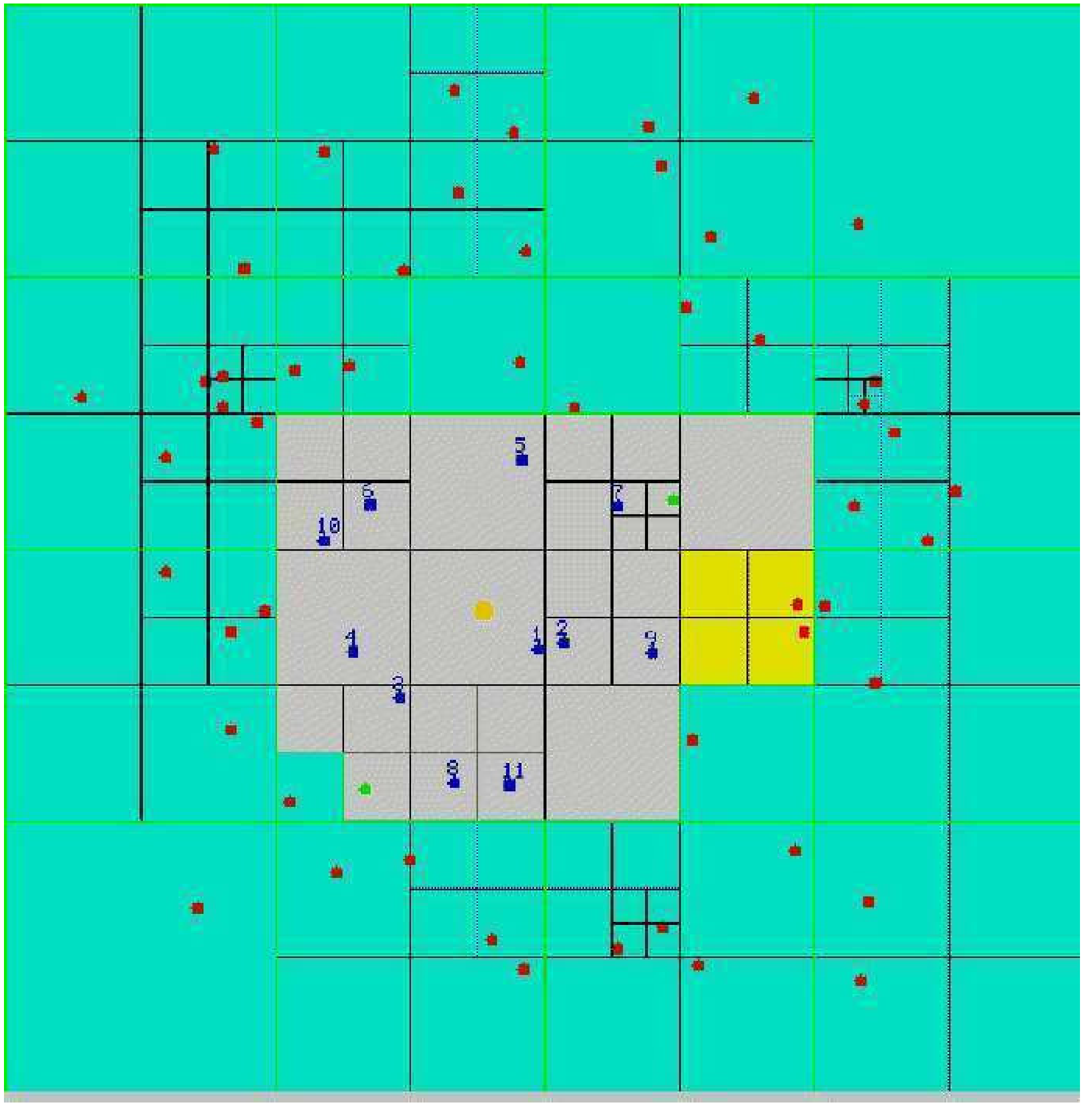
$$\frac{\sum_{i=0}^{n-1} \sum_{j=i+1}^{n} 2^{n-i} \cdot 2^{n-j} \cdot 2 \cdot (j-i)}{\sum_{i=0}^{n-1} 2^{n-i} \cdot (2^{n-i} - 1)}$$

nodes are visited on the average ≤ 4

Outline

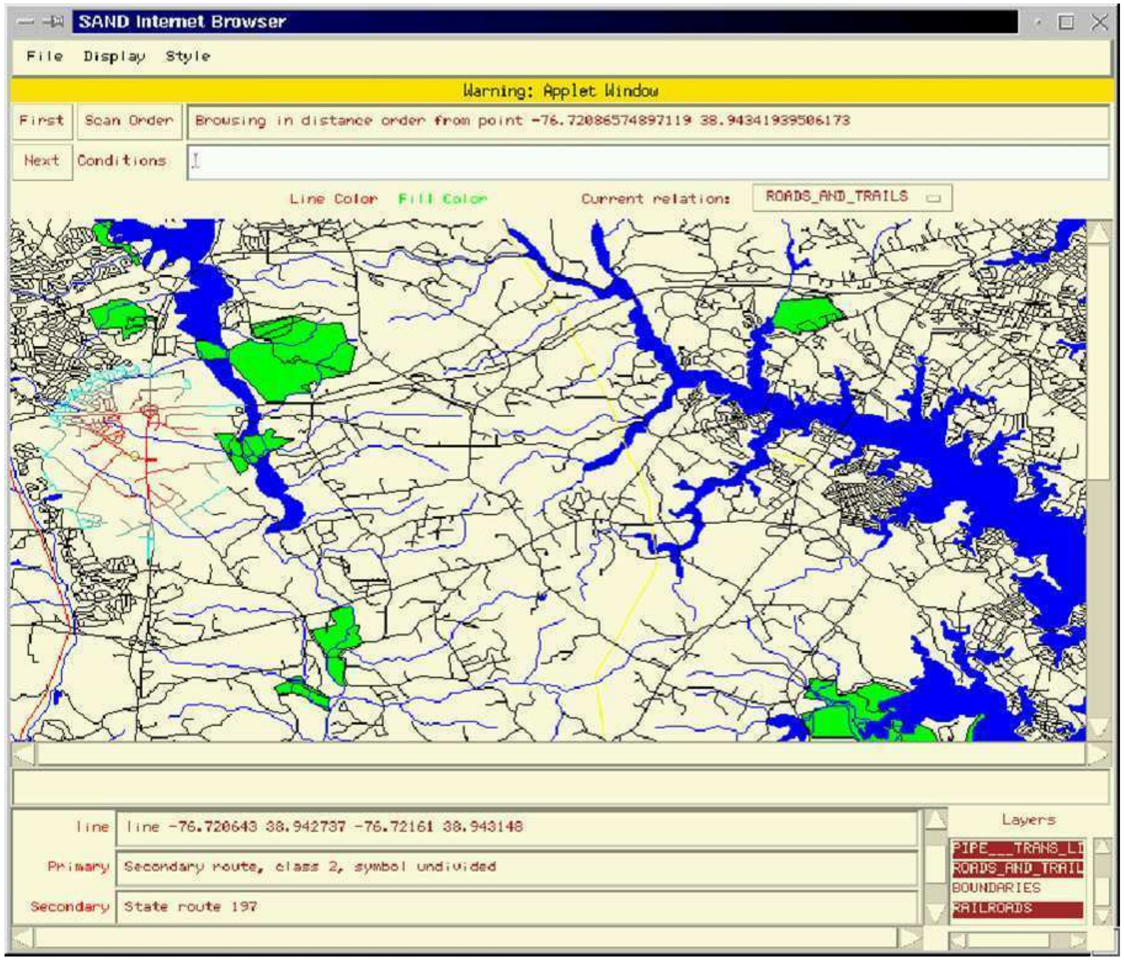
1. Introduction
2. Points
3. Lines
4. Rectangles
5. Regions
6. Surfaces and Volumes
7. Metric Data
8. Operations
9. Example system

VASCO Spatial Applet



http://www.cs.umd.edu/˜hjs/quadtree/index.html

SAND Internet Browser



`http://www.cs.umd.edu/~brabec/sandjava/`

Spatial Data Structures*

Hanan Samet
Computer Science Department and
Institute of Advanced Computer Studies and
Center for Automation Research
University of Maryland
College Park, MD 20742

Abstract

An overview is presented of the use of spatial data structures in spatial databases. The focus is on hierarchical data structures, including a number of variants of quadtrees, which sort the data with respect to the space occupied by it. Such techniques are known as spatial indexing methods. Hierarchical data structures are based on the principle of recursive decomposition. They are attractive because they are compact and depending on the nature of the data they save space as well as time and also facilitate operations such as search. Examples are given of the use of these data structures in the representation of different data types such as regions, points, rectangles, lines, and volumes.

Keywords and phrases: spatial databases, hierarchical spatial data structures, points, lines, rectangles, quadtrees, octrees, R-tree, R^+-tree image processing.

*This work was supported in part by the National Science Foundation under Grant IRI–9017393. Appears in *Modern Database Systems: The Object Model, Interoperability, and Beyond*, W. Kim, ed., Addison Wesley/ACM Press, Reading, MA, 1995, 361-385. Copyright ACM. Permission to reprint in Course Notes of ACM SIGGRAPH 2007 granted.

1 Introduction

Spatial data consists of spatial objects made up of points, lines, regions, rectangles, surfaces, volumes, and even data of higher dimension which includes time. Examples of spatial data include cities, rivers, roads, counties, states, crop coverages, mountain ranges, parts in a CAD system, etc. Examples of spatial properties include the extent of a given river, or the boundary of a given county, etc. Often it is also desirable to attach non-spatial attribute information such as elevation heights, city names, etc. to the spatial data. Spatial databases facilitate the storage and efficient processing of spatial and non-spatial information ideally without favoring one over the other. Such databases are finding increasing use in applications in environmental monitoring, space, urban planning, resource management, and geographic information systems (GIS) [Buchmann et al. 1989; Günther and Schek 1991].

A common way to deal with spatial data is to store it explicitly by parametrizing it and thereby obtaining a reduction to a point in a possibly higher dimensional space. This is usually quite easy to do in a conventional database management system since the system is just a collection of records, where each record has many fields. In particular, we simply add a field (or several fields) to the record that deals with the desired item of spatial information. This approach is fine if we just want to perform a simple retrieval of the data.

However, if our query involves the space occupied by the data (and hence other records by virtue of their proximity), then the situation is not so straightforward. In such a case we need to be able to retrieve records based on some spatial properties which are not stored explicitly in the database. For example, in a roads database, we may not wish to force the user to specify explicitly which roads intersect which other roads or regions. The problem is that the potential volume of such information may be very large and the cost of preprocessing it high, while the cost of computing it on the fly may be quite reasonable, especially if the spatial data is stored in an appropriate manner. Thus we prefer to store the data implicitly so that a wide class of spatial queries can be handled. In particular, we need not know the types of queries a priori.

Being able to respond to spatial queries in a flexible manner places a premium on the appropriate representation of the spatial data. In order to be able to deal with proximity queries the data must be sorted. Of course, all database management systems sort the data. The issue is which keys do they sort on. In the case of spatial data, the sort should be based on all of the spatial keys, which means that, unlike conventional database management systems, the sort is based on the space occupied by the data. Such techniques are known as *spatial indexing* methods.

One approach to the representation of spatial data is to separate it structurally from the nonspatial data while maintaining appropriate links between the two [Aref and Samet 1991a]. This leads to a much higher bandwidth for the retrieval of the spatial data. In such a case, the spatial operations are performed directly on the spatial data structures. This provides the freedom to choose a more appropriate spatial structure than the imposed non-spatial structure (e.g., a relational database). In such a case, a spatial processor can be used that is specifically designed for efficiently dealing with the part of the queries that involve proximity relations and search, and a relational database management system for the part of the queries that involve non-spatial data. Its proper functioning depends on the existence of a query optimizer to determine the appropriate processor for each part of the query [Aref and Samet 1991b].

As an example of the type of query to be posed to a spatial database system, consider a request to "find the names of the roads that pass through the University of Maryland region". This requires the extraction of the region locations of all the database records whose "region name" field has the value "University of Maryland" and build a map A. Next, map A is intersected with the road map B to yield a new map C with the selected roads. Now, create a new relation having just one attribute which is the relevant road names of the roads in map C. Of course, there are other approaches to answering the above query. Their efficiency depends on the nature of the data and its volume.

In the rest of this review we concentrate on the data structures used by the spatial processor. In particular, we focus on hierarchical data structures. They are based on the principle of recursive decomposition (similar to *divide and conquer* methods). The term *quadtree* is often used to describe many elements of this class of data structures. We concentrate primarily on region, point, rectangle, and line data. For a more extensive treatment of this subject, see [Samet 1990a; Samet 1990b].

Our presentation is organized as follows. Section 2 describes a number of different methods of indexing spatial data. Section 3 focusses on region data and also briefly reviews the historical background of the origins of hierarchical spatial data structures such as the quadtree. Sections 4, 5, and 6 describe hierarchical representations for point, rectangle, and line data, respectively, as well as give examples of their utility. Section 7 contains concluding remarks in the context of a geographic information system that makes use of these concepts.

2 Spatial Indexing

Each record in a database management system can be conceptualized as a point in a multi-dimensional space. This analogy is used by many researchers (e.g., [Hinrichs and Nievergelt 1983; Jagadish 1990]) to deal with spatial data as well by use of suitable transformations that map the spatial object (henceforth we just use the term *object*) into a point (termed a *representative point*) in either the same (e.g., [Jagadish 1990]), lower (e.g., [Orenstein and Merrett 1984]), or higher (e.g., [Hinrichs and Nievergelt 1983]) dimensional spaces. This analogy is not always appropriate for spatial data. One problem is that the dimensionality of the representative point may be too high [Orenstein 1989]. One solution is to approximate the spatial object by reducing the dimensionality of the representative point. Another more serious problem is that use of these transformations does not preserve proximity.

To see the drawback of just mapping spatial data into points in another space, consider the representation of a database of line segments. We use the term *polygonal map* to refer to such a line segment database, consisting of vertices and edges, regardless of whether or not the line segments are connected to each other. Such a database can arise in a network of roads, power lines, rail lines, etc. Using a representative point (e.g., [Jagadish 1990]), each line segment can be represented by its endpoints[1]. This means that each line segment is represented by a tuple of four items (i.e., a pair of x coordinate values and a pair of y coordinate values). Thus, in effect, we have constructed a mapping from a two-dimensional

[1] Of course, there are other mappings but they have similar drawbacks. We shall use this example in the rest of this section.

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).

This mapping is fine for storage purposes and for queries that only involve the points that comprise the line segments (including their endpoints). For example, finding all the line segments that intersect a given point or set of points or a given line segment. However, it is not good for queries that involve points or sets of points that are not part of the line segments as they are not transformed to the higher dimensional space by the mapping. Answering such a query involves performing a search in the space from which the lines are drawn rather than in the space into which they are mapped.

As a more concrete example of the shortcoming of the mapping approach suppose that we want to detect if two lines are near each other, or, alternatively, to find the nearest line to a given point or line. This is difficult to do in the four-dimensional space since proximity in the two-dimensional space from which the lines are drawn is not necessarily preserved in the four-dimensional space into which the lines are mapped. 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.

Thus we need different representations for spatial data. One way to overcome these problems is to use data structures that are based on spatial occupancy. Spatial occupancy methods decompose the space from which the data is drawn (e.g., the two-dimensional space containing the lines) into regions called *buckets*. They are also commonly known as *bucketing methods*. Traditionally, bucketing methods such as the grid file [Nievergelt et al. 1984], BANG file [Freeston 1987], LSD trees [Henrich et al. 1989], buddy trees [Seeger and Kriegel 1990], etc. have always been applied to the transformed data (i.e., the representative points). In contrast, we are applying the bucketing methods to the space from which the data is drawn (i.e., two-dimensions in the case of a collection of line segments).

There are four principal approaches to decomposing the space from which the data is drawn. One approach buckets the data based on the concept of a minimum bounding (or enclosing) rectangle. In this case, objects are grouped (hopefully by proximity) into hierarchies, and then stored in another structure such as a B-tree [Comer 1979]. The R-tree (e.g., [Beckmann et al. 1990; Guttman 1984]) is an example of this approach.

The R-tree and its variants are designed to organize a collection of arbitrary spatial objects (most notably two-dimensional rectangles) by representing them as d-dimensional rectangles. Each node in the tree corresponds to the smallest d-dimensional rectangle that encloses its son nodes. Leaf nodes contain pointers to the actual objects in the database, instead of sons. The objects are represented by the smallest aligned rectangle containing them.

Often the nodes correspond to disk pages and, thus, the parameters defining the tree are chosen so that a small number of nodes is visited during a spatial query. Note that the bounding rectangles corresponding to different nodes may overlap. Also, an object may be spatially contained in several nodes, yet it is only associated with one node. This means that a spatial query may often require several nodes to be visited before ascertaining the presence or absence of a particular object.

The basic rules for the formation of an R-tree are very similar to those for a B-tree. All leaf nodes appear at the same level. Each entry in a leaf node is a 2-tuple of the form

(R,O) such that R is the smallest rectangle that spatially contains object O. Each entry in a non-leaf node is a 2-tuple of the form (R,P) such that R is the smallest rectangle that spatially contains the rectangles in the child node pointed at by P. An R-tree of order (m,M) means that each node in the tree, with the exception of the root, contains between $m \leq \lceil M/2 \rceil$ and M entries. The root node has at least two entries unless it is a leaf node.

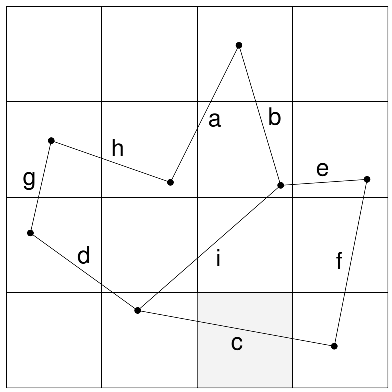
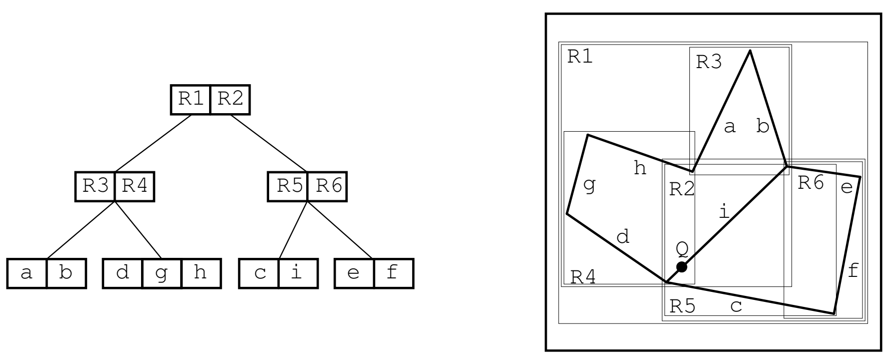


Figure 1: Example collection of line segments embedded in a 4×4 grid.

For example, consider the collection of line segments given in Figure 1 shown embedded in a 4×4 grid. Let $M = 3$ and $m = 2$. One possible R-tree for this collection is given in Figure 2a. Figure 2b shows the spatial extent of the bounding rectangles of the nodes in Figure 2a, with broken lines denoting the rectangles corresponding to the subtrees rooted at the non-leaf nodes. Note that the R-tree is not unique. Its structure depends heavily on the order in which the individual line segments were inserted into (and possibly deleted from) the tree.



(a) (b)

Figure 2: (a) R-tree for the collection of line segments in Figure 1, and (b) the spatial extents of the bounding rectangles.

The drawback of these methods is that they do 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 Figure2 is associated with rectangle `R5`, yet it passes through `R1`, `R2`, `R4`, and `R5`). In the worst case, this means that when we wish to determine which object is associated with a particular point (e.g., the containing rectangle in a rectangle database, or an intersecting line in a line segment database) in the two-dimensional space from which the objects are drawn, we may have to search the entire database.

For example, suppose we wish to determine the identity of the line segment in the collection of line segments given in Figure 2 that passes through point `Q`. Since `Q` can be in either of `R1` or `R2`, we must search both of their subtrees. Searching `R1` first, we find that `Q` could only be contained in `R4`. Searching `R4` does not lead to the line segment that contains `Q` even though `Q` is in a portion of bounding rectangle `R4` that is in `R1`. Thus, we must search `R2` and we find that `Q` can only be contained in `R5`. Searching `R5` results in locating `i`, the desired line segment.

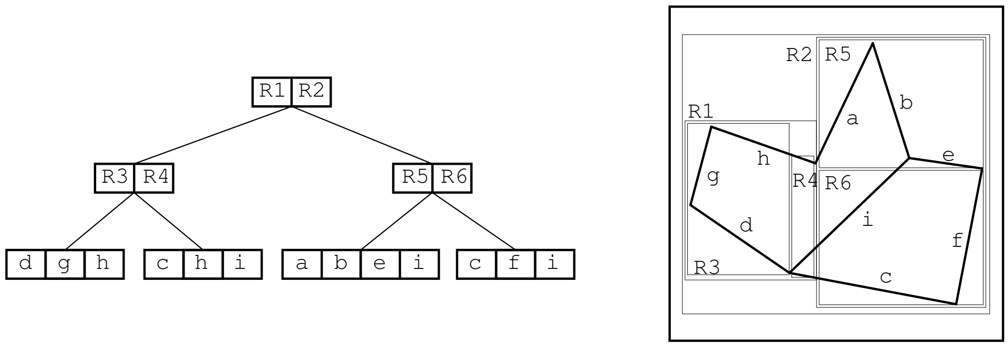
The other approaches are based on a decomposition of space into disjoint cells, which are mapped into buckets. Their common property is that the objects are decomposed into disjoint subobjects such that each of the subobjects is associated with a different cell. They differ in the degree of regularity imposed by their underlying decomposition rules and by the way in which the cells are aggregated. 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 these databases. A related drawback is that when we wish to determine all the objects that occur in a particular region we often retrieve many of the objects more than once. This is particularly problematic 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 [Aref and Samet 1992] for a discussion of how to deal with this problem in a database of line segments).

The first method based on disjointness partitions the objects into arbitrary disjoint subobjects and then groups the subobjects in another structure such as a B-tree. The partition and the subsequent groupings are such that the bounding rectangles are disjoint at each level of the structure. The R$^+$-tree [Sellis et al. 1987] and the cell tree [Günther 1988] are examples of this approach. They differ in the data with which they deal. The R$^+$-tree deals with collections of objects that are bounded by rectangles, while the cell tree deals with convex polyhedra.

The R$^+$-tree is an extension of the k-d-B-tree [Robinson 1981]. The R$^+$-tree is motivated by a desire to avoid overlap among the bounding rectangles. Each object is associated with all the bounding 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 [2]. 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 3 is an example of one possible R$^+$-tree for the collection of line segments in Figure 1. 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 segments `c` and `h` appear in two different nodes, while line segment `i` appears in three different nodes. Of course, other variants are possible since the R$^+$-tree is

[2]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 where many line segments intersect at a point).



(a) (b)

Figure 3: (a) R$^+$-tree for the collection of line segments in Figure 1 and (b) the spatial extents of the bounding rectangles.

not unique.

Methods such as the R$^+$-tree and the cell tree (as well as the R*-tree [Beckmann et al. 1990]) 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). In contrast, the remaining two methods, while also yielding a disjoint decomposition, 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 [Franklin 1984]) or adapt the decomposition to the distribution of the data (e.g., a quadtree-based approach such as [Samet and Webber 1985]). In the former case, all the blocks are of the same size (e.g., the 4×4 grid in Figure 1). In the latter case, the widths of the blocks are restricted to be powers of two, and their positions are also restricted.

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 n objects.

All of the spatial occupancy methods discussed above are characterized as employing spatial indexing because with each block the only information that is stored is whether or not the block is occupied by the object or part of the object. This information is usually in the form of a pointer to a descriptor of the object. For example, in the case of a collection of line segments in the uniform grid of Figure 1, the shaded block only records the fact that a line segment crosses it or passes through it. The part of the line segment that passes through the block (or terminates within it) is termed a *q-edge*. Each q-edge in the block is represented by a pointer to a record containing the endpoints of the line segment of which the q-edge is a part [Nelson and Samet 1986]. This pointer is really nothing more than a spatial index and hence the use of this term to characterize this approach. Thus no information is associated with the shaded block as to what part of the line (i.e., q-edge) crosses it. This information can be obtained by clipping [Foley et al. 1990] the original line segment to the block. This is important for often the precision necessary to compute these intersection points is not available.

3 Region Data

A region can be represented either by its interior or by its boundary. In this section we focus on the representations of regions by their interior, while the use of a boundary is discussed in Section 6 in the context of collections of line segments as found, for example, in polygonal maps. The most common region representation is the image array. In this case, we have a collection of picture elements (termed *pixels*). Since the number of elements in the array can be quite large, there is interest in reducing its size by aggregating similar (i.e., homogeneous or equal-valued) pixels. There are two basic approaches. The first approach breaks up the array into $1 \times m$ blocks [Rutovitz 1968]. This is a row representation and is known as a *runlength code*. A more general approach treats the region as a union of maximal square blocks (or blocks of any other desired shape) that may possibly overlap. Usually the blocks are specified by their centers and radii. This representation is called the *medial axis transformation (MAT)* [Blum 1967].

When the maximal blocks are required to be disjoint, to have standard sizes (squares whose sides are powers of two), and to be at standard locations (as a result of a halving process in both the x and y directions), the result is known as a *region quadtree* [Klinger 1971]. It is based on the successive subdivision of the image array into four equal-size quadrants. If the array does not consist entirely of 1s or entirely of 0s (i.e., the region does not cover the entire array), it is then subdivided into quadrants, subquadrants, etc., until blocks are obtained (possibly 1×1 blocks) that consist entirely of 1s or entirely of 0s. Thus, the region quadtree can be characterized as a variable resolution data structure.

As an example of the region quadtree, consider the region shown in Figure 4a which is represented by the $2^3 \times 2^3$ binary array in Figure 4b. Observe that the 1s correspond to pixels that are in the region and the 0s correspond to pixels that are outside the region. The resulting blocks for the array of Figure 4b are shown in Figure 4c. This process is represented by a tree of degree 4.

In the tree representation, the root node corresponds to the entire array. Each son of a node represents a quadrant (labeled in order NW, NE, SW, SE of the region represented by that node. The leaf nodes of the tree correspond to those blocks for which no further subdivision is necessary. A leaf node is said to be BLACK or WHITE, depending on whether its corresponding block is entirely inside or entirely outside of the represented region. All non-leaf nodes are said to be GRAY. The quadtree representation for Figure 4c is shown in Figure 4d. Of course, quadtrees can also be used to represent non-binary images. In this case, the same merging criteria is applied to each color. For example, in the case of a landuse map, simply merge all wheat growing regions, and likewise for corn, rice, etc. [Samet et al. 1984].

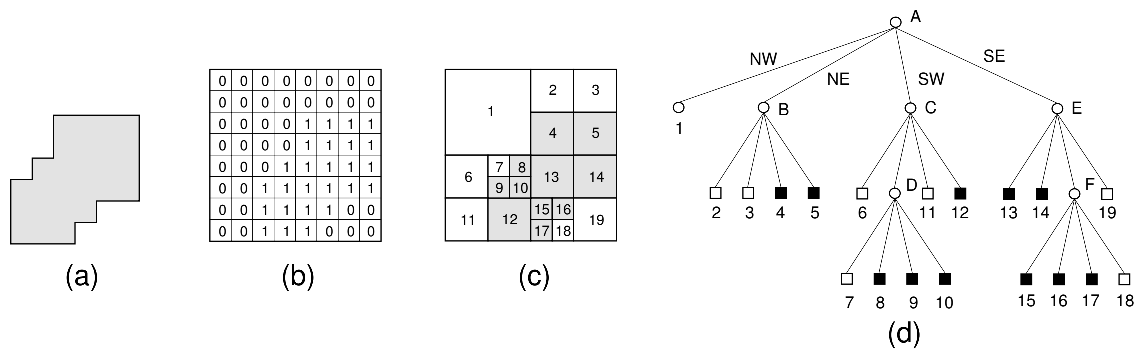


Figure 4: (a) Sample region, (b) its binary array representation, (c) its maximal blocks with the blocks in the region being shaded, and (d) the corresponding quadtree.

The term *quadtree* is often used in a more general senses to describe a class of hierarchical data structures whose common property is that they are based on the principle of recursive decomposition of space. They can be differentiated on the following bases:

1. the type of data that they are used to represent,
2. the principle guiding the decomposition process, and
3. the resolution (variable or not).

Currently, they are used for points, rectangles, regions, curves, surfaces, and volumes (see the remaining sections for further details on the adaptation of the quadtree to them). The decomposition may be into equal parts on each level (termed a *regular decomposition*), or it may be governed by the input. The resolution of the decomposition (i.e., the number of times that the decomposition process is applied) may be fixed beforehand or it may be governed by properties of the input data.

Unfortunately, the term *quadtree* has taken on more than one meaning. The region quadtree, as shown above, is a partition of space into a set of squares whose sides are all a power of two long. A similar partition of space into rectangular quadrants is termed a *point quadtree* [Finkel and Bentley 1974]. It is an adaptation of the binary search tree to two dimensions (which can be easily extended to an arbitrary number of dimensions). It is primarily used to represent multidimensional point data where the rectangular regions need not be square. The quadtree is also often confused with the pyramid [Tanimoto and Pavlidis 1975]. The pyramid is a multiresolution representation which is an exponentially tapering stack of arrays, each one-quarter the size of the previous array. In contrast, the region quadtree is a variable resolution data structure.

The distinction between a quadtree and a pyramid is important in the domain of spatial databases, and can be easily seen by considering the types of spatial queries. There are two principal types [Aref and Samet 1990]. The first is location-based. In this case, we are searching for the nature of the feature associated with a particular location or in its proximity. For example, “what is the feature at location X?”, “what is the nearest city to location X?”, or “what is the nearest road to location X?” The second is feature-based. In this case, we are probing for the presence or absence of a feature, as well as seeking its actual location. For example, "does wheat grow anywhere in California?", "what crops grow in California?", or "where is wheat grown in California?"

Location-based queries are easy to answer with a quadtree representation as they involve descending the tree until finding the object. If a nearest neighbor is desired, then the search is continued in the neighborhood of the node containing the object. This search can also be achieved by unwinding the process used to access the node containing the object. On the other hand, feature-based queries are more difficult. The problem is that there is no indexing by features. The indexing is only based on spatial occupancy. The goal is to process the query without examining every location in space. The pyramid is useful for such queries since the nodes that are not at the maximum level of resolution (i.e., at the bottom level) contain summary information. Thus we could view these nodes as feature vectors which indicate whether or not a feature is present at a higher level of resolution. Therefore, by examining the root of the pyramid (i.e., the node that represents the entire image) we can quickly tell if a feature is present without having to examine every location.

For example, consider the block decomposition of the non-binary image in Figure 5a. Its truncated pyramid is given in Figure 5b. The values of a nonleaf node p in the truncated pyramid indicate if the feature is present in the subtrees of p. In the interest of saving space, the pyramid is not shown in its entirety here.

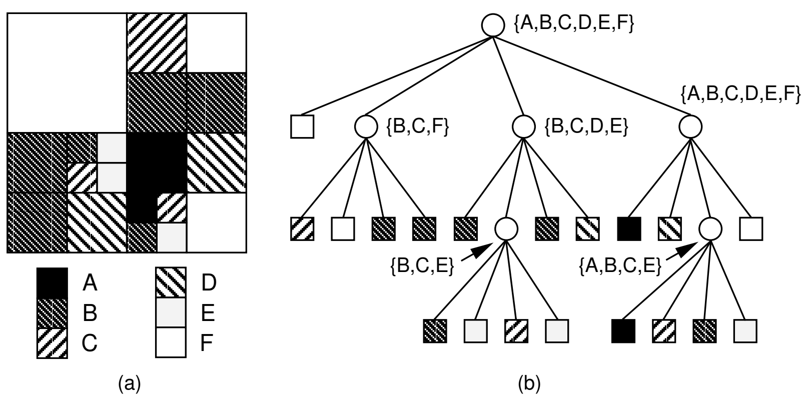


Figure 5: (a) Sample non-binary image, and (b) its corresponding truncated pyramid.

Quadtree-like data structures can also be used to represent images in three dimensions and higher. The octree [Hunter 1978; Meagher 1982] data structure is the three-dimensional analog of the 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 6a is an example of a simple three-dimensional object whose raster octree block decomposition is given in Figure 6b and whose tree representation is given in Figure 6c.

The quadtree is particularly useful for performing set operations as they form the basis of most complicated queries. For example, to "find the names of the roads that pass through the University of Maryland region," we will need to intersect a region map with a line map. For a binary image, set-theoretic operations such as union and intersection are quite simple to implement [Hunter and Steiglitz 1979].

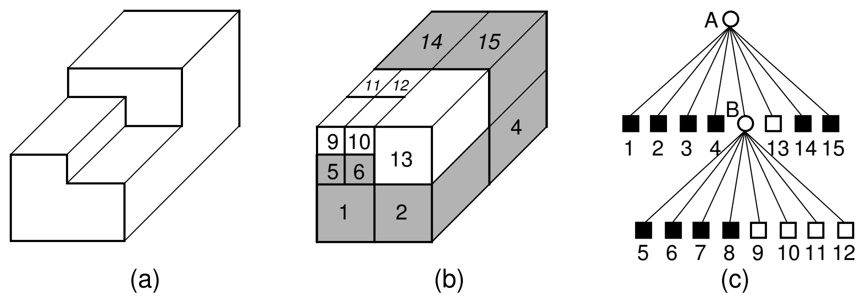


Figure 6: (a) Example three-dimensional object; (b) its octree block decomposition; and (c) its tree representation.

In particular, the intersection of two quadtrees yields a BLACK node only when the corresponding regions in both quadtrees are BLACK. This operation is performed by simultaneously traversing three quadtrees. The first two trees correspond to the trees being intersected and the third tree represents the result of the operation. If any of the input nodes are WHITE, then the result is WHITE. When corresponding nodes in the input trees are GRAY, then their sons are recursively processed and a check is made for the mergibility of WHITE leaf nodes. The worst-case execution time of this algorithm is proportional to the sum of the number of nodes in the two input quadtrees, although it is possible for the intersection algorithm to visit fewer nodes than the sum of the nodes in the two input quadtrees.

Performing the set operations on an image represented by a region quadtree is much more efficient than when the image is represented by a boundary representation (e.g., vectors) as it makes use of global data. In particular, to be efficient, a vector-based solution must sort the boundaries of the region with respect to the space which they occupy, while in the case of a region quadtree, the regions are already sorted.

One of the motivations for the development of hierarchical data structures such as the quadtree is a desire to save space. The original formulation of the quadtree encodes it as a tree structure that uses pointers. This requires additional overhead to encode the internal nodes of the tree. In order to further reduce the space requirements, two other approaches have been proposed. The first treats the image as a collection of leaf nodes where each leaf is encoded by a pair of numbers. The first is a base 4 number termed a *locational code*, corresponding to a sequence of directional codes that locate the leaf along a path from the root of the quadtree (e.g., [Gargantini 1982]). It is analogous to taking the binary representation of the x and y coordinates of a designated pixel in the block (e.g., the one at the lower left corner) and interleaving them (i.e., alternating the bits for each coordinate). The second number indicates the depth at which the leaf node is found (or alternatively its size).

The second, termed a *DF-expression*, represents the image in the form of a traversal of the nodes of its quadtree [Kawaguchi and Endo 1980]. It is very compact as each node type can be encoded with two bits. However, it is not easy to use when random access to nodes is desired. For a static collection of nodes, an efficient implementation of the pointer-based representation is often more economical spacewise than a locational code representation [Samet and Webber 1989]. This is especially true for images of higher dimension.

Nevertheless, depending on the particular implementation of the quadtree we may not necessarily save space (e.g., in many cases a binary array representation may still be more economical than a quadtree). However, the effects of the underlying hierarchical aggregation on the execution time of the algorithms are more important. Most quadtree algorithms are simply preorder traversals 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* [Hunter 1978] which states that the number of nodes in a quadtree region representation is $O(p+q)$ for a $2^q \times 2^q$ image with perimeter p measured in pixel-widths. 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 holds for three-dimensional data [Meagher 1980] 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 Quadtree Complexity Theorem also directly impacts the analysis of the execution time of algorithms. In particular, most algorithms that execute on a quadtree representation of an image instead of an array representation have an execution time that is proportional to the number of blocks in the image rather than the number of pixels. In its most general case, this means that the application of a quadtree algorithm to a problem in d-dimensional space executes in time proportional to the analogous array-based algorithm in the $(d-1)$-dimensional space of the surface of the original d-dimensional image. Therefore, quadtrees act like dimension-reducing devices.

4 Point Data

Multidimensional point data can be represented in a variety of ways. The representation ultimately chosen for a specific task is influenced by the type of operations to be performed on the data. Our focus is on dynamic files (i.e., the amount of data can grow and shrink at will) and on applications involving search. In Section 3 we briefly mentioned the point quadtree. In higher dimensions (i.e., greater than 3) it is preferable to use the k-d tree [Bentley 1975] as every node has degree 2 since the partitions cycle through the different attributes.

There are many different representations for point data. Most of them are some variants of the bucket methods discussed in Section 2. These include the grid file and EXCELL which are described in Section 6. For more details, see [Samet 1990b]. In this section we present the PR quadtree (P for point and R for region) [Orenstein 1982; Samet 1990b] as it is based on a regular decomposition. It is an adaptation of the region quadtree to point data which associates data points (that need not be discrete) with quadrants. The PR quadtree is organized in the same way as the region quadtree. The difference is that leaf nodes are either empty (i.e., WHITE) or contain a data point (i.e., BLACK) and its coordinate values. A quadrant contains at most one data point. For example, Figure 7 is a PR quadtree corresponding to some point data.

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

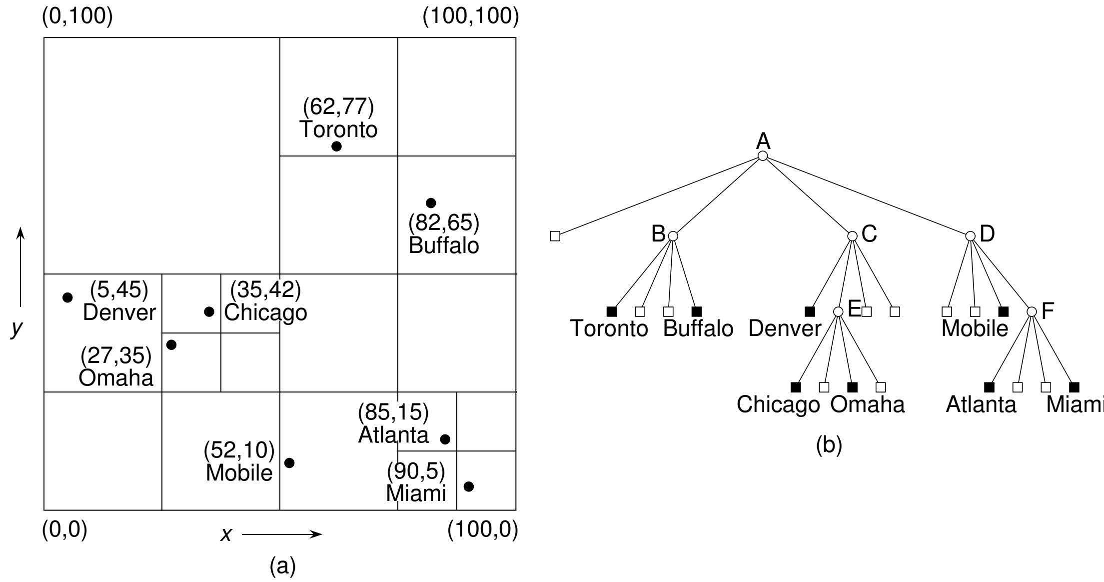


Figure 7: A PR quadtree.

points are very close, then the decomposition can be very deep. This can be overcome by viewing the blocks or nodes as buckets with capacity c and only decomposing a block when it contains more than c points. Of course, bucketing methods such as the R-tree and the R$^+$-tree can also be used.

PR quadtrees, as well as other quadtree-like representations for point data, are especially attractive in applications that involve search. A typical query is one that requests the determination of all records within a specified distance of a given record - e.g., all cities within 100 miles of Washington, DC. The efficiency of the PR quadtree lies in its role as a pruning device on the amount of search that is required. Thus many records will not need to be examined. For example, suppose that in the hypothetical database of Figure 7 we wish to find all cities within 8 units of a data point with coordinates (84,10). In such a case, there is no need to search the NW, NE, and SW quadrants of the root (i.e., (50,50)). Thus we can restrict our search to the SE quadrant of the tree rooted at root. Similarly, there is no need to search the NW, NE, and SW quadrants of the tree rooted at the SE quadrant (i.e., (75,25)). Note that the search ranges are usually orthogonally defined regions such as rectangles, boxes, etc. Other shapes are also feasible as the above example demonstrated (i.e., a circle).

5 Rectangle Data

The rectangle data type lies somewhere between the point and region data types. 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 can be 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.

The representation that is used depends heavily on the problem environment. If the environment is static, then frequently the solutions are based on the use of the plane-sweep paradigm [Preparata and Shamos 1985], which usually yields optimal solutions in time and space. However, the addition of a single object to the database forces the re-execution of the algorithm on the entire database. We are primarily interested in dynamic problem environments. 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 2 reduces each rectangle to a point in a higher dimensional space, and then treats the problem as if we have a collection of points [Hinrichs and Nievergelt 1983]. Each rectangle is a Cartesian product of two one-dimensional intervals where 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 [Nievergelt et al. 1984].

The grid file is a two-level grid for storing multidimensional points. It uses a grid directory (a two-dimensional array of grid blocks for two-dimensional point data) on disk indicating the address of the bucket (i.e., page) that contains the data associated with the grid block. A set of linear scales (actually a pair of one-dimensional arrays in the case of two-dimensional data) are kept in core. The linear scales access the grid block in the grid directory (on disk) that is associated with a particular point. The grid file guarantees access to any record with two disk operations – that is, one for each level of the grid. The first access is to the grid block, while the second access is to the grid bucket. The linear scales are necessary because the grid lines in the grid directory can be in arbitrary positions.

In contrast, EXCELL [Tamminen 1981] also guarantees access to any record with two disk operations but makes use of regular decomposition. This means that the linear scales are not necessary. However, a grid partition results in doubling the size of the grid directory.

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 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 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 adapt the R-tree and the R$^+$-tree (discussed in Section 2) to store rectangle data (i.e., in this case the objects are rectangles instead of line segments as in Figures 2 and 3). The third is a quadtree-based approach and uses the MX-CIF quadtree [Kedem 1982].

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 [Kedem 1982]. For example, Figure 8 is the MX-CIF quadtree for a collection of rectangles. Note that rectangle F occupies an entire block and hence it is associated with the block's father. Also rectangles can be associated with both terminal and non-terminal nodes.

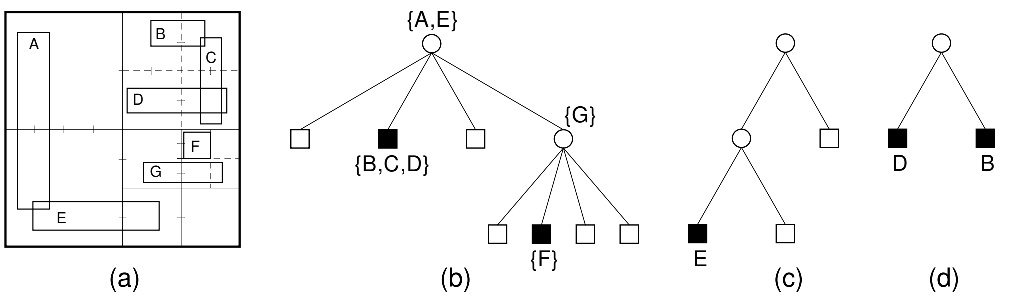


Figure 8: (a) Collection of rectangles and the block decomposition induced by the MX-CIF quadtree; (b) the tree representation of (a); the binary trees for the y axes passing through the root of the tree in (b), and (d) the NE son of the root of the tree in (b).

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 [Kedem 1982]. 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 8c and Figure 8d illustrate the binary trees associated with the y axes passing through the root and the NE son of the root of the MX-CIF quadtree of Figure 8c, respectively. Interestingly, the MX-CIF quadtree is a two-dimensional analog of the interval tree [Edelsbrunner 1980], which is a data structure that is used to support optimal solutions based on the plane-sweep paradigm to some rectangle problems.

6 Line Data

Section 3 was devoted to the region quadtree, an approach to region representation that is based on a description of the region's interior. In this section, we focus on a representation that specifies the boundaries of regions. The simplest representation is the polygon consisting 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 [Freeman 1974] which is an approximation of a polygon's boundary by use of a sequence of unit vectors in the four principal directions. 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. Nevertheless, the vector representation is used in many commercial systems (e.g., ARC/INFO [Peuquet and Marble 1990]) on account of its compactness.

In this section we concentrate on the use of bucketing methods. There are a number of choices (see [Hoel and Samet 1992] for an empirical comparison). The first two are the R-tree and the R^+-tree which have already been explained in Section 2. The third uses regular decomposition to adaptively sort the line segments into buckets of varying size. There is a one-to-one correspondence between buckets and blocks in the two-dimensional space from which the line segments are drawn. There are a number of approaches to this problem [Samet 1990b]. They differ by being either vertex-based or edge-based. Their implementations make use of the same basic data structure. All are built by applying the same principle of repeatedly breaking up the collection of vertices and edges (making up the polygonal map) into groups of four blocks of equal size (termed *brothers*) until obtaining a subset that is sufficiently simple so that it can be organized by some other data structure. This is achieved by successively weakening the definition of what constitutes a legal block, thereby enabling more information to be stored in each bucket.

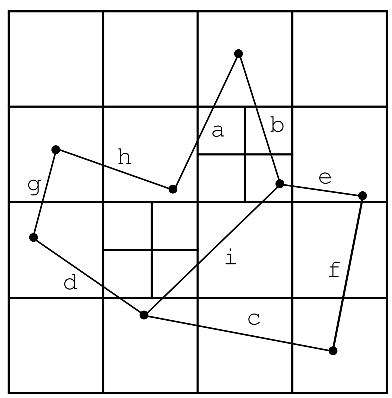


Figure 9: PM_1 quadtree for the collection of line segments of Figure 1.

The PM quadtree family [Samet and Webber 1985] is vertex-based. We illustrate the PM_1 quadtree. It is based on a decomposition rule stipulating 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 9). A similar representation has been devised for three-dimensional images (e.g., [Ayala et al. 1985]). 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 conventional octree.

The PMR quadtree [Nelson and Samet 1986; Nelson and Samet 1987] is an edge-based variant of the PM quadtree (see also edge-EXCELL [Tamminen 1981]). It makes use of a probabilistic splitting rule. A block is permitted to contain a variable number of line segments. The PMR quadtree is constructed by inserting them one-by-one into an initially empty structure consisting of one block. Each line segment is inserted into all of the blocks that it intersects or occupies in its entirety. During this process, the occupancy of each affected block is checked to see if the insertion causes it to exceed a predetermined *splitting threshold*. If the splitting threshold is exceeded, then the block is split *once*, and only once, into four blocks of equal size. The rationale is to avoid splitting a node many times when there are a few very close lines in a block. In this manner, we avoid pathologically bad cases. For more details, see [Nelson and Samet 1986].

A line segment is deleted from a PMR quadtree by removing it from all the blocks that it intersects or occupies in its entirety. During this process, the occupancy of the block and its siblings (the ones that were created when its predecessor was split) is checked to see if the deletion causes the total number of line segments in them to be less than the splitting threshold. If the splitting threshold exceeds the occupancy of the block and its siblings, then they are merged and the merging process is recursively reapplied to the resulting block and its siblings. Notice the asymmetry between splitting and merging rules.

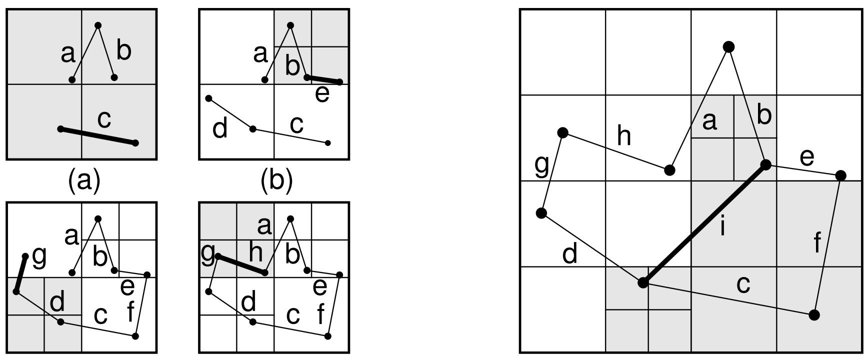


Figure 10: PMR quadtree for the collection of line segments of Figure 1. (a) – (e) illustrate snapshots of the construction process with the final PMR quadtree given in (e).

Figure 10e is an example of a PMR quadtree corresponding to a set of 9 edges labeled *a*–*i* inserted in increasing order. Observe that the shape of the PMR quadtree for a given polygonal map is not unique; instead it depends on the order in which the lines are inserted into it. In contrast, the shape of the PM_1 quadtree is unique. Figure 10a–e shows some of the steps in the process of building the PMR quadtree of Figure 10e. This structure assumes that the splitting threshold value is two. In each part of Figure 10a–e, the line segment that caused the subdivision is denoted by a thick line, while the gray regions indicate the blocks where a subdivision has taken place. The insertion of line segments c, e, g, h, and i cause the subdivisions in parts a, b, c, d, and e, respectively, of Figure 10. The insertion of line segment i causes three blocks to be subdivided (i.e., the SE block in the SW quadrant, the SE quadrant, and the SW block in the NE quadrant). The final result is shown in Figure 10e. Note the difference from the PM_1 quadtree in Figure 9 – 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.

The PMR quadtree is very good for answering queries such as finding the nearest line to a given point [Hoel and Samet 1991]. It is preferred over the PM_1 quadtree 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 occur, many empty blocks are created and thus the storage requirements of the PMR quadtree are considerably lower than those of the PM_1 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. Another advantage of the PMR quadtree over the PM_1 quadtree is that by virtue of being edge based, it can easily deal with nonplanar graphs.

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 [Samet 1990b] 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).

7 Concluding Remarks

The use of hierarchical data structures in spatial databases enables the focussing of 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 of a spatial index. All features, be they regions, points, rectangles, lines, volumes, etc., can be represented by maps which are in registration. This means that a query such as "finding the names of the roads that pass through the University of Maryland region" can be executed by simply overlaying the region and road maps even though they represent data of different types. The overlay performs an intersection operation where the common feature is the area spanned by the University of Maryland and the roads that pass through it (i.e., a spatial join).

In fact, such a system, known as QUILT, has been built [Shaffer et al. 1990] for representing geographic information using the quadtree variants described here. In this case, the quadtree is implemented as a collection of leaf nodes where each leaf node is represented by its locational code, which is a spatial index. The collection is in turn represented as a B-tree. There are leaf nodes corresponding to region, point, and line data.

The disadvantage of quadtree methods is that they are shift sensitive in the sense that their space requirements are dependent on the position of the origin. However, for complicated images the optimal positioning of the origin will usually lead to little improvement in the space requirements. The process of obtaining this optimal positioning is computationally expensive and is usually not worth the effort [Li et al. 1982].

The fact that we are working in a digitized space may also lead to problems. For example, the rotation operation is not generally invertible. In particular, a rotated square usually cannot be represented accurately by a collection of rectilinear squares. However, when we rotate by 90°, then the rotation is invertible. This problem arises whenever one uses a digitized representation. Thus, it is also common to the array representation.

8 Acknowledgements

The assistance of Erik G. Hoel in the preparation of the figures is greatly appreciated.

References

Aref, W. G. and Samet, H. 1990. Efficient processing of window queries in the pyramid data structure. In *Proceedings of the 9th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, (Nashville, TN, Apr.), pp. 265–272.

Aref, W. G. and Samet, H. 1991a. Extending a DBMS with spatial operations. In Günther, O. and Schek, H.-J., Eds., *Advances in Spatial Databases*, pp. 299–318. Lecture Notes in Computer Science 525. Springer-Verlag, Berlin.

Aref, W. G. and Samet, H. 1991b. Optimization strategies for spatial query processing. In Lohman, G., Ed., *Proceedings of the Seventeenth International Conference on Very Large Databases (VLDB)*, (Barcelona, Spain, Sept.), pp. 81–90.

Aref, W. G. and Samet, H. 1992. Uniquely reporting spatial objects: yet another operation for comparing spatial data structures. In *Proceedings of the Fifth International Symposium on Spatial Data Handling*, volume 1, (Charleston, SC, Aug.), pp. 178–189.

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

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

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

Blum, H. 1967. A transformation for extracting new descriptors of shape. In Wathen-Dunn, W., Ed., *Models for the Perception of Speech and Visual Form*, pp. 362–380. MIT Press, Cambridge, MA.

Buchmann, A., Günther, O., Smith, T. R., and Wang, Y.-F., Eds. 1989. *Design and Implementation of Large Spatial Databases — First Symposium, SSD'89*, Santa Barbara, CA, (July). Also Springer-Verlag Lecture Notes in Computer Science 409.

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

Edelsbrunner, H. 1980. Dynamic rectangle intersection searching. Institute for Information Processing Report 47, Institute for Information Processing, Technical University of Graz, Graz, Austria, (Feb.).

Finkel, R. and Bentley, J. 1974. Quad trees: a data structure for retrieval on composite keys. *Acta Informatica*, *4*, 1, 1–9.

Foley, J. D., van Dam, A., Feiner, S. K., and Hughes, J. F. 1990. *Computer Graphics: Principles and Practice*. Addison-Wesley, Reading, MA, second edition.

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

Freeman, H. 1974. Computer processing of line–drawing images. *ACM Computing Surveys*, *6*, 1 (Mar.), 57–97.

Freeston, M. 1987. The BANG file: a new kind of grid file. In *Proceedings of the SIGMOD Conference*, (San Francisco, May), pp. 260–269.

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

Günther, O. 1988. *Efficient structures for geometric data management.* PhD thesis, Electronics Research Laboratory, College of Engineering, University of California at Berkeley, Berkeley, CA. (Lecture Notes in Computer Science 337, Springer-Verlag, Berlin, 1988).

Günther, O. and Schek, H.-J., Eds. 1991. *Advances in Spatial Databases.* Lecture Notes in Computer Science 525. Springer-Verlag, Berlin.

Guttman, A. 1984. R–trees: a dynamic index structure for spatial searching. In *Proceedings of the SIGMOD Conference*, (Boston, MA, June), pp. 47–57.

Henrich, A., Six, H. W., and Widmayer, P. 1989. The lsd tree: spatial access to multidimensional point and non-point data. In Apers, P. M. G. and Wiederhold, G., Eds., *Proceedings of the Fifteenth International Conference on Very Large Databases (VLDB)*, (Amsterdam, The Netherlands, Aug.), pp. 45–53.

Hinrichs, K. and Nievergelt, J. 1983. The grid file: a data structure designed to support proximity queries on spatial objects. In Nagl, M. and J. Perl, e., Eds., *Proceedings of the WG'83 (International Workshop on Graphtheoretic Concepts in Computer Science)*, (Truner Verlag, Linz, Austria,), pp. 100–113.

Hoel, E. G. and Samet, H. 1991. Efficient processing of spatial queries in line segment databases. In Günther, O. and Schek, H.-J., Eds., *Advances in Spatial Databases*, pp. 237–256. Lecture Notes in Computer Science 525. Springer-Verlag, Berlin.

Hoel, E. G. and Samet, H. 1992. A qualitative comparison study of data structures for large line segment databases. In *Proceedings of the SIGMOD Conference*, (San Diego, June), pp. 205–214.

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

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

Jagadish, H. V. 1990. On indexing line segments. In McLeod, D., Sacks-Davis, R., and Schek, H., Eds., *Proceedings of the Sixteenth International Conference on Very Large Databases (VLDB)*, (Brisbane, Australia, Aug.), pp. 614–625.

Kawaguchi, E. and Endo, T. 1980. On a method of binary picture representation and its application to data compression. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *2*, 1 (Jan.), 27–35.

Kedem, G. 1982. The quad–cif tree: a data structure for hierarchical on–line algorithms. In *Proceedings of the Nineteenth Design Automation Conference*, (Las Vegas, NV, June), pp. 352–357.

Klinger, A. 1971. Patterns and search statistics. In Rustagi, J. S., Ed., *Optimizing Methods in Statistics*, pp. 303–337. Academic Press, New York.

Li, M., Grosky, W. I., and Jain, R. 1982. Normalized quadtrees with respect to translations. *Computer Graphics and Image Processing*, *20*, 1 (Sept.), 72–81.

Meagher, D. 1980. Octree encoding: a new technique for the representation, the manipulation, and display of arbitrary 3–d objects by computer. Technical Report IPL-TR-80-111, Electrical and Systems Engineering Technical Report, Rensselaer Polytechnic Institute, Troy, NY, (Oct.).

MEAGHER, D. 1982. Geometric modeling using octree encoding. *Computer Graphics and Image Processing*, *19*, 2 (June), 129–147.

NELSON, R. C. AND SAMET, H. 1986. A consistent hierarchical representation for vector data. *Computer Graphics*, *20*, 4 (Aug.), 197–206. (Also *Proceedings of the SIGGRAPH'86 Conference*, Dallas, August 1986).

NELSON, R. C. AND SAMET, H. 1987. A population analysis for hierarchical data structures. In *Proceedings of the SIGMOD Conference*, (May), pp. 270–277.

NIEVERGELT, J., HINTERBERGER, H., AND SEVCIK, K. C. 1984. The grid file: an adaptable, symmetric multikey file structure. *ACM Transactions on Database Systems*, *9*, 1 (Mar.), 38–71.

ORENSTEIN, J. A. 1982. Multidimensional tries used for associative searching. *Information Processing Letters*, *14*, 4 (June), 150–157.

ORENSTEIN, J. A. 1989. Redundancy in spatial databases. In *Proceedings of the SIGMOD Conference*, (Portland, OR, June), pp. 294–305.

ORENSTEIN, J. A. AND MERRETT, T. H. 1984. A class of data structures for associative searching. In *Proceedings of the Third ACM SIGACT–SIGMOD Symposium on Principles of Database Systems*, (Waterloo, Canada, Apr.), pp. 181–190.

PEUQUET, D. J. AND MARBLE, D. F. 1990. ARC/INFO: An example of a contemporary geographic information system. In PEUQUET, D. J. AND MARBLE, D. F., EDS., *Introductory Readings In Geographic Information Systems*, pp. 90–99. Taylor & Francis, London.

PREPARATA, F. P. AND SHAMOS, M. I. 1985. *Computational Geometry: An Introduction.* Springer–Verlag, New York.

ROBINSON, J. T. 1981. The k–d–b–tree: a search structure for large multidimensional dynamic indexes. In *Proceedings of the SIGMOD Conference*, (Ann Arbor, MI, Apr.), pp. 10–18.

RUTOVITZ, D. 1968. Data structures for operations on digital images. In ET AL., G. C. C., ED., *Pictorial Pattern Recognition*, pp. 105–133. Thompson Book Co., Washington, DC.

SAMET, H. 1990a. *Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS.* Addison-Wesley, Reading, MA.

SAMET, H. 1990b. *The Design and Analysis of Spatial Data Structures.* Addison-Wesley, Reading, MA.

SAMET, H., ROSENFELD, A., SHAFFER, C. A., AND WEBBER, R. E. 1984. A geographic information system using quadtrees. *Pattern Recognition*, *17*, 6 (November/December), 647–656.

SAMET, H. AND WEBBER, R. E. 1985. Storing a collection of polygons using quadtrees. *ACM Transactions on Graphics*, *4*, 3 (July), 182–222. (Also *Proceedings of Computer Vision and Pattern Recognition 83*, Washington, DC, June 1983, 127–132; and University of Maryland Computer Science TR-1372).

SAMET, H. AND WEBBER, R. E. 1989. A comparison of the space requirements of multi–dimensional quadtree–based file structures. *Visual Computer*, *5*, 6 (Dec.), 349–359. (Also University of Maryland Computer Science TR-1711).

SEEGER, B. AND KRIEGEL, H. P. 1990. The buddy-tree: an efficient and robust access method for spatial data base systems. In MCLEOD, D., SACKS-DAVIS, R., AND SCHEK, H., EDS., *Proceedings of the Sixteenth International Conference on Very Large Databases (VLDB)*, (Brisbane, Australia, Aug.), pp. 590–601.

SELLIS, T., ROUSSOPOULOS, N., AND FALOUTSOS, C. 1987. The R$^+$-tree: a dynamic index for multi-dimensional objects. In STOCKER, P. M. AND KENT, W., EDS., *Proceedings of the Thirteenth International Conference on Very Large Databases (VLDB)*, (Brighton, England, Sept.), pp. 507–518.

SHAFFER, C. A., SAMET, H., AND NELSON, R. C. 1990. QUILT: A geographic information system based on quadtrees,. *International Journal of Geographical Information Systems*, *4*, 2 (April–June), 103–131.

TAMMINEN, M. 1981. The EXCELL method for efficient geometric access to data. *Acta Polytechnica Scandinavica*, Mathematics and Computer Science Series, No. 34, Helsinki, Finland.

TANIMOTO, S. AND PAVLIDIS, T. 1975. A hierarchical data structure for picture processing. *Computer Graphics and Image Processing*, *4*, 2 (June), 104–119.

References

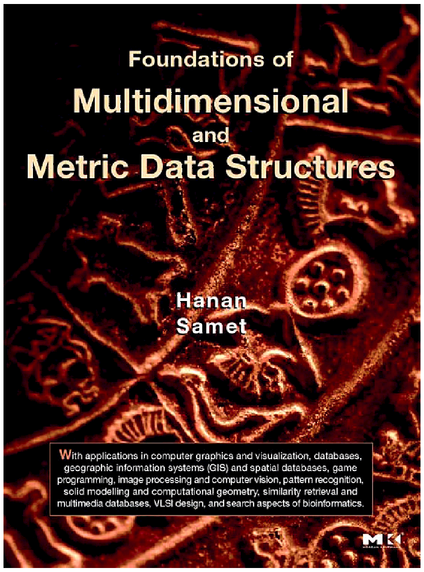
1. H. Samet. Foundations of Multidimensional and Metric Data Structures. Morgan-Kaufmann, San Francisco, CA, USA, 2005. (table of contents and order form on next page)
2. H. Samet. The Design and Analysis of Spatial Data Structures, Addison-Wesley, Reading, MA, 1990. (out of print; contact Hanan Samet for details)
3. H. Samet. Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS, Addison-Wesley, Reading, MA, 1990. (out of print; contact Hanan Samet for details)

MORGAN KAUFMANN PUBLISHERS

Foundations of Multidimensional and Metric Data Structures

By Hanan Samet, *University of Maryland at College Park* **1024 pages**

August 2006 ▪ ISBN 0-12-369446-9 ▪ Hardcover ▪ ~~$59.95~~▪ ~~£34.99~~ ▪ ~~€49.95~~ ▪ $47.96 ▪ £27.99 ▪ €39.96



20% OFF!

The field of multidimensional and metric data structures is large and growing very quickly. Here, for the first time, is a thorough treatment of multidimensional point data, object and image-based object representations, intervals and small rectangles, high-dimensional datasets, as well as datasets for which we only know that they reside in a metric space.

The book includes a thorough introduction; a comprehensive survey of multidimensional (including spatial) and metric data structures and algorithms; and implementation details for the most useful data structures. Along with the hundreds of worked exercises and hundreds of illustrations, the result is an excellent and valuable reference tool for professionals in many areas, including computer graphics and visualization, databases, geographic information systems (GIS), and spatial databases, game programming, image processing and computer vision, pattern recognition, solid modelling and computational geometry, similarity retrieval and multimedia databases, and VLSI design, and search aspects of bioinformatics.

Features

- First comprehensive work on multidimensional and metric data structures available, a thorough and authoritative treatment.
- An algorithmic rather than mathematical approach, with a liberal use of examples that allows the readers to easily see the possible implementation and use.
- Each section includes a large number of exercises and solutions to self-test and confirm the reader's understanding and suggest future directions.
- Written by a well-known authority in the area of multidimensional (including spatial) data structures who has made many significant contributions to the field.

SIGGRAPH 2007

Hanan Samet is the dean of "spatial indexing"... This book is encyclopedic... this book will be invaluable for those of us who struggle with spatial data, scientific datasets, graphics, vision problems involving volumetric queries, or with higher dimensional datasets common in data mining.

- From the foreword by Jim Gray, Microsoft Research

Samet's book on multidimensional and metric data structures is the most complete and thorough presentation on this topic. It has broad coverage of material from computational geometry, databases, graphics, GIS, and similarity retrieval literature. Written by the leading authority on hierarchical spatial representations, this book is a "must have" for all instructor, researches, and developers working and teaching in these areas.

- Dinesh Manocha, University of North Carolina at Chapel Hill

To summarize, this book is excellent! It's a very comprehensive survey of spatial and multidimensional data structures and algorithms, which is badly needed. The breadth and depth of coverage is astounding and I would consider several parts of it required reading for real time graphics and game developers.

- Bretton Wade, University of Washington and Microsoft Corp.

Order from Morgan Kaufmann Publishers and receive 20% off!

Please refer to code 85511.

Mail: Elsevier Science, Order Fulfillment, 11830 Westline Industrial Dr., St. Louis, MO 63146
Phone: US/Canada 800-545-2522, 1-314-453-7010 (Intl.) ▪ **Fax:** 800-535-9935, 1-314-453-7095 (Intl.)
Email: usbkinfo@elsevier.com ▪ **Visit Morgan** Kaufmann **on the Web:** www.mkp.com
Volume discounts available, contact: NASpecialSales@elsevier.com

MORGAN KAUFMANN PUBLISHERS
an imprint of Elsevier

Foundations of Multidimensional and Metric Data Structures

By Hanan Samet, *University of Maryland at College Park*

Available August 2006 • ISBN 0-12-369446-9 • 1024 pages • Hardcover ~~$59.95~~ • $47.96

Table of Contents and Topics

About the Author

Hanan Samet is Professor in the Department of Computer Science at the University of Maryland at College Park, and a member of the Center for Automation Research and the Institute for Advanced Computer Studies. He is widely published in the fields of spatial databases and data structures, computer graphics, image databases and image processing, and geographic information systems (GIS), and is considered an authority on the use and design of hierarchical spatial data structures such as the quadtree and octree for geographic information systems, image processing, and computer graphics. He is the author of the first two books on spatial data structures: *The Design and Analysis of Spatial Data Structures* and *Applications of Spatial Data Structures: Computer Graphics, Image Processing and GIS*. He holds a Ph.D. in computer science from Stanford University

SIGGRAPH 2007