SORTING IN SPACE

HANAN SAMET

COMPUTER SCIENCE DEPARTMENT AND
CENTER FOR AUTOMATION RESEARCH AND
INSTITUTE FOR ADVANCED COMPUTER STUDIES
UNIVERSITY OF MARYLAND

COLLEGE PARK, MARYLAND 20742-3411 USA

e–mail: hjs@cs.umd.edu
url: http://www.cs.umd.edu/~hjs

Foundations of Multidimensional and Metric Data Structures

Hanan Samet

With applications in 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, VLSI design, and search aspects of bioinformatics.

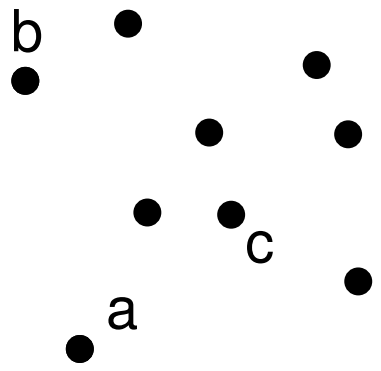
MK MORGAN KAUFMANN

Why Sorting of Spatial Data is Important

- Most operations invariably involve search
- Search is sped up by sorting the data
- *sort* - Definition: verb
  1. to put in a certain place or rank according to kind, class, or nature
  2. to arrange according to characteristics
- Examples
  1. Warnock algorithm: sorting objects for display
     - vector: hidden-line elimination
     - raster: hidden-surface elimination
  2. Back-to-front and front-to-back algorithms
  3. BSP trees for visibility determination
  4. Accelerating ray tracing and ray casting by finding ray-object intersections
  5. Bounding box hierarchies arrange space according to whether occupied or unoccupied

Sorting Implies the Existence of an Ordering

1. Fine for one-dimensional data
   - sort people by weight and find closest in weight to Bill and can also find closest in weight to Larry
   - sort cities by distance from Chicago and find closest to Chicago but cannot find closest to New York unless resort

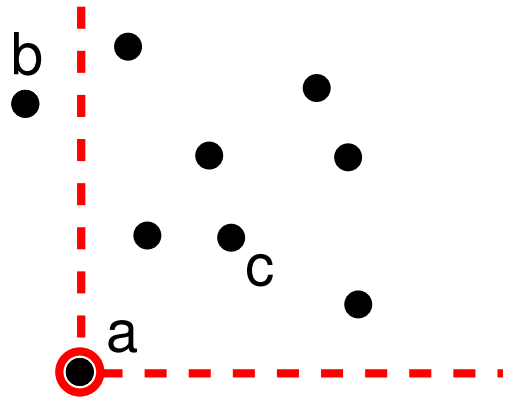


2. Hard for two-dimensions as higher as notion of ordering does not exist unless a dominance relation holds
   - point $a = \{a_i | 1 \leq i \leq d\}$ dominates point $b = \{b_i | 1 \leq i \leq d\}$ if $a_i \leq b_i, 1 \leq i \leq d$.

3. Only solution is to linearize data as in a space-filling curve
   - sort is explicit
   - need implicit sort so no need to resort if reference point changes

Sorting Implies the Existence of an Ordering

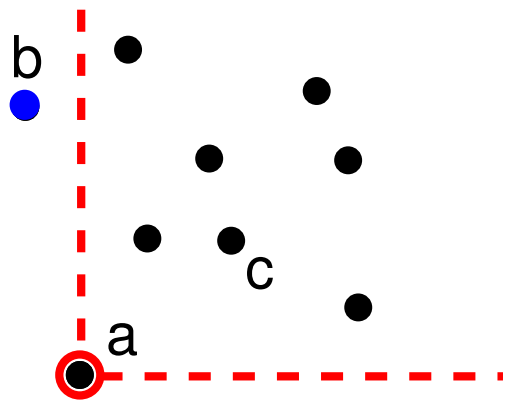
1. Fine for one-dimensional data
   - sort people by weight and find closest in weight to Bill and can also find closest in weight to Larry
   - sort cities by distance from Chicago and find closest to Chicago but cannot find closest to New York unless resort



2. Hard for two-dimensions as higher as notion of ordering does not exist unless a dominance relation holds
   - point $a = \{a_i | 1 \leq i \leq d\}$ dominates point $b = \{b_i | 1 \leq i \leq d\}$ if $a_i \leq b_i, 1 \leq i \leq d$.
   - a
3. Only solution is to linearize data as in a space-filling curve
   - sort is explicit
   - need implicit sort so no need to resort if reference point changes

Sorting Implies the Existence of an Ordering

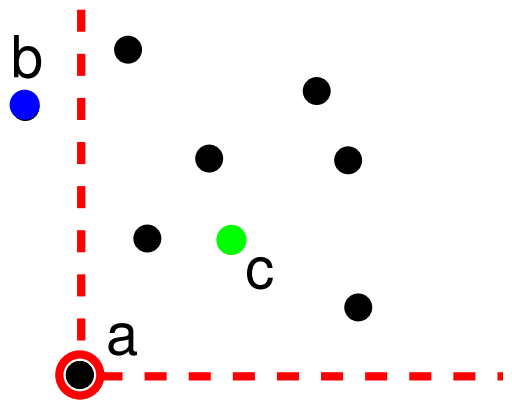
1. Fine for one-dimensional data
   - sort people by weight and find closest in weight to Bill and can also find closest in weight to Larry
   - sort cities by distance from Chicago and find closest to Chicago but cannot find closest to New York unless resort



2. Hard for two-dimensions as higher as notion of ordering does not exist unless a dominance relation holds
   - point $a = \{a_i | 1 \leq i \leq d\}$ dominates point $b = \{b_i | 1 \leq i \leq d\}$ if $a_i \leq b_i, 1 \leq i \leq d$.
   - a does not dominate b
3. Only solution is to linearize data as in a space-filling curve
   - sort is explicit
   - need implicit sort so no need to resort if reference point changes

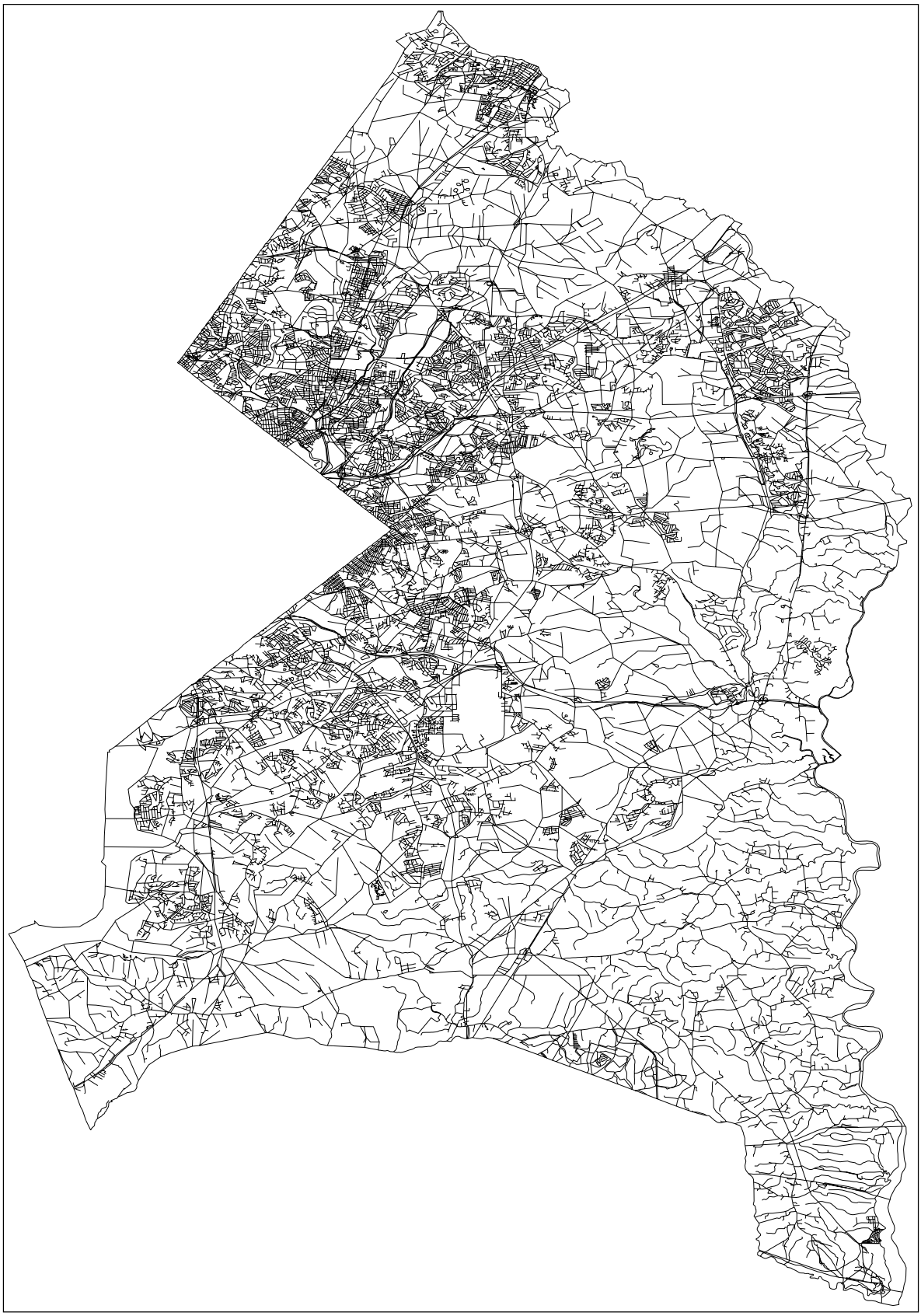
Sorting Implies the Existence of an Ordering

1. Fine for one-dimensional data
   - sort people by weight and find closest in weight to Bill and can also find closest in weight to Larry
   - sort cities by distance from Chicago and find closest to Chicago but cannot find closest to New York unless resort



2. Hard for two-dimensions as higher as notion of ordering does not exist unless a dominance relation holds
   - point $a = \{a_i | 1 \leq i \leq d\}$ dominates point $b = \{b_i | 1 \leq i \leq d\}$ if $a_i \leq b_i, 1 \leq i \leq d$.
   - a does not dominate b but dominates c
3. Only solution is to linearize data as in a space-filling curve
   - sort is explicit
   - need implicit sort so no need to resort if reference point changes

PRINCE GEORGES COUNTY



EXAMPLE QUERIES ON LINE SEGMENT DATABASES

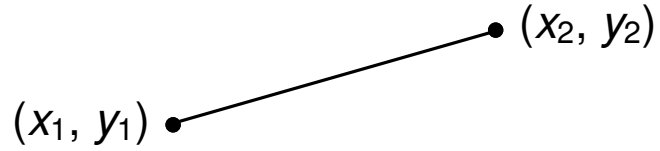
- Queries about line segments
  1. All segments that intersect a given point or set of points
  2. All segments that have a given set of endpoints
  3. All segments that intersect a given line segment
  4. All segments that are coincident with a given line segment
- Proximity queries
  1. The nearest 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

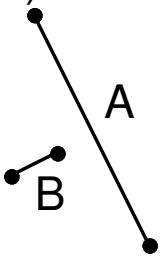
1. Spatial extent of the objects is the key to the difference

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

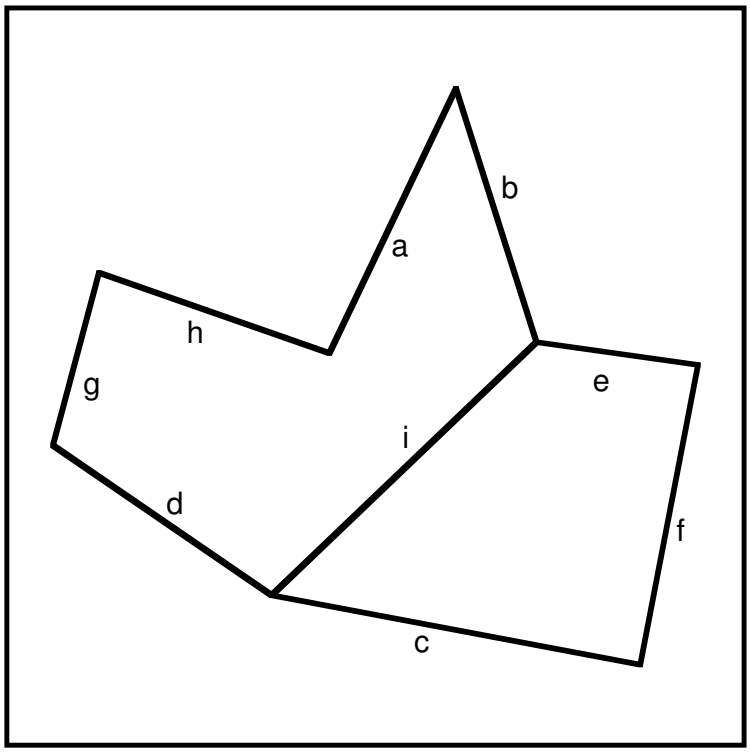
1. Compatibility with the data being stored
2. Choose an appropriate zero or reference point
3. Need an implicit rather than an explicit index
   - impossible to foresee all possible queries in advance
   - cannot have an attribute for every possible spatial relationship
     a. derive adjacency relations
     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

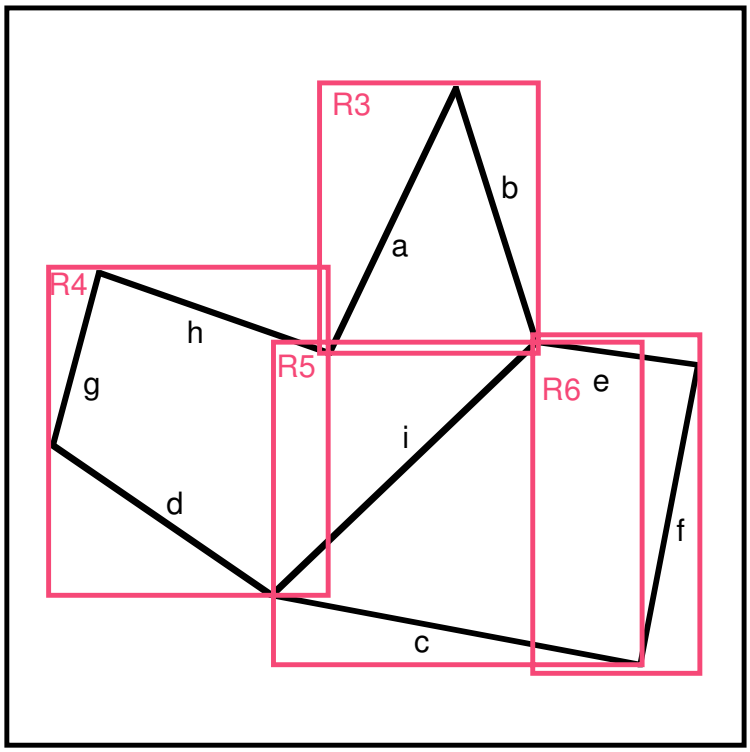
MINIMUM BOUNDING RECTANGLES

- 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



MINIMUM BOUNDING RECTANGLES

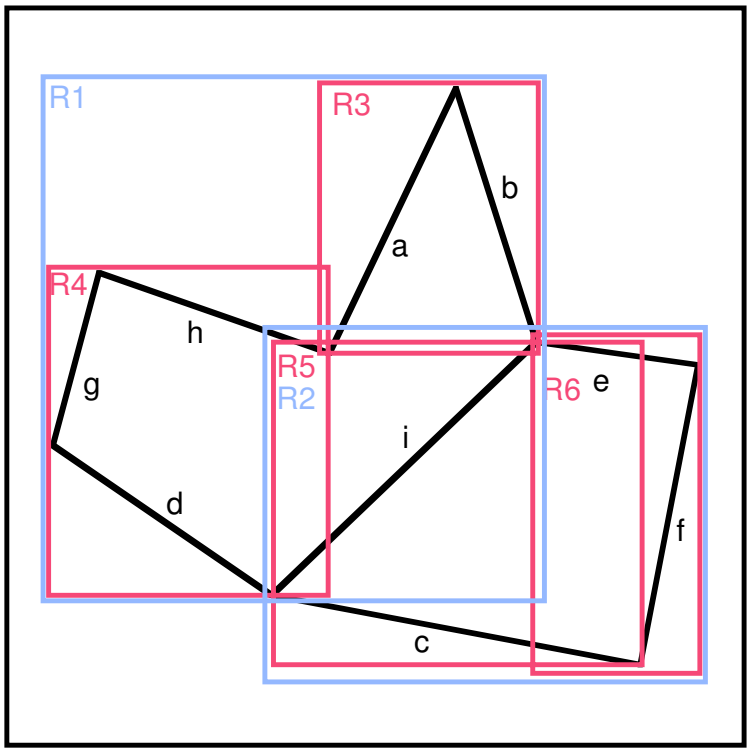
- Objects grouped into hierarchies, stored in a structure similar to a B-tree
- Drawback: not a disjoint decomposition of space
- Object has single bounding rectangle, yet area that it spans may be included in several bounding rectangles
- Examples include the R-tree and the R*-tree
- Order (m,M) R-tree
  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

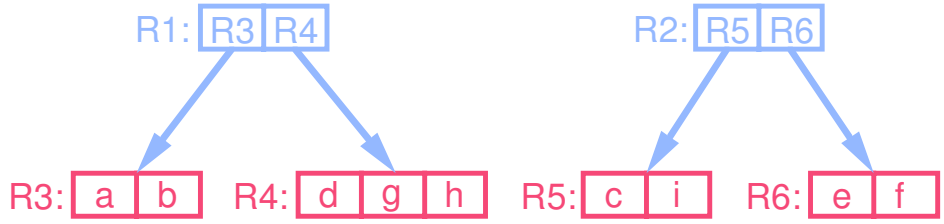


R3: a b R4: d g h R5: c i R6: e f

MINIMUM BOUNDING RECTANGLES

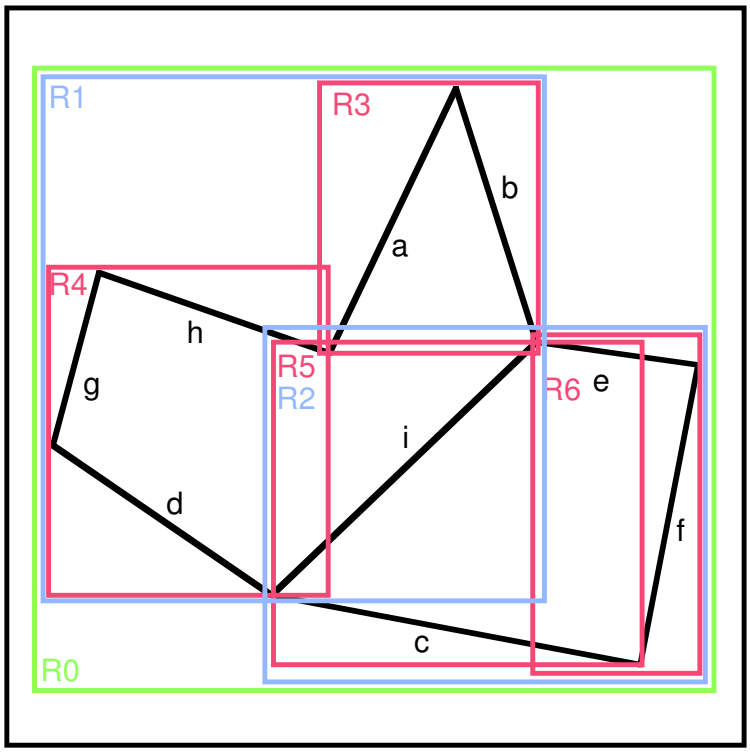
- Objects grouped into hierarchies, stored in a structure similar to a B-tree
- Drawback: not a disjoint decomposition of space
- Object has single bounding rectangle, yet area that it spans may be included in several bounding rectangles
- Examples include the R-tree and the R*-tree
- Order (m,M) R-tree
  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

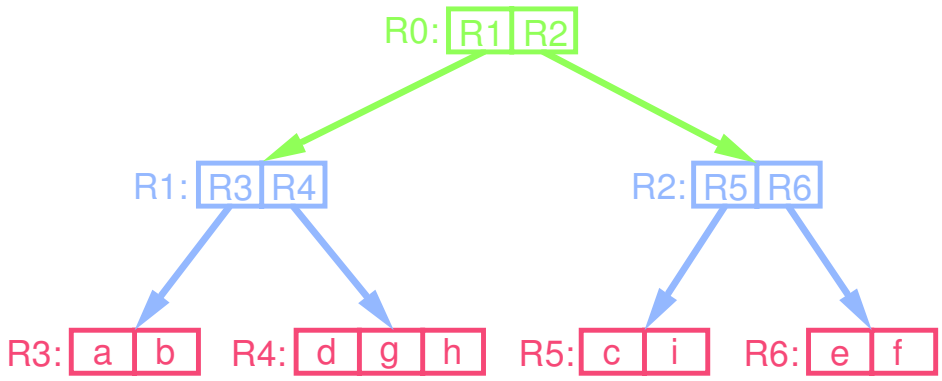




MINIMUM BOUNDING RECTANGLES

- 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

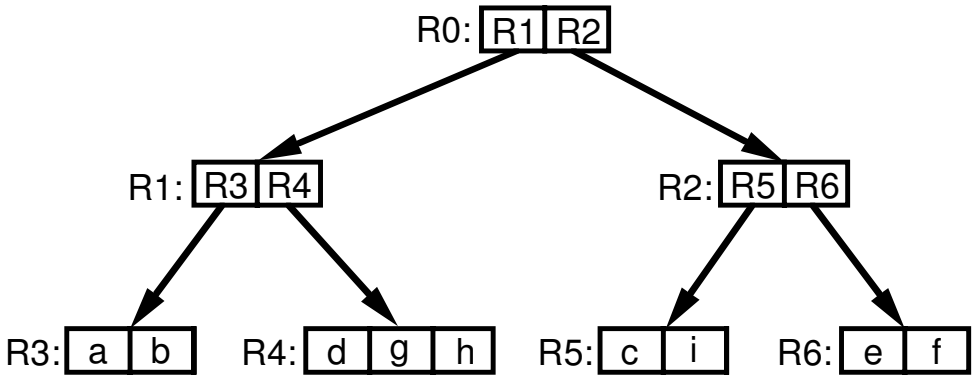


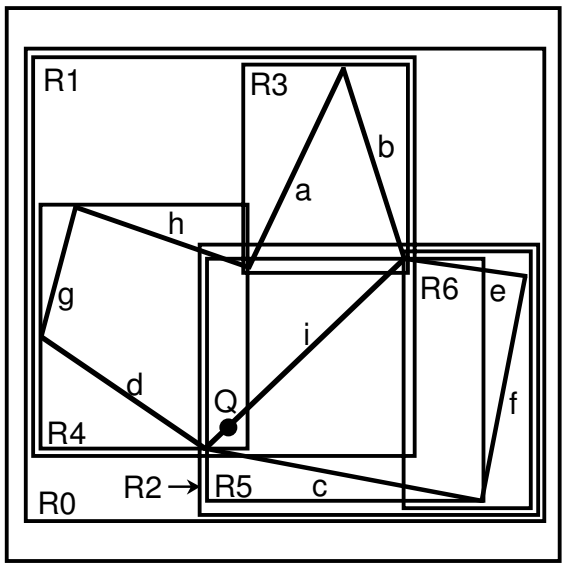


SEARCHING FOR A POINT OR LINE SEGMENT IN AN R-TREE

- 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

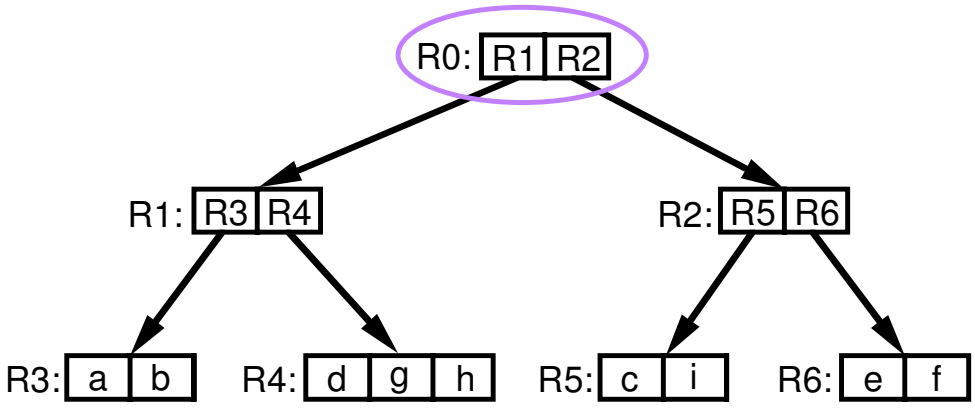


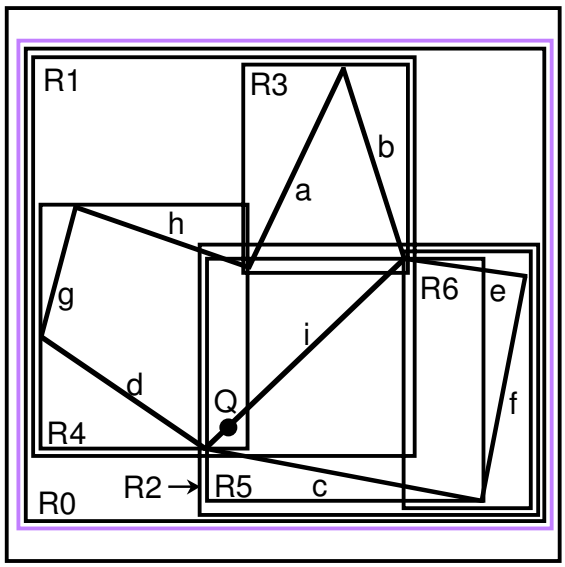


SEARCHING FOR A POINT OR LINE SEGMENT IN AN R-TREE

- 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



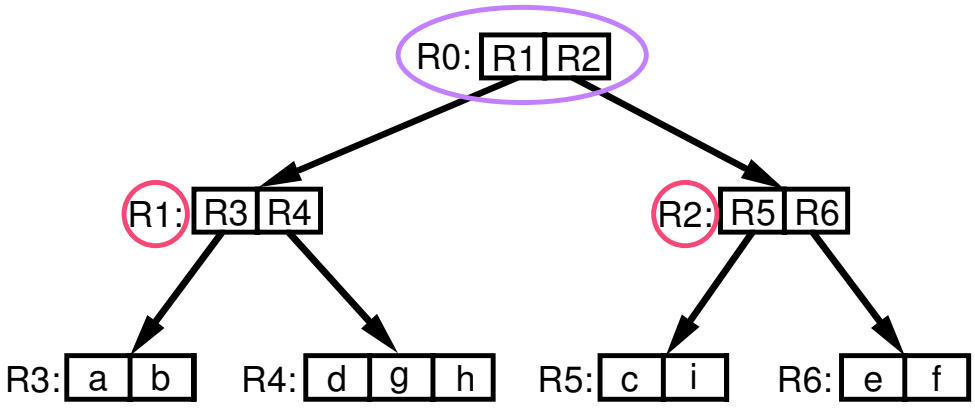


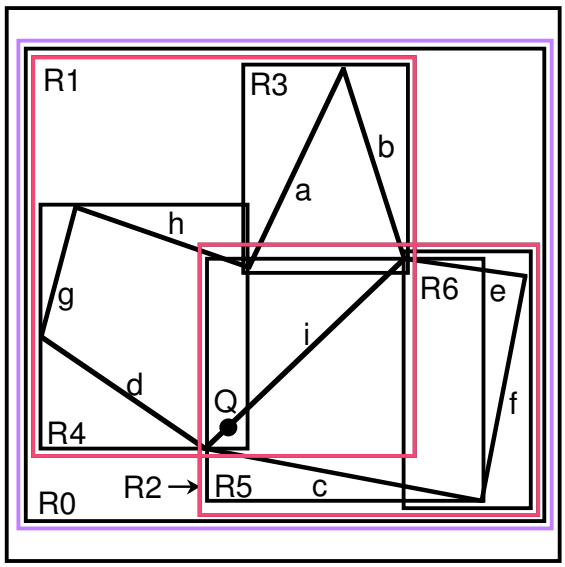
- Q is in R0

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



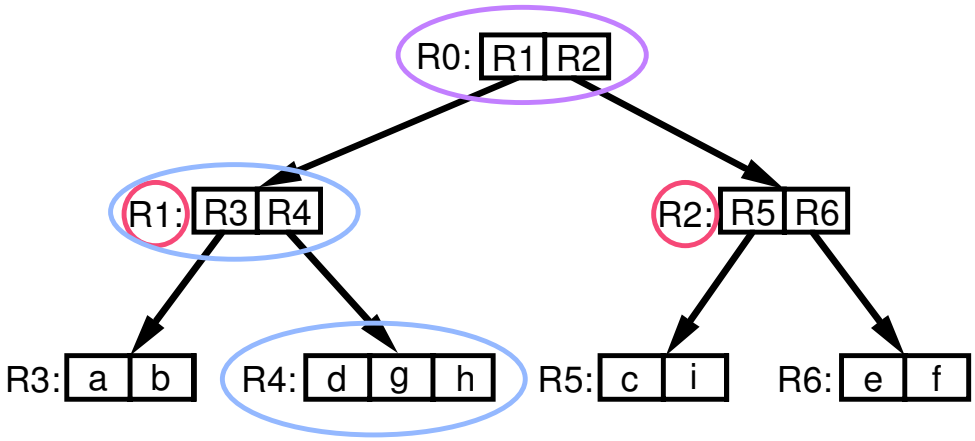


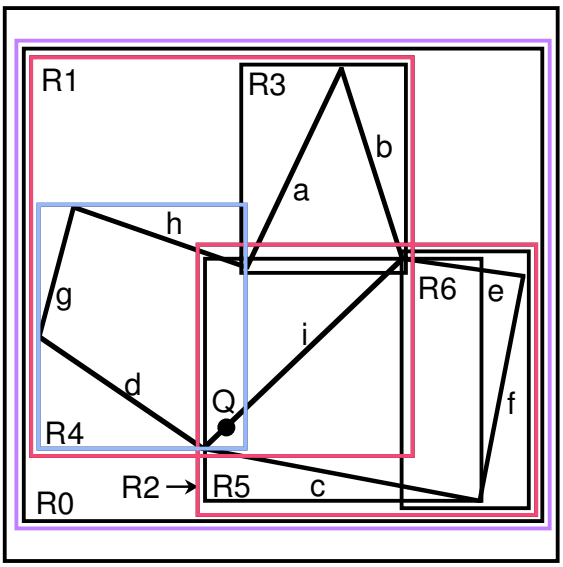
- Q is in R0
- Q can be in both R1 and R2

SEARCHING 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



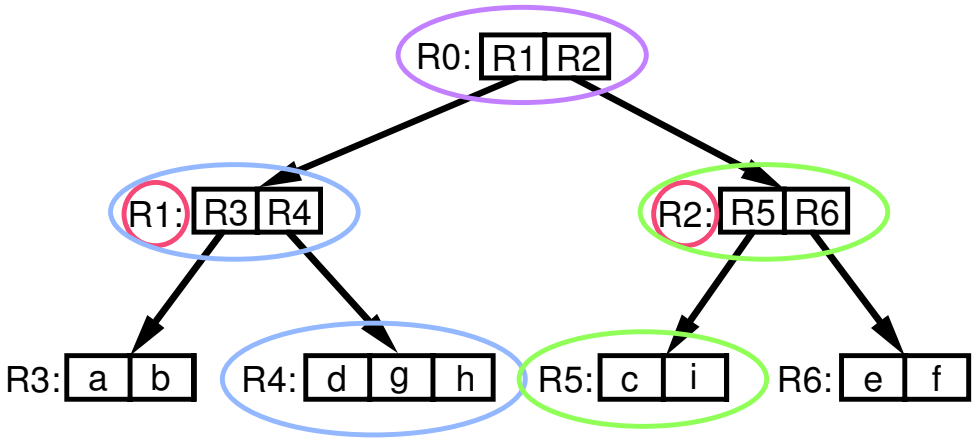


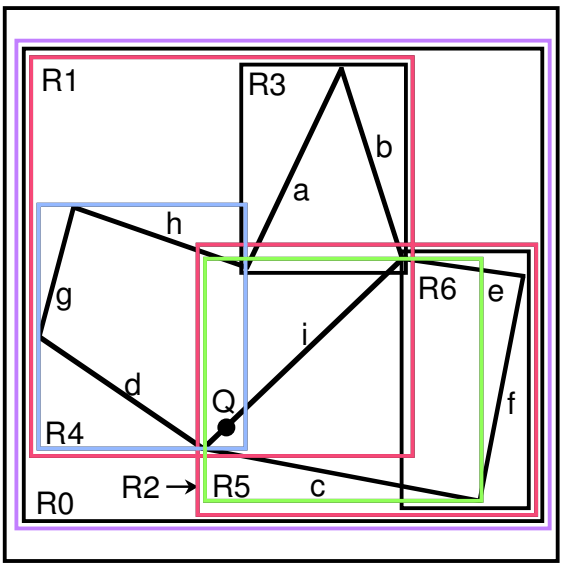
- Q is in R0
- Q can be in both R1 and R2
- Searching R1 first means that R4 is searched but this leads to failure even though Q is part of i which is in R4

SEARCHING 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

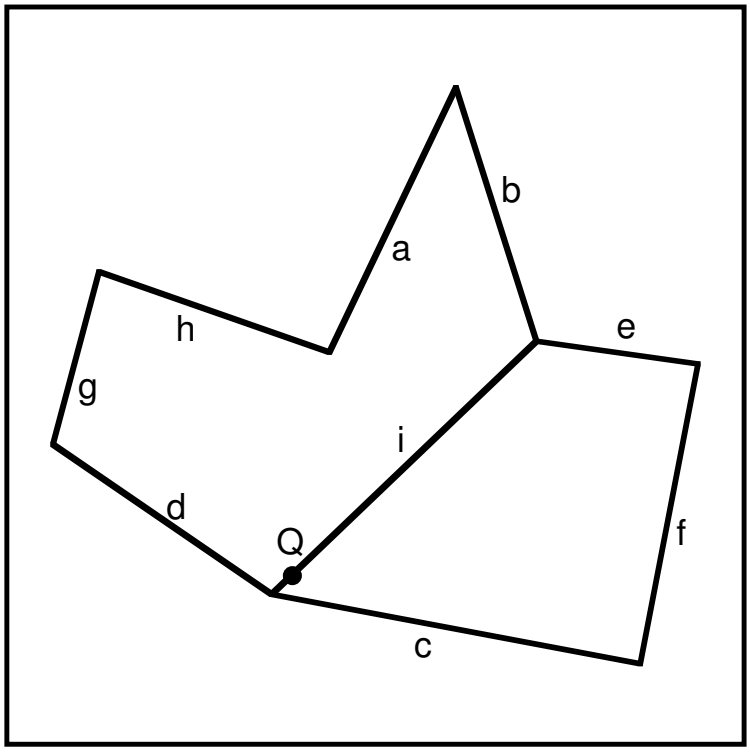




- Q is in R0
- Q can be in both R1 and R2
- Searching R1 first means that R4 is searched but this leads to failure even though Q is part of i which is in R4
- Searching R2 finds that Q can only be in R5

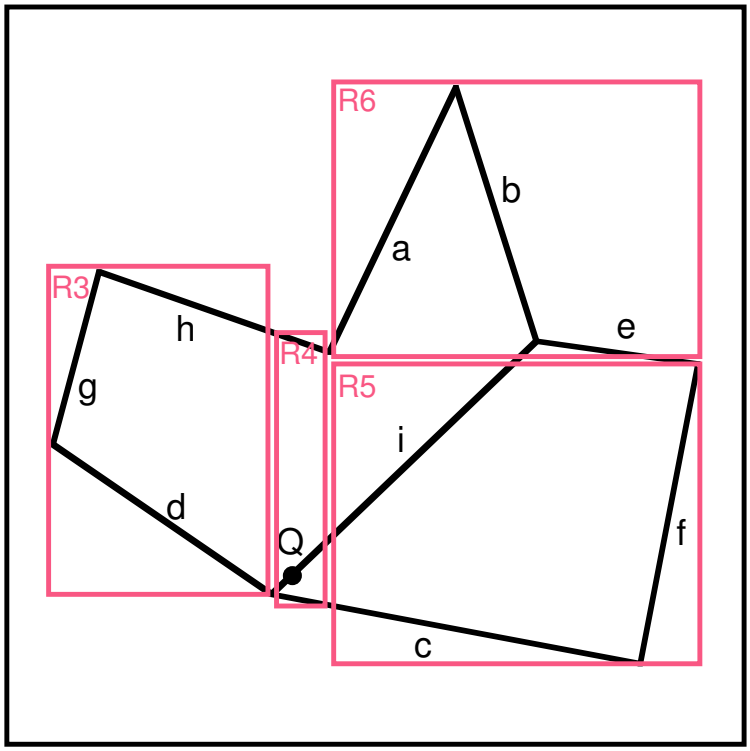
DISJOINT CELLS

- 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



DISJOINT CELLS

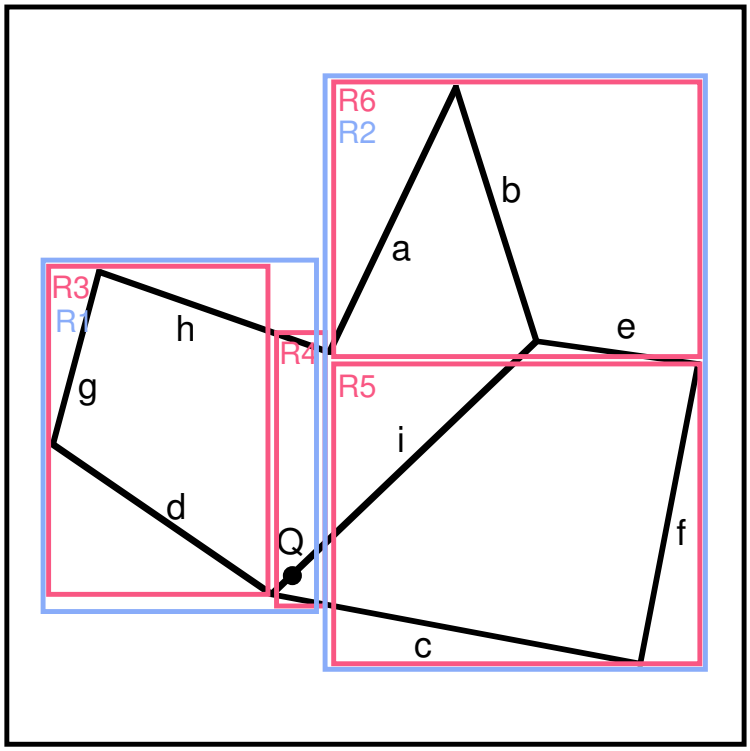
- Objects decomposed into disjoint subobjects; each subobject in different cell
- Techniques differ in degree of regularity
- Drawback: in order to determine area covered by object, must retrieve all cells that it occupies
- R+-tree (also k-d-B-tree) and cell tree are examples of this technique



R3: d g h R4: c h i R5: c f i R6: a b e i

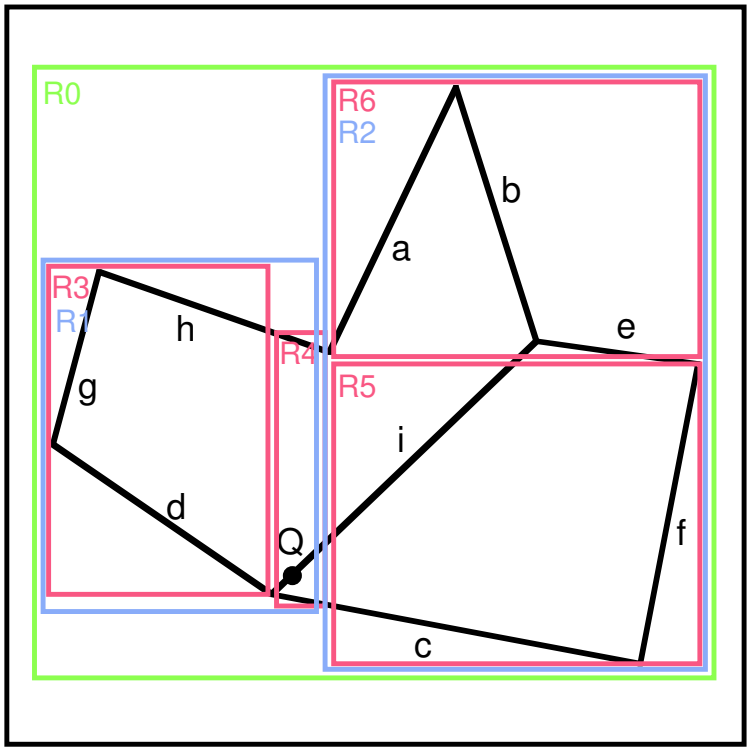
DISJOINT CELLS

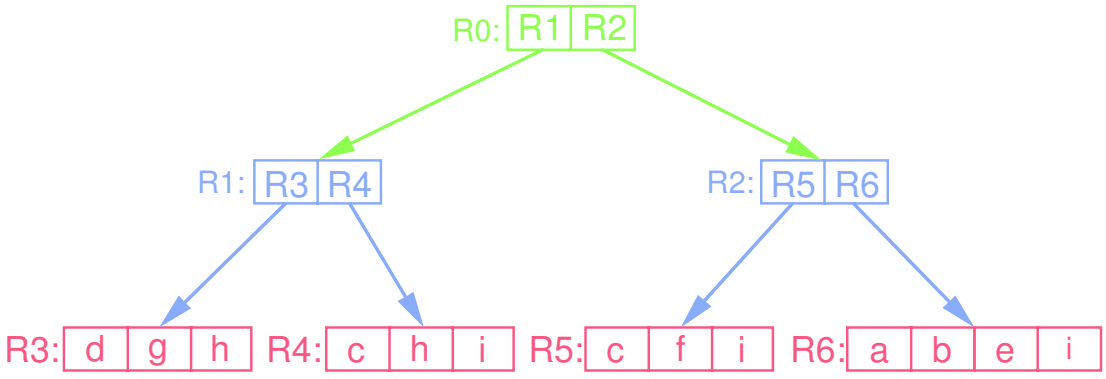
- Objects decomposed into disjoint subobjects; each subobject in different cell
- Techniques differ in degree of regularity
- Drawback: in order to determine area covered by object, must retrieve all cells that it occupies
- R+-tree (also k-d-B-tree) and cell tree are examples of this technique



DISJOINT CELLS

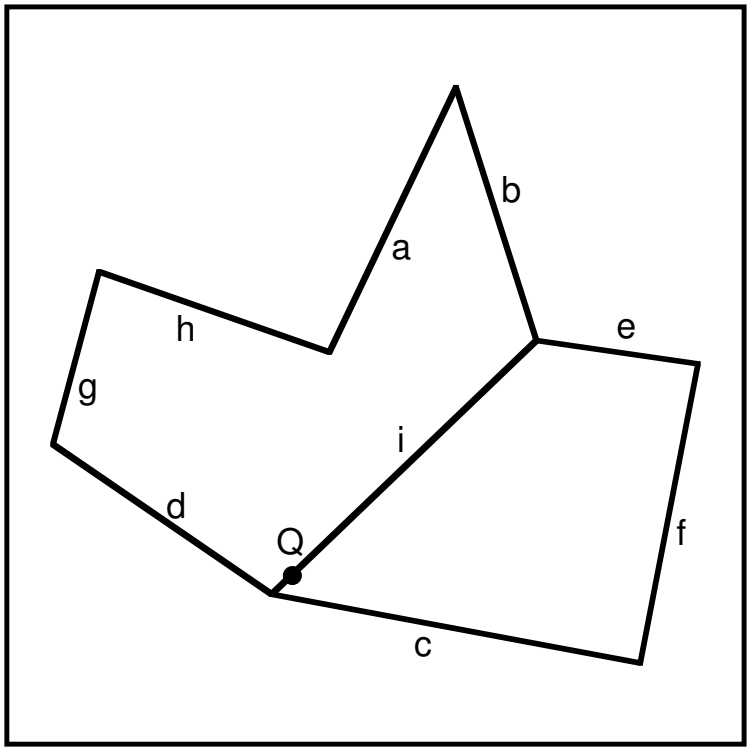
- Objects decomposed into disjoint subobjects; each subobject in different cell
- Techniques differ in degree of regularity
- Drawback: in order to determine area covered by object, must retrieve all cells that it occupies
- R+-tree (also k-d-B-tree) and cell tree are examples of this technique





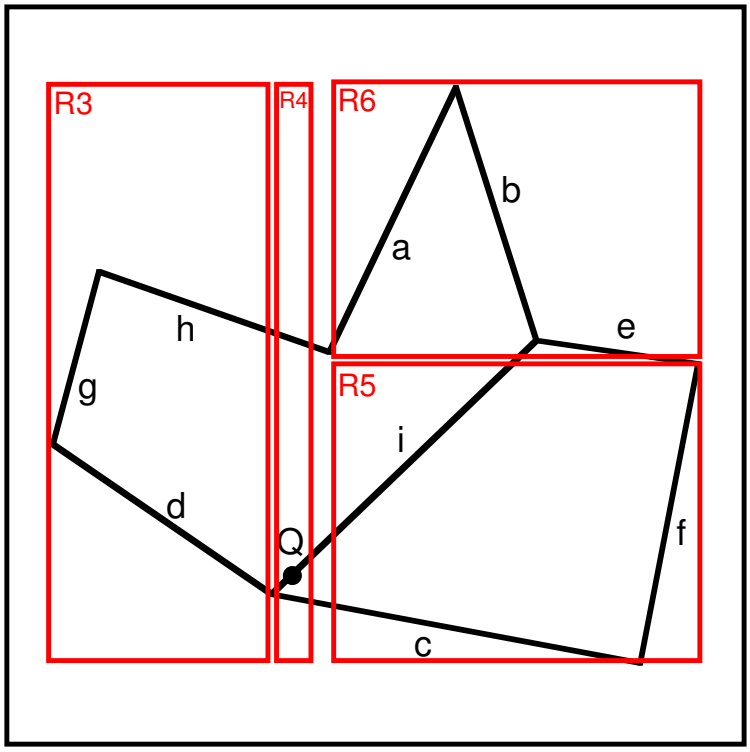
K-D-B-TREES

- 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



K-D-B-TREES

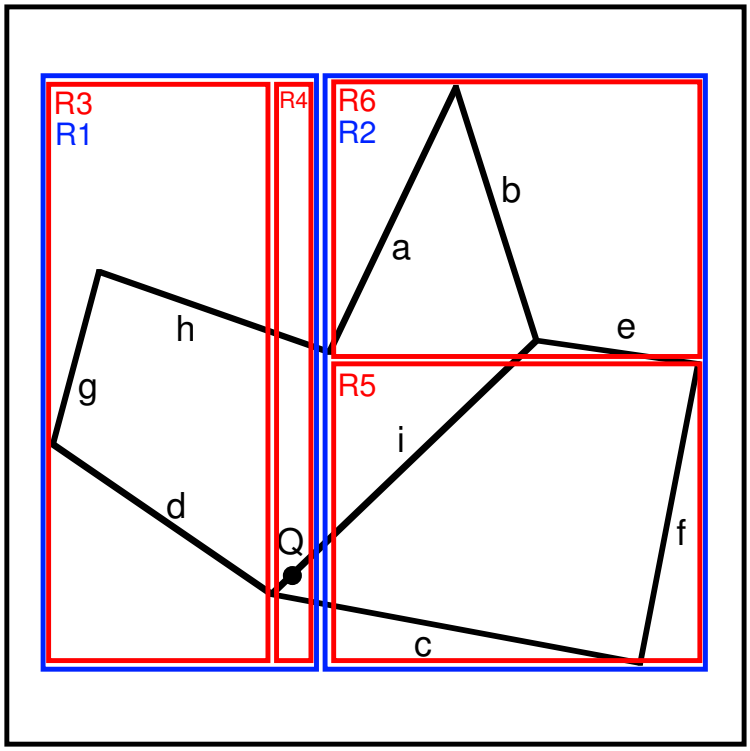
- Rectangular embedding space is hierarchically decomposed into disjoint rectangular regions
- No dead space in the sense that at any level of the tree, entire embedding space is covered by one of the nodes
- Blocks of k-d tree partition of space are aggregated into nodes of a finite capacity
- When 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



R3: d g h R4: c h i R5: c f i R6: a b e i

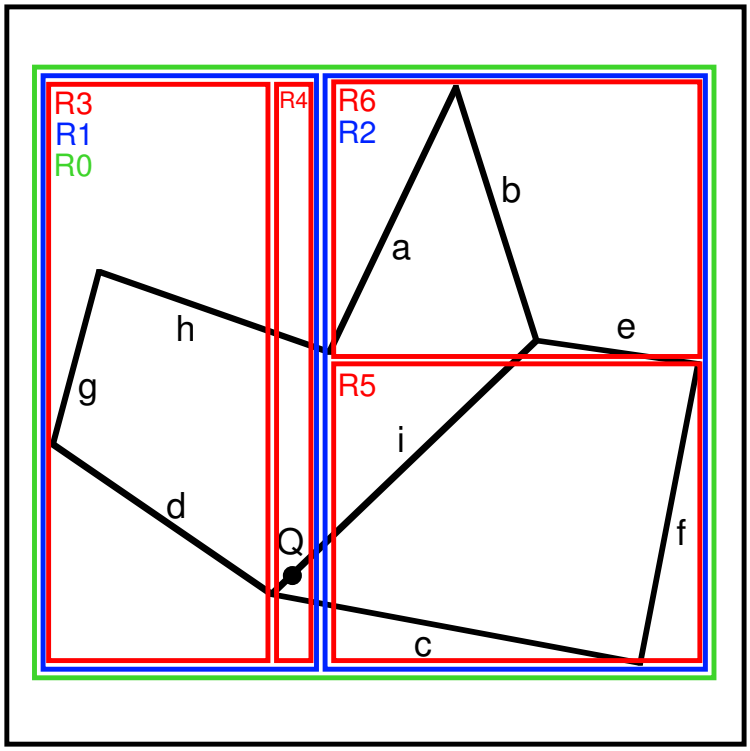
K-D-B-TREES

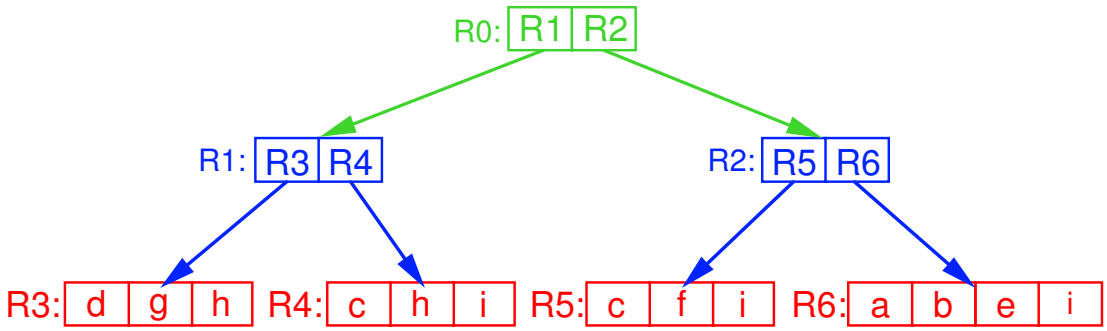
- Rectangular embedding space is hierarchically decomposed into disjoint rectangular regions
- No dead space in the sense that at any level of the tree, entire embedding space is covered by one of the nodes
- Blocks of k-d tree partition of space are aggregated into nodes of a finite capacity
- When 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



K-D-B-TREES

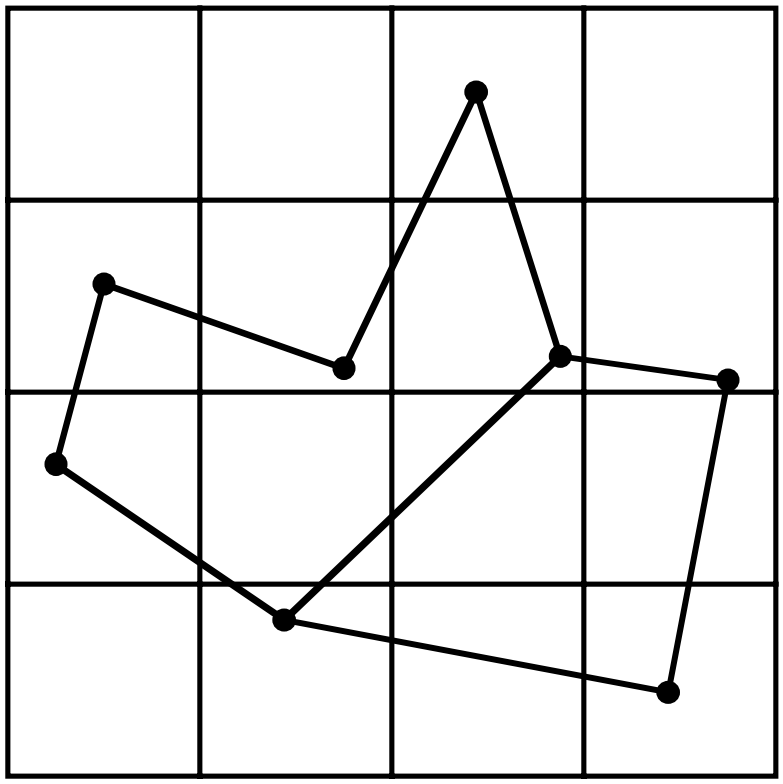
- Rectangular embedding space is hierarchically decomposed into disjoint rectangular regions
- No dead space in the sense that at any level of the tree, entire embedding space is covered by one of the nodes
- Blocks of k-d tree partition of space are aggregated into nodes of a finite capacity
- When 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





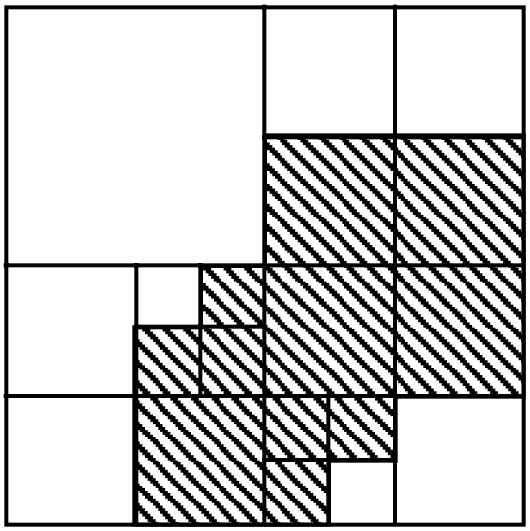
UNIFORM GRID

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



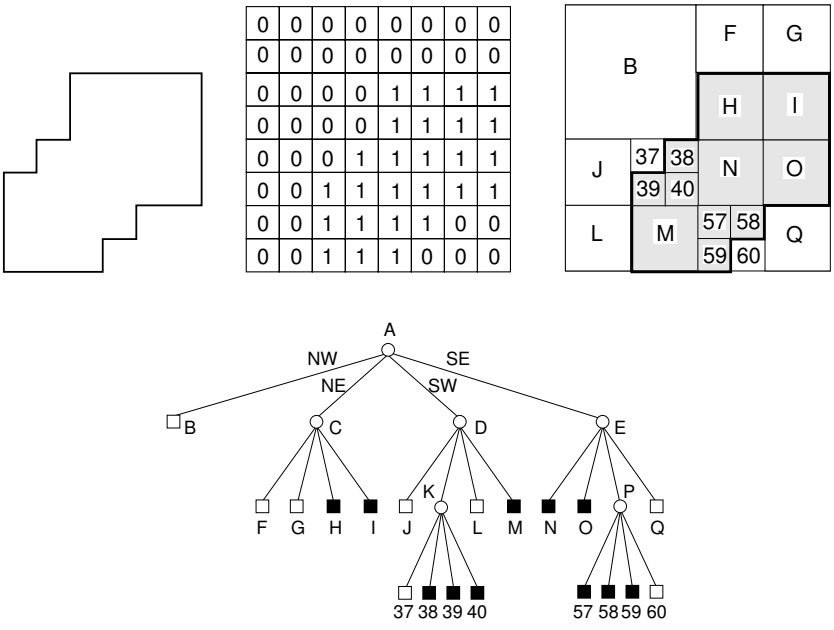
QUADTREES

- 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



SPACE REQUIREMENTS

1. Rationale for using quadtrees/octrees is not so much for saving space but for saving execution time

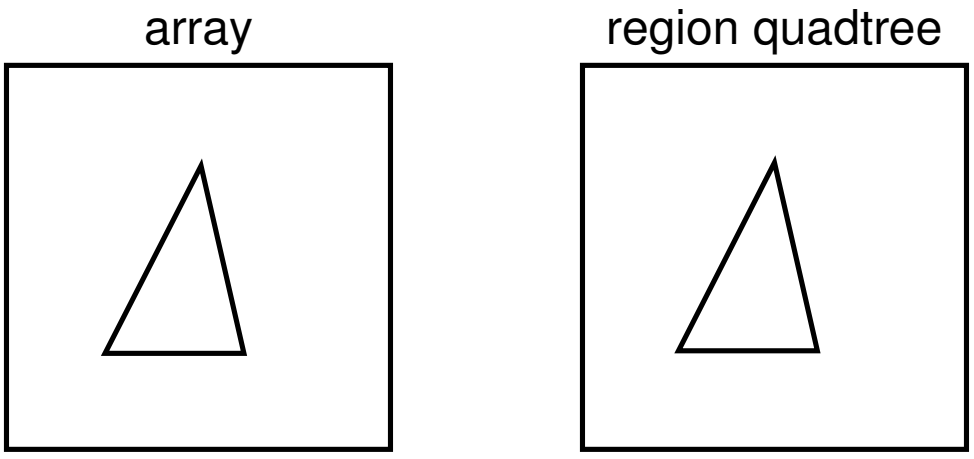
2. Execution time of standard image processing algorithms that are based on traversing the entire image and performing a computation at each image element is proportional to the number of blocks in the decomposition of the image rather than their size
   - aggregation of space leads directly to execution 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)

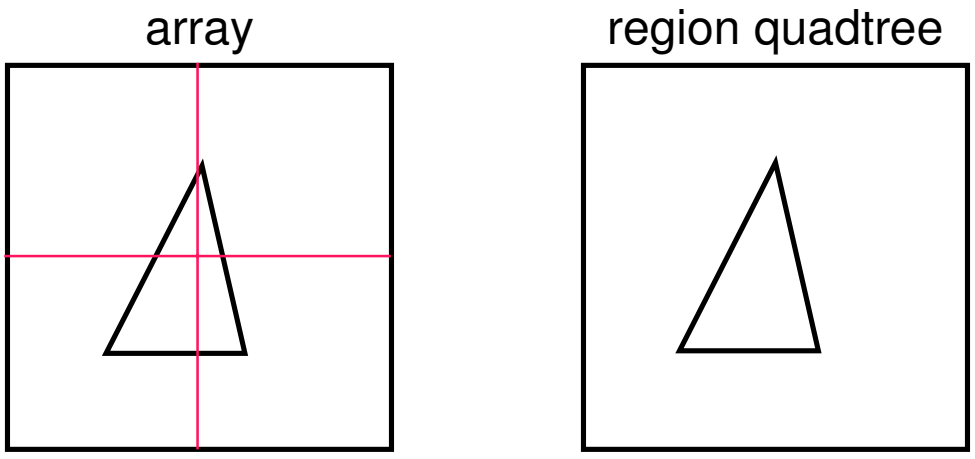
DIMENSION REDUCTION

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.



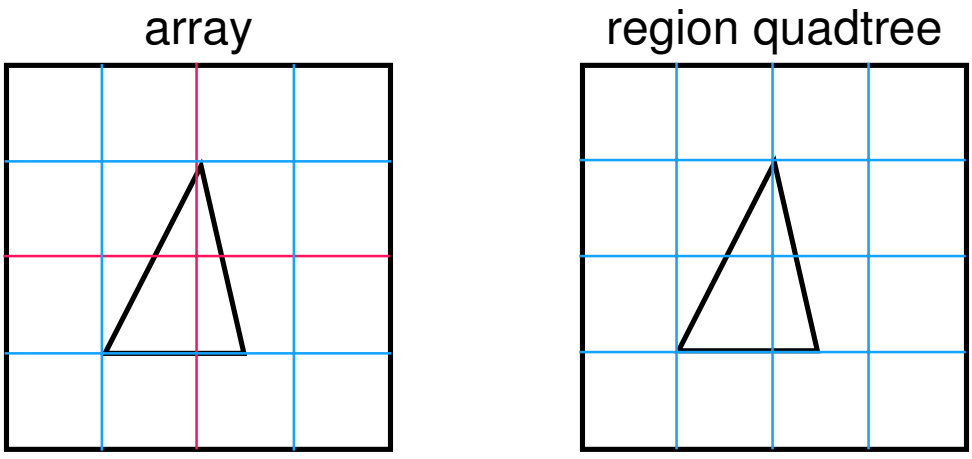
DIMENSION REDUCTION

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.



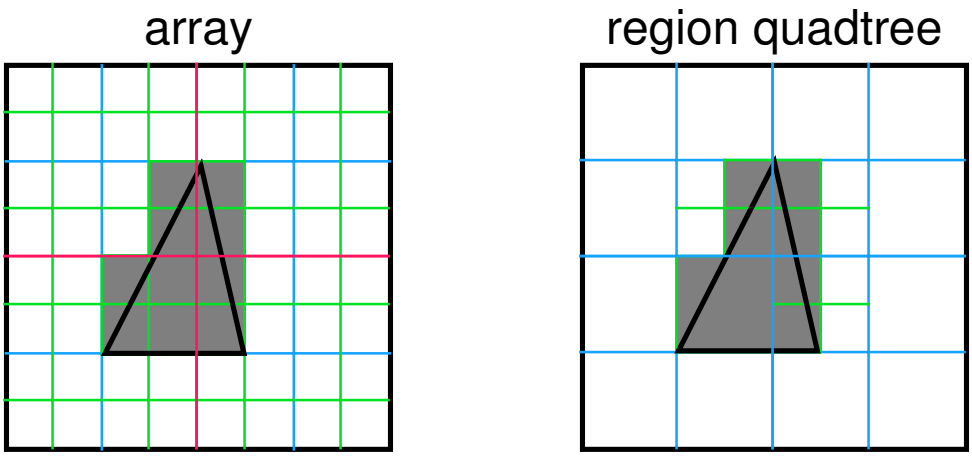
DIMENSION REDUCTION

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.



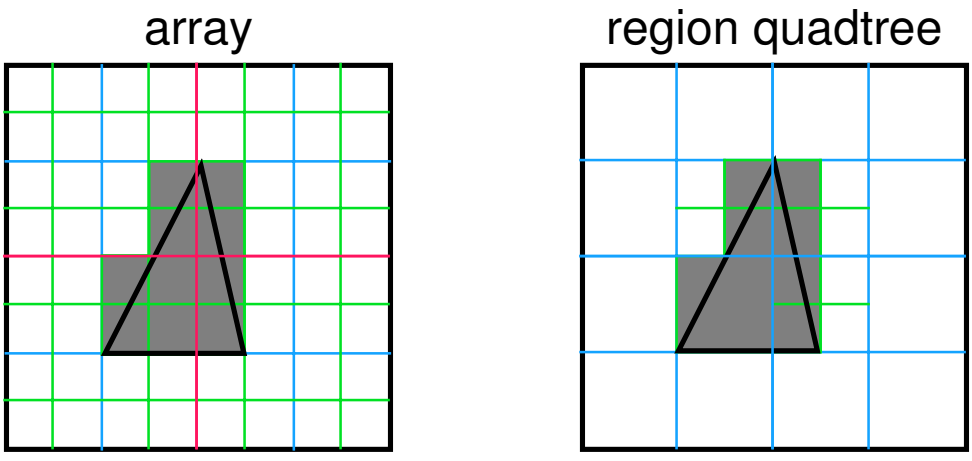
DIMENSION REDUCTION

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.



DIMENSION REDUCTION

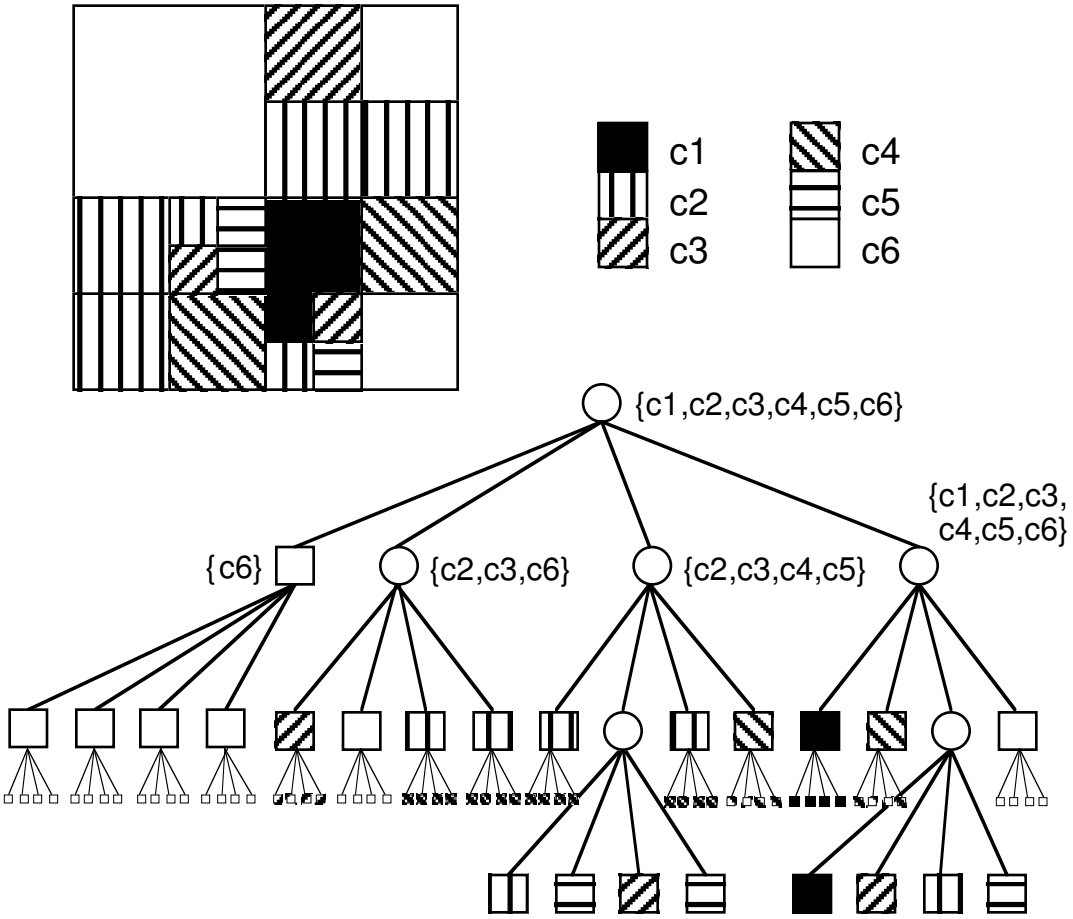
1. Number of blocks necessary to store a simple polygon as a region quadtree is proportional to its perimeter (Hunter)
   - implies that many quadtree algorithms execute in *O*(perimeter) time as they are tree traversals
   - the region quadtree is a dimension reducing device as perimeter (ignoring fractal 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

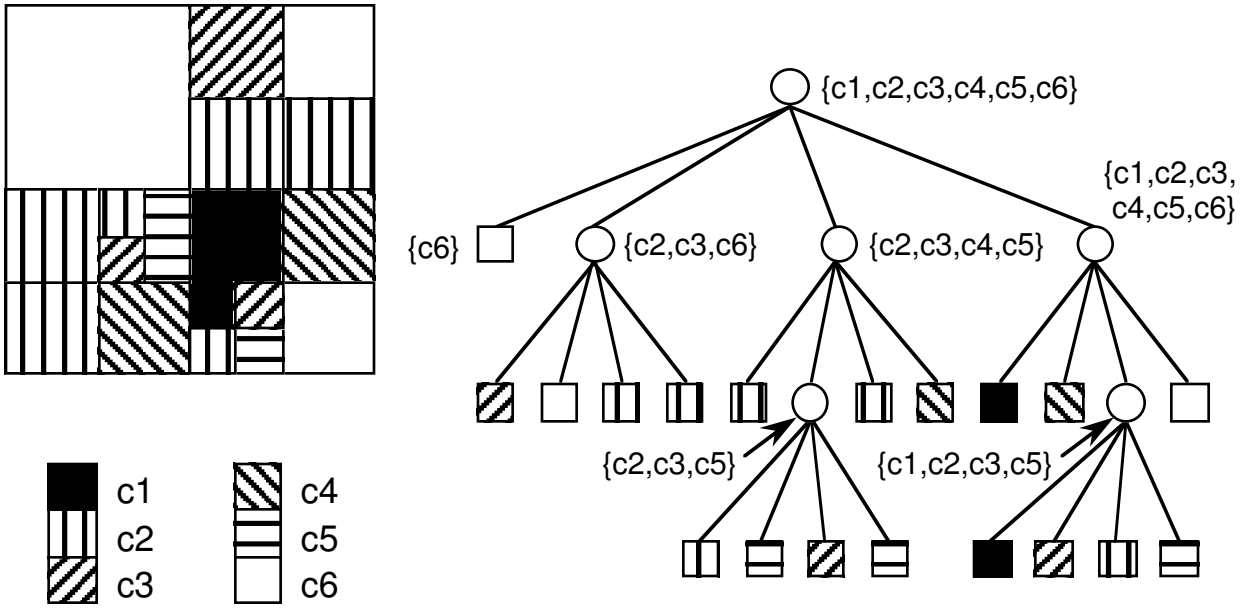
PYRAMID

- 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

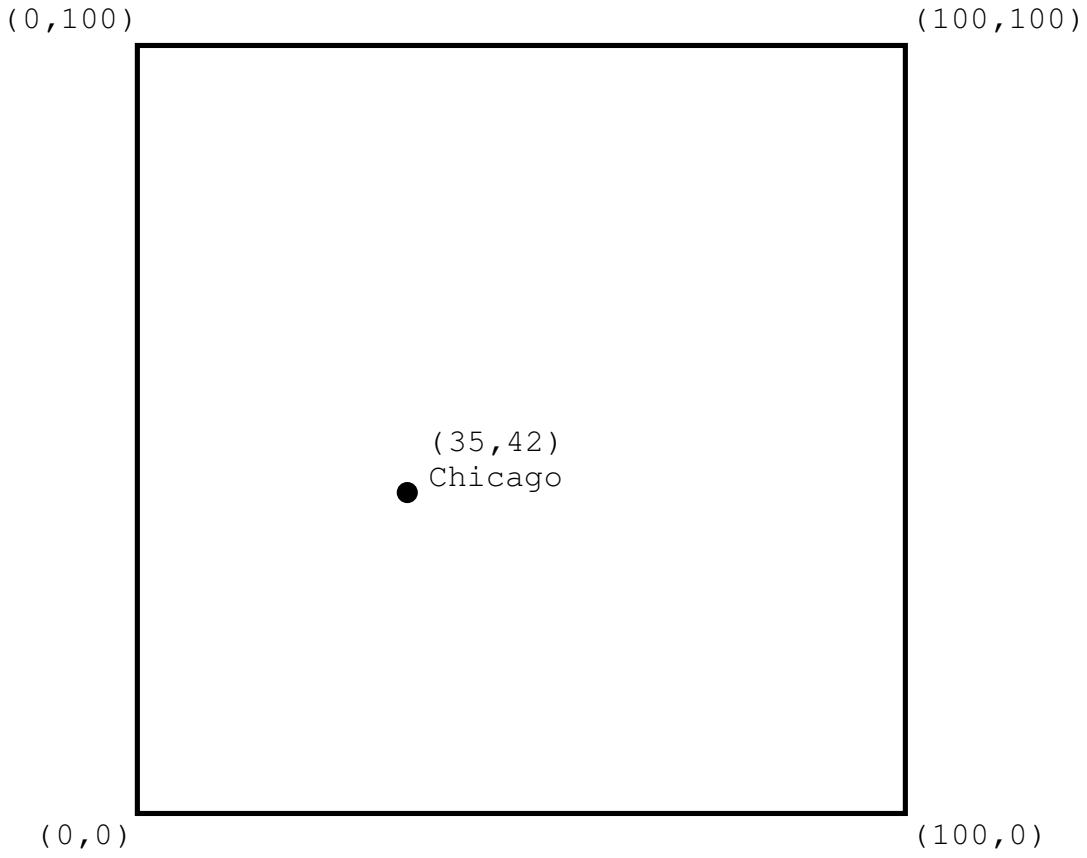


- Can represent as a list of leaf and nonleaf blocks (e.g., as a linear quadtree)

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

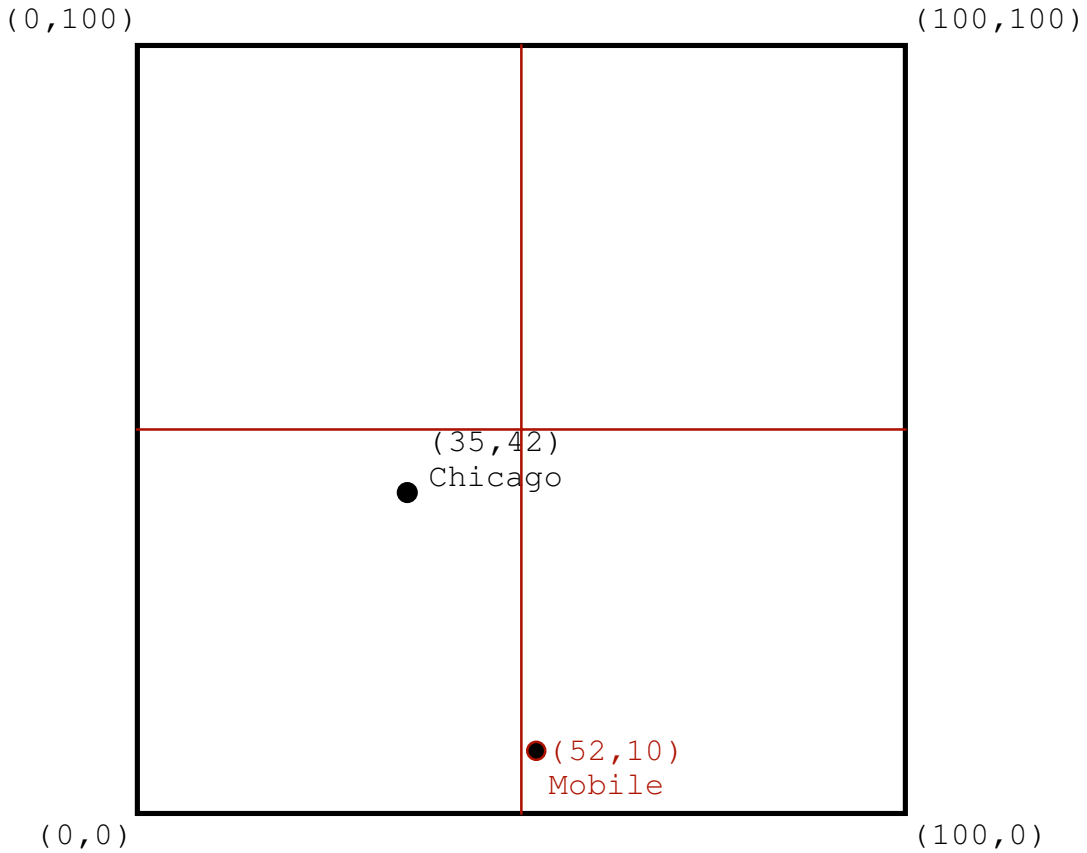
   Ex: $c = 1$



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

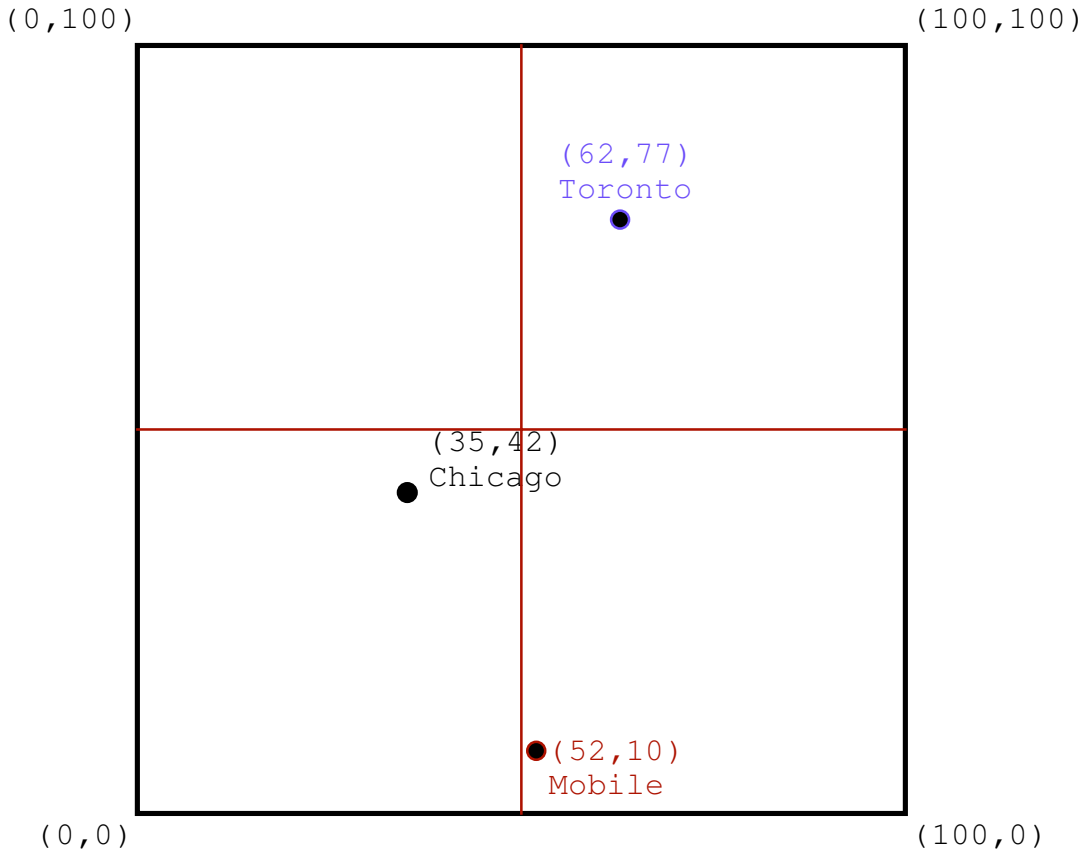
   Ex: *c* = 1



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

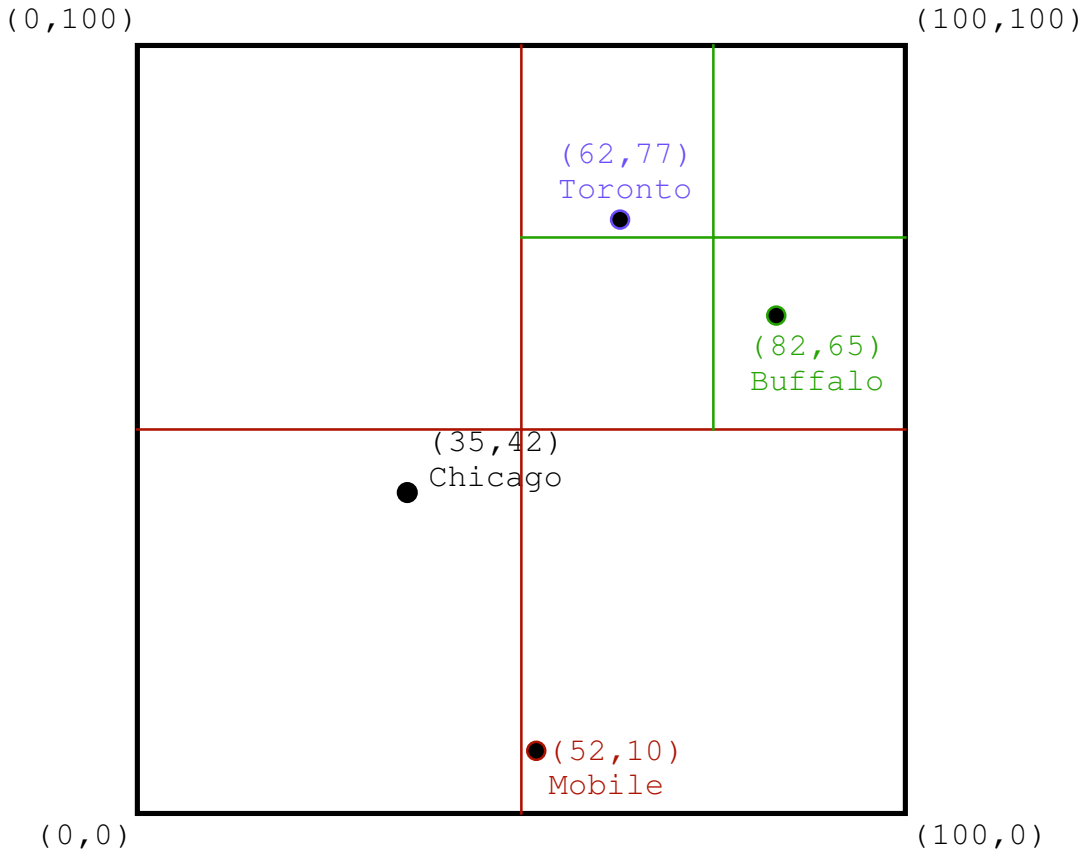
   Ex: $c = 1$



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

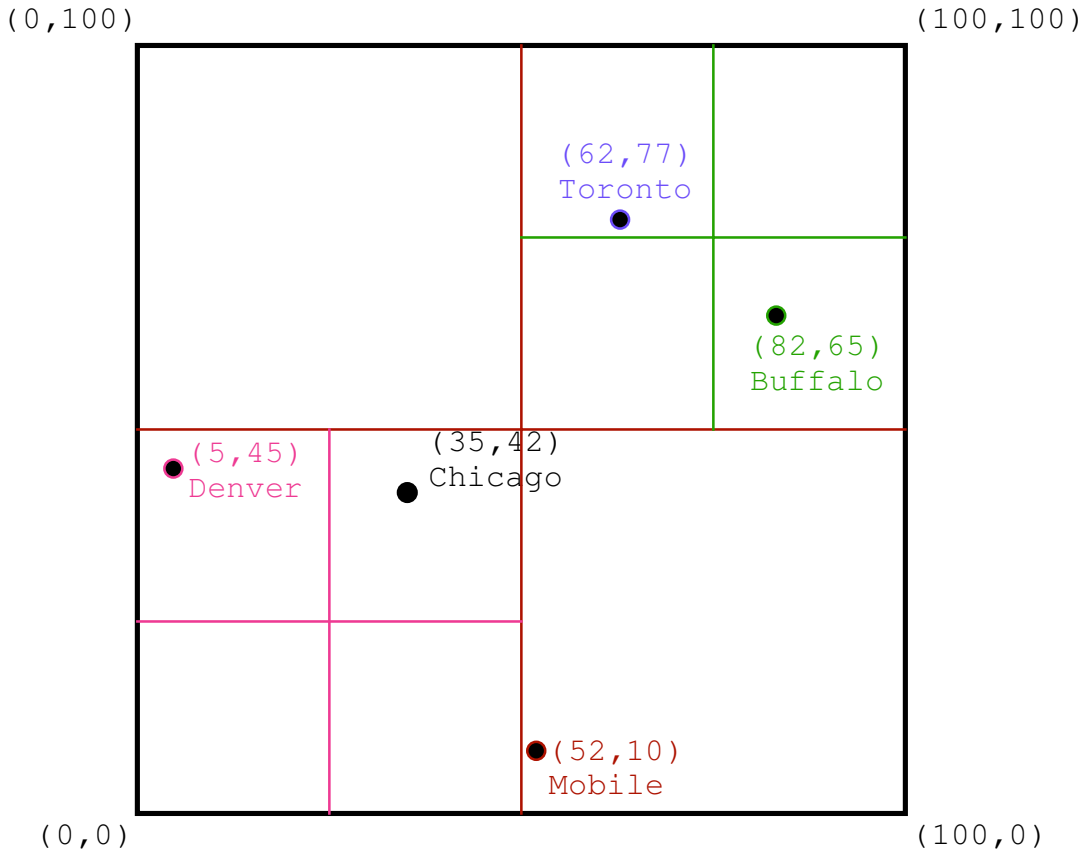
   Ex: $c = 1$



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

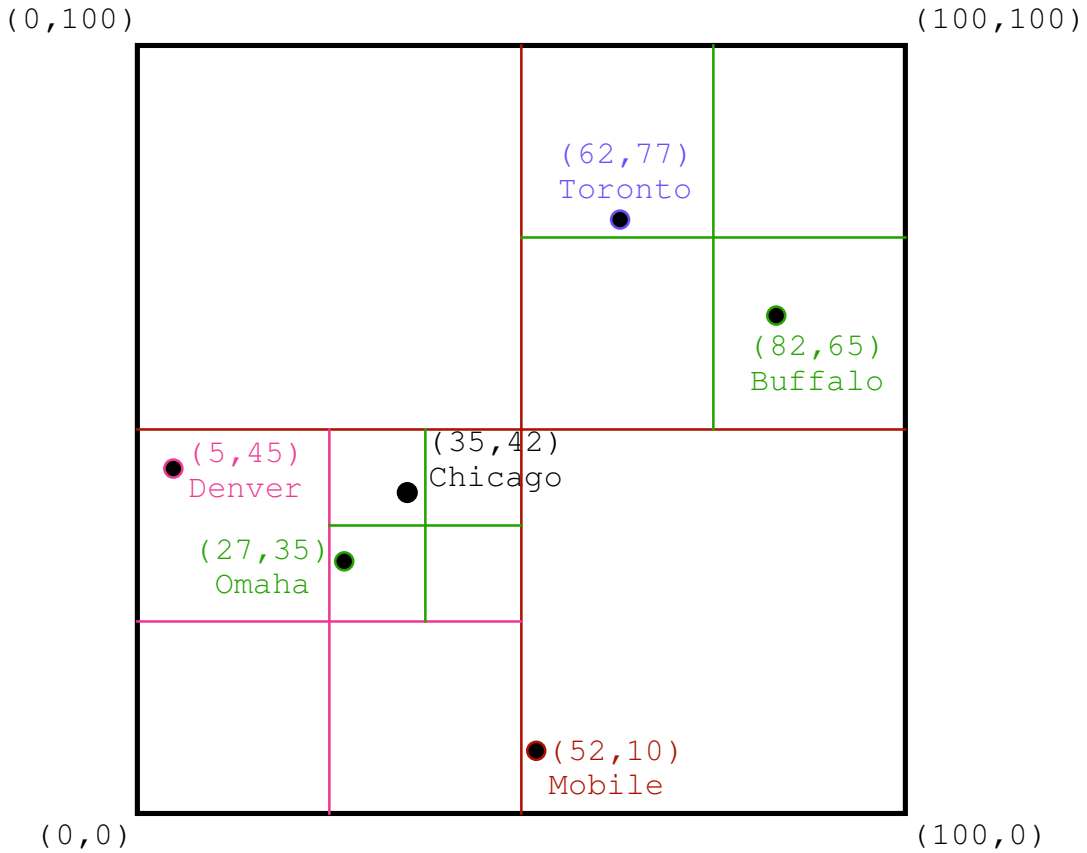
   Ex: *c* = 1



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

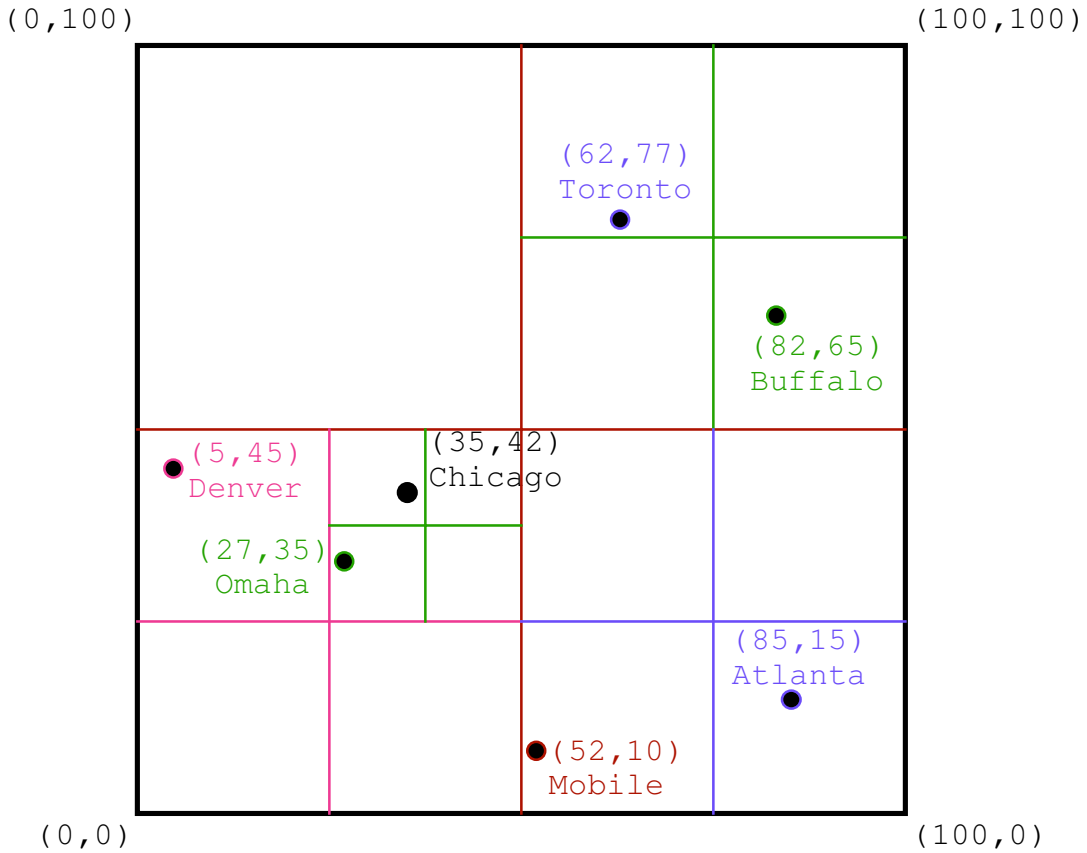
   Ex: *c* = 1



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

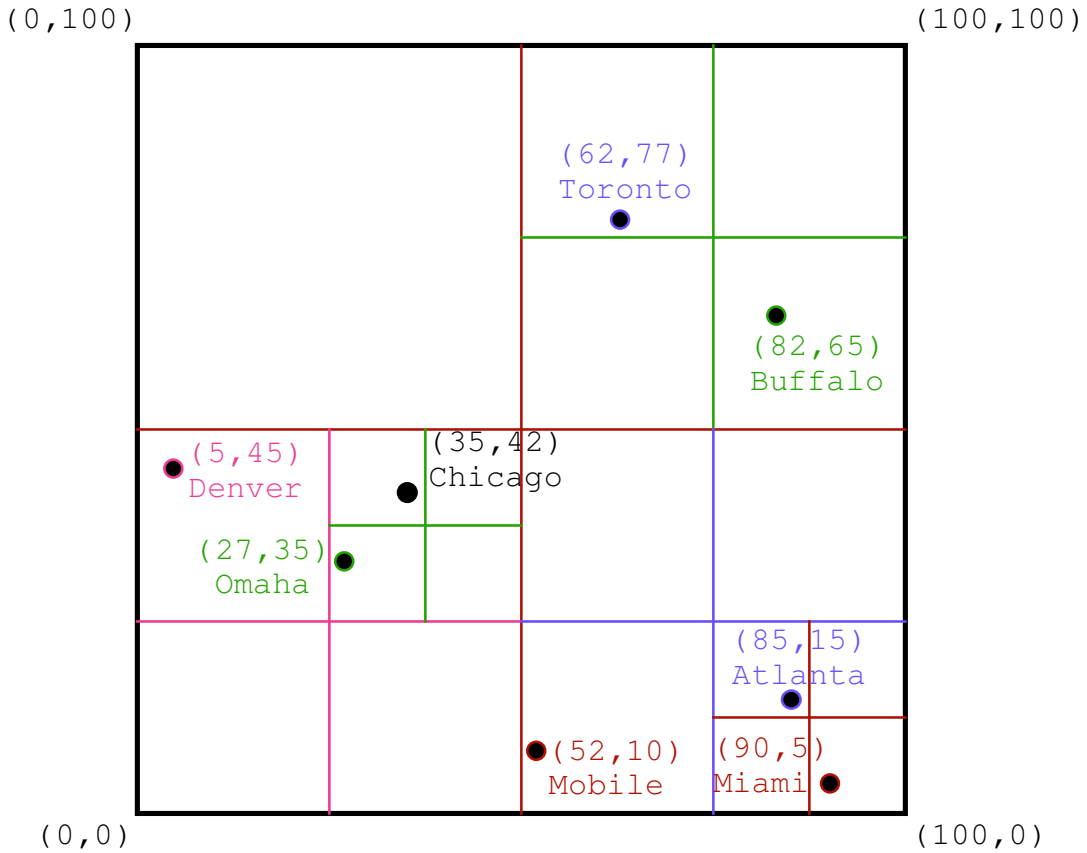
   Ex: *c* = 1



PR QUADTREE (Orenstein)

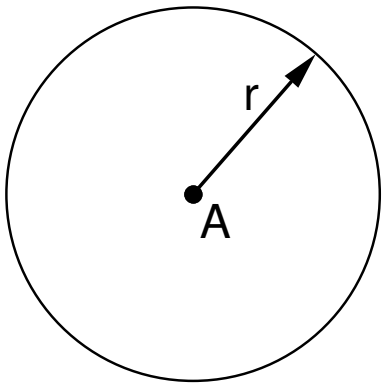
1. Regular decomposition point representation
2. Decomposition occurs whenever a block contains more than one point
3. Useful when the domain of data points is not discrete but finite
4. Maximum level of decomposition depends on the minimum separation between two points
   - if two points are very close, 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



REGION SEARCH

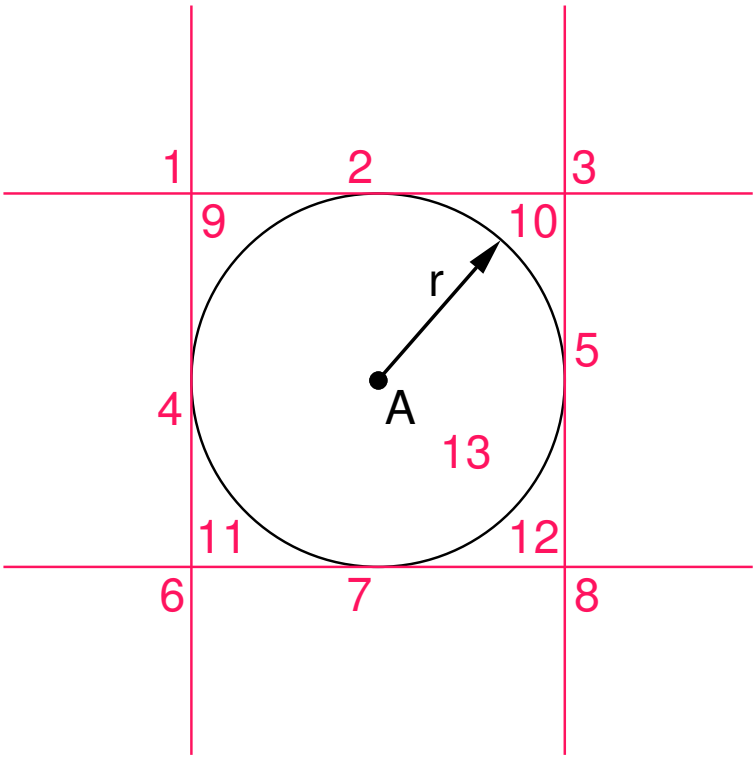
- Ex: Find all points within radius *r* of point A



- Use of quadtree results in pruning the search space

REGION SEARCH

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

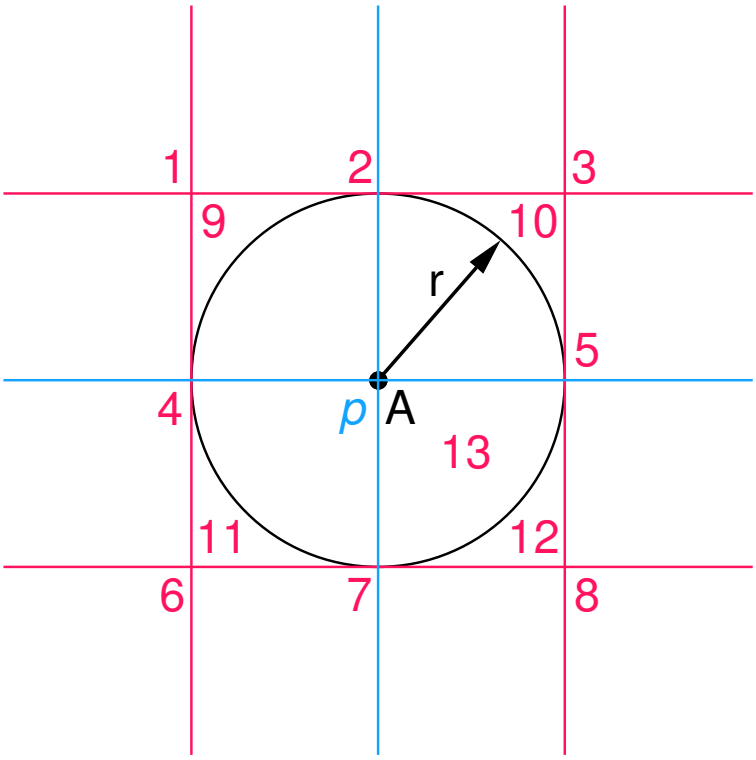


- Use of quadtree results in pruning the search space
- If a quadrant subdivision point *p* lies in a region *l*, then search the quadrants of *p* specified by *l*

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

REGION SEARCH

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

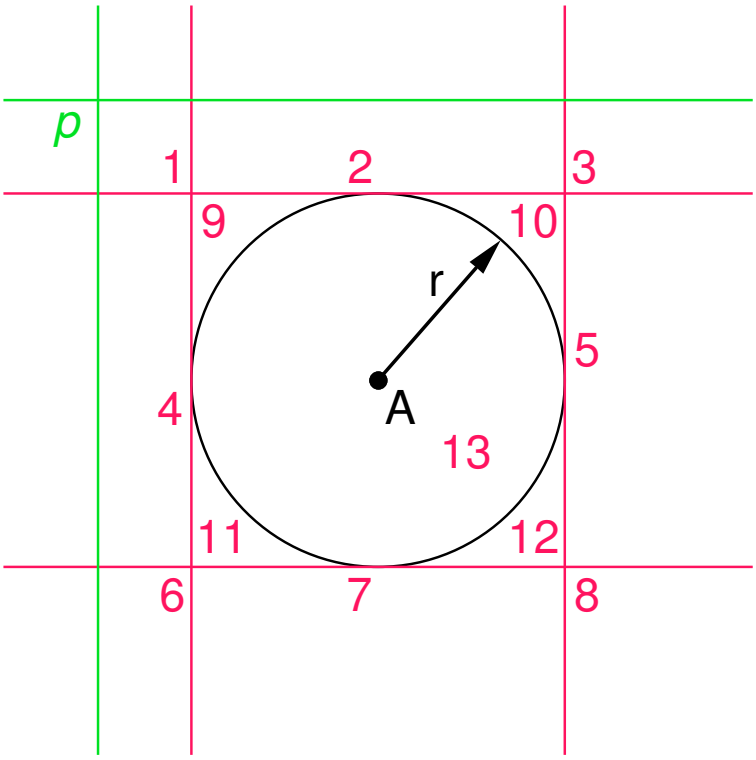


- Use of quadtree results in pruning the search space
- If a quadrant subdivision point *p* lies in a region *l*, then search the quadrants of *p* specified by *l*

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

REGION SEARCH

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

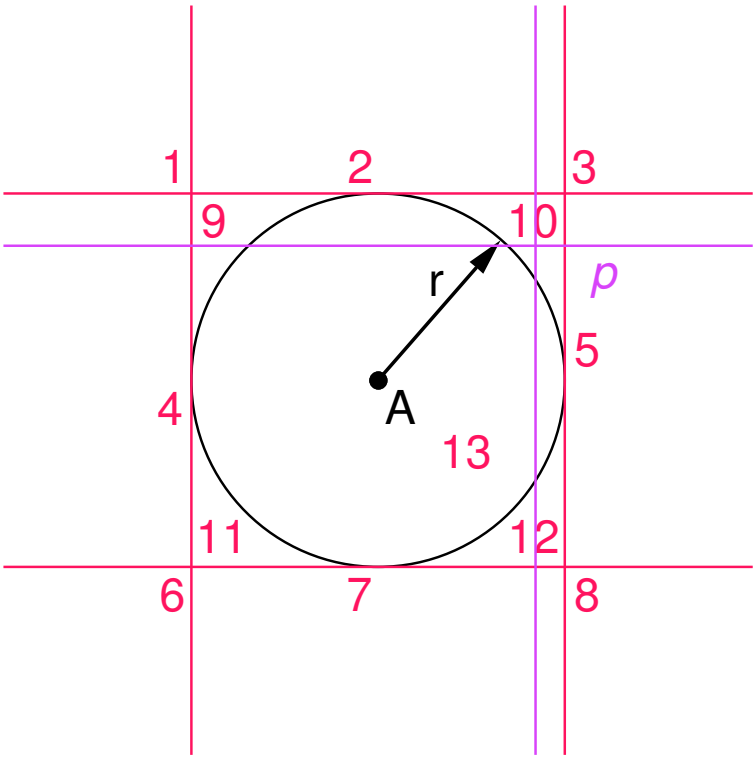


- Use of quadtree results in pruning the search space
- If a quadrant subdivision point *p* lies in a region *l*, then search the quadrants of *p* specified by *l*

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

REGION SEARCH

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

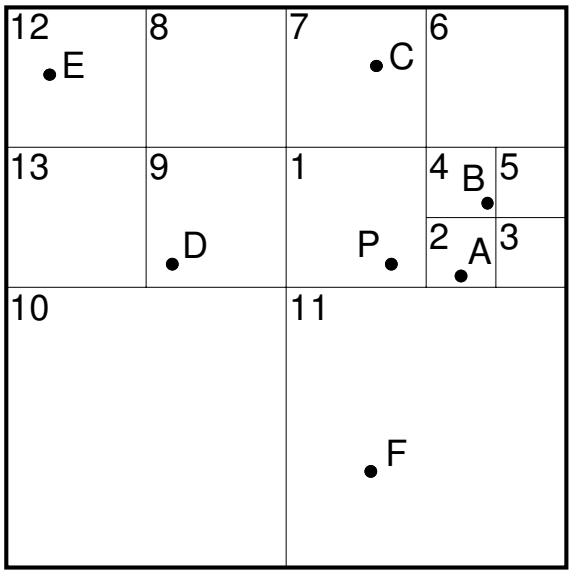


- Use of quadtree results in pruning the search space
- If a quadrant subdivision point *p* lies in a region *l*, then search the quadrants of *p* specified by *l*

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

FINDING THE NEAREST OBJECT

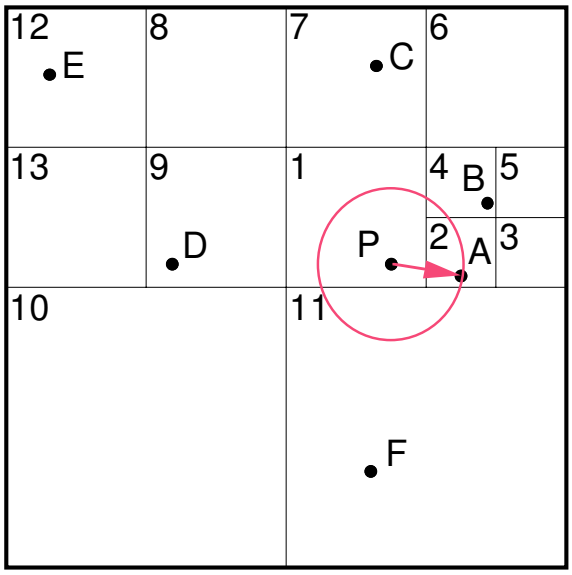
- Ex: find the nearest object to P



- 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:

FINDING THE NEAREST OBJECT

- Ex: find the nearest object to P

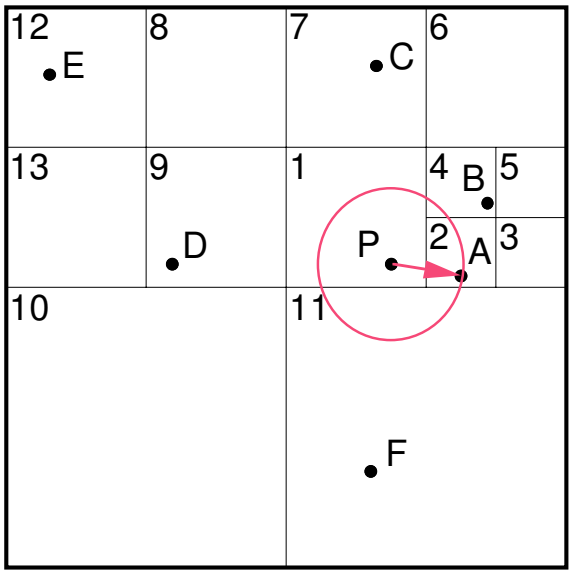


- Assume PR quadtree for points (i.e., at most one point per block)
- Search neighbors of block 1 in counterclockwise order
- Points are sorted with respect to the space they occupy which enables pruning the search space
- Algorithm:

  1. start at block 2 and compute distance to P from A

FINDING THE NEAREST OBJECT

- Ex: find the nearest object to P

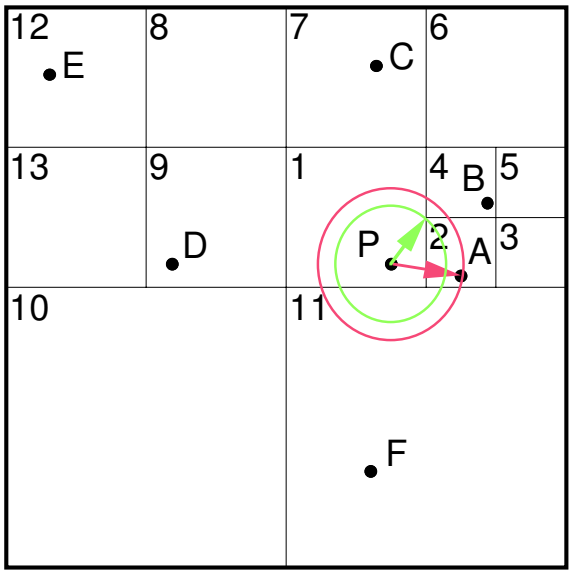


- Assume PR quadtree for points (i.e., at most one point per block)
- Search neighbors of block 1 in counterclockwise order
- Points are sorted with respect to the space they occupy which enables pruning the search space
- Algorithm:

  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

FINDING THE NEAREST OBJECT

- Ex: find the nearest object to P

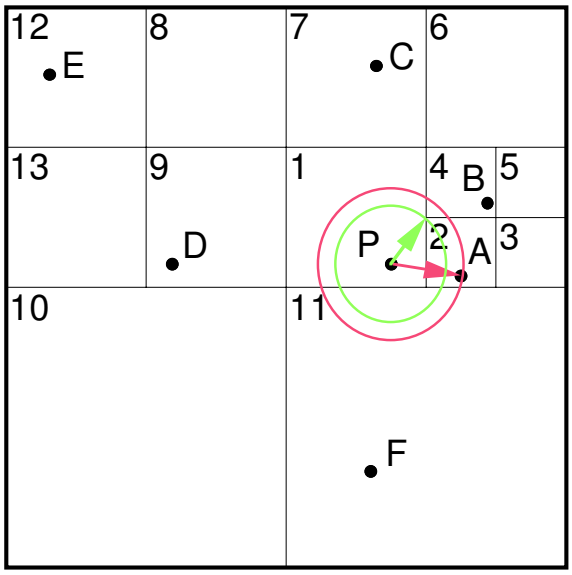


- Assume PR quadtree for points (i.e., at most one point per block)
- Search neighbors of block 1 in counterclockwise order
- Points are sorted with respect to the space they occupy which enables pruning the search space
- Algorithm:

  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

FINDING THE NEAREST OBJECT

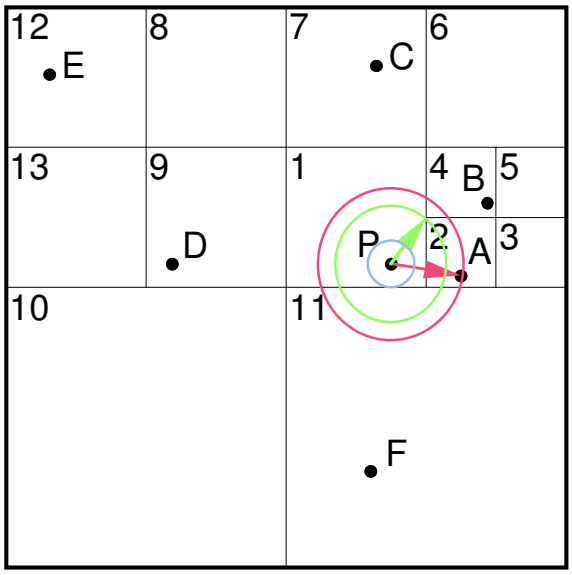
- Ex: find the nearest object to P



- 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

FINDING THE NEAREST OBJECT

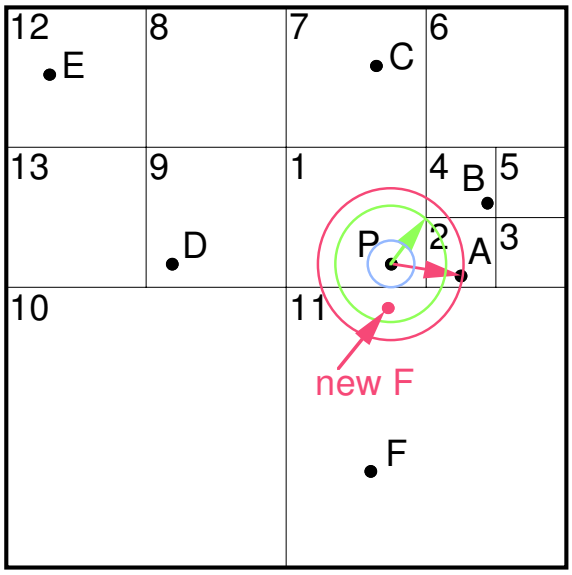
- Ex: find the nearest object to P



- 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

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

INCREMENTAL NEAREST NEIGHBORS (HJATASON/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:

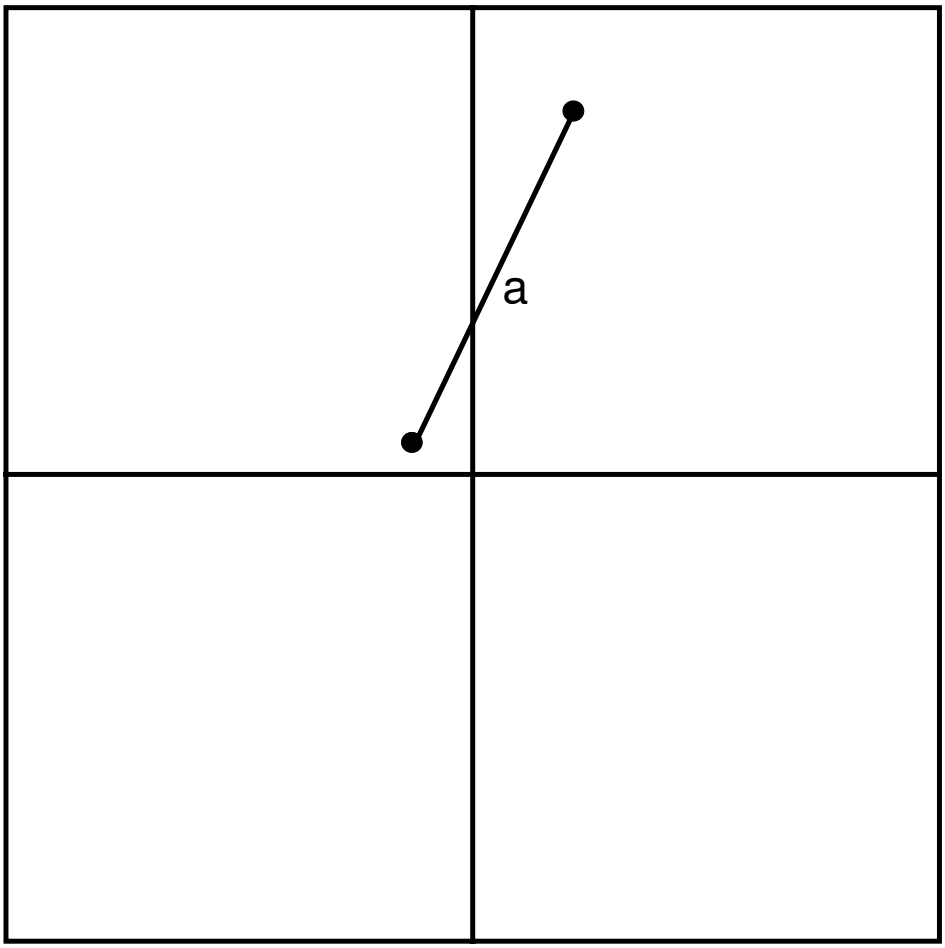
INCNEAREST(q, S, T)

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, $e_{t'}$, $d_{t'}(q, e_{t'})$)

3. Lines 1-3 initialize priority queue with root
4. 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

PM1 QUADTREE

- Vertex-based (one vertex per block)



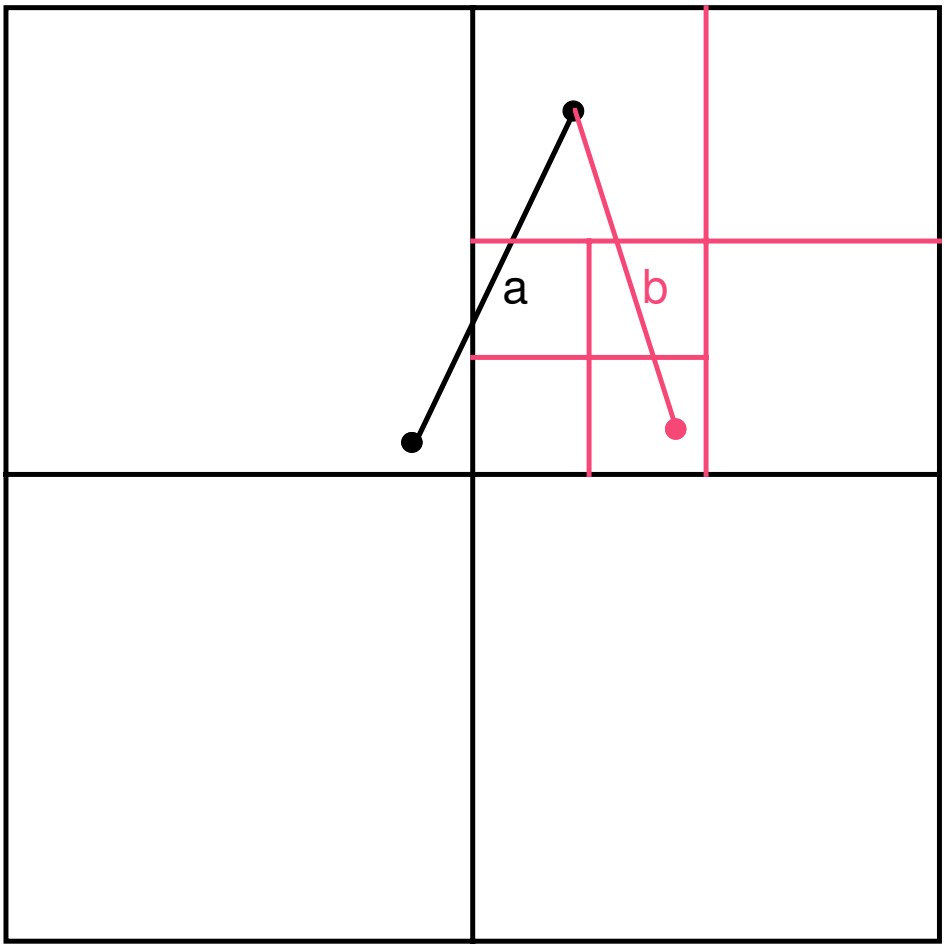
DECOMPOSITION RULE:

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

PM1 QUADTREE

- Vertex-based (one vertex per block)



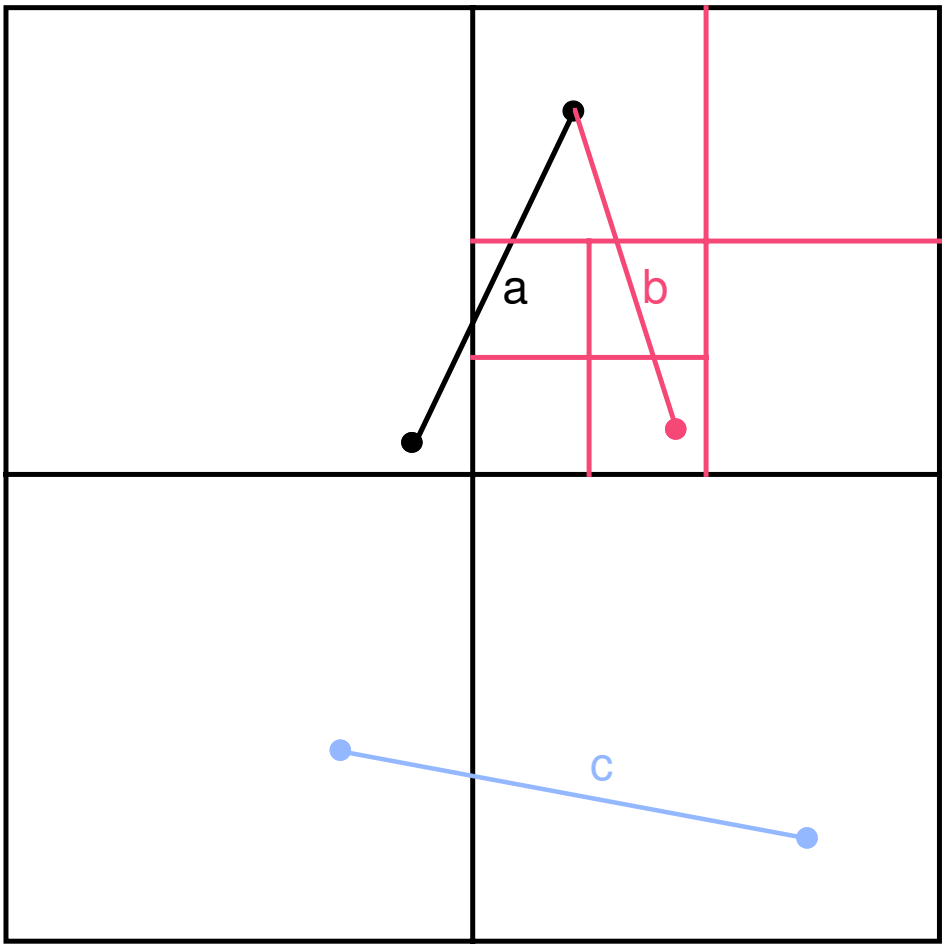
DECOMPOSITION RULE:

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

PM1 QUADTREE

- Vertex-based (one vertex per block)



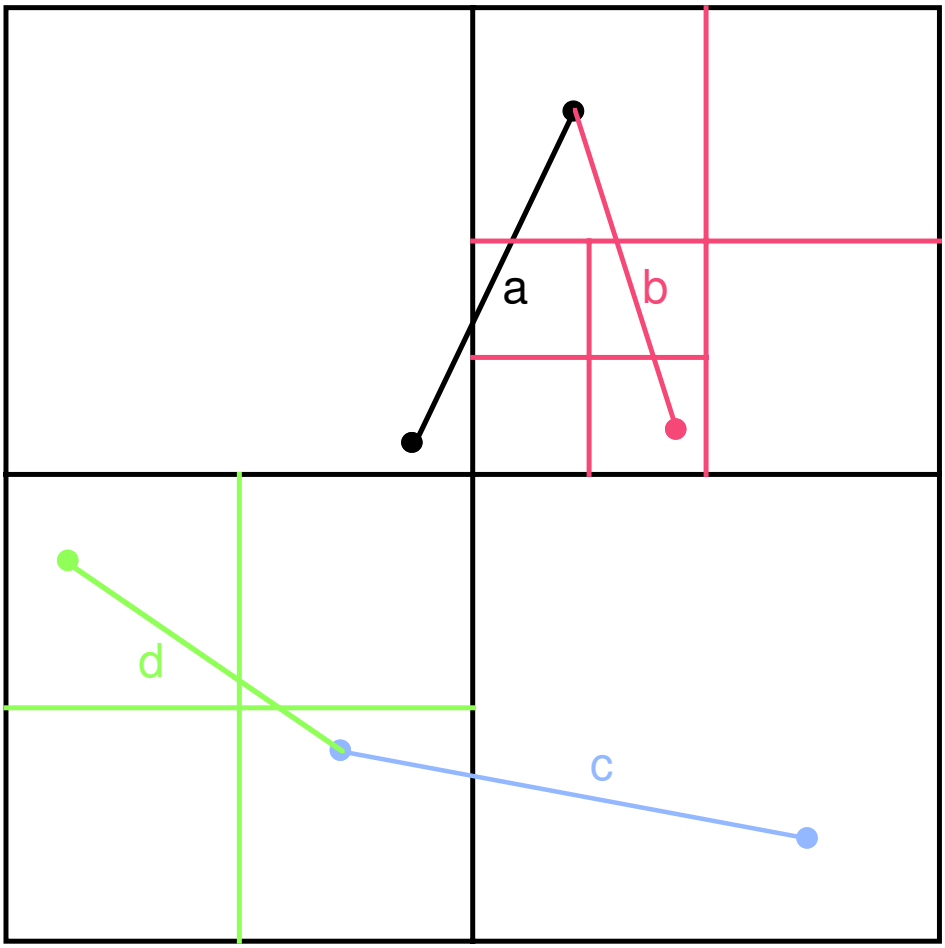
DECOMPOSITION RULE:

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

PM1 QUADTREE

- Vertex-based (one vertex per block)



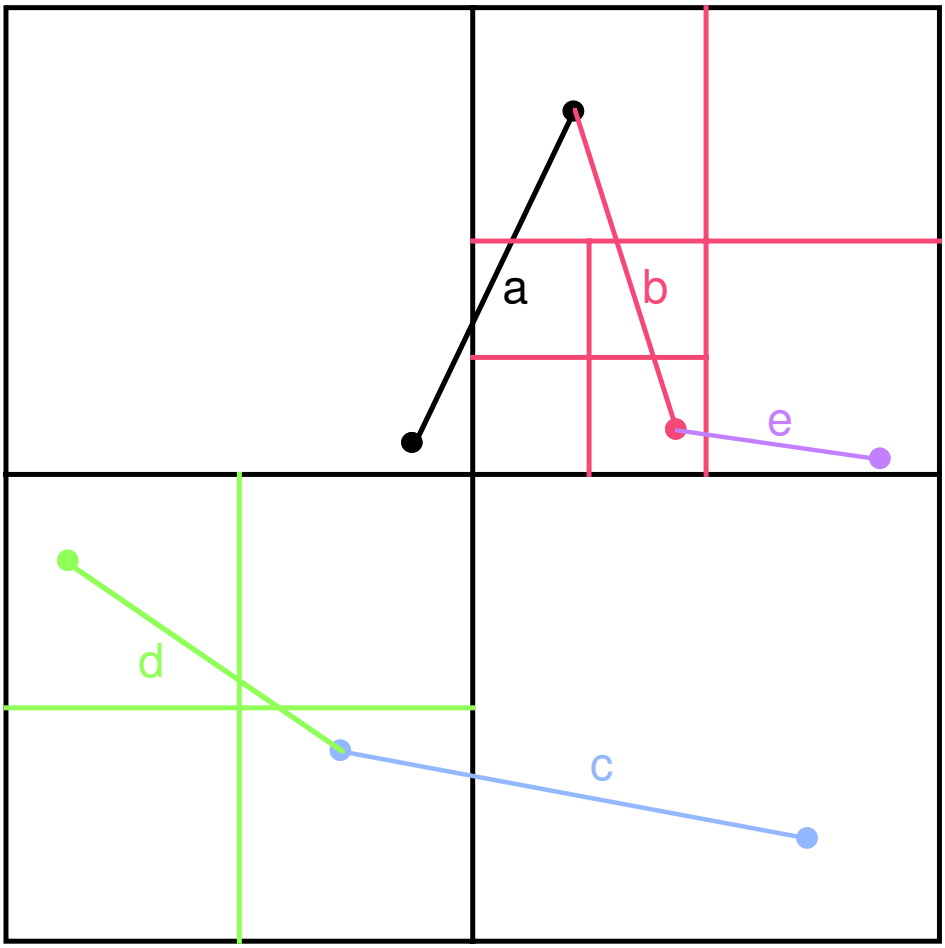
DECOMPOSITION RULE:

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

PM1 QUADTREE

- Vertex-based (one vertex per block)



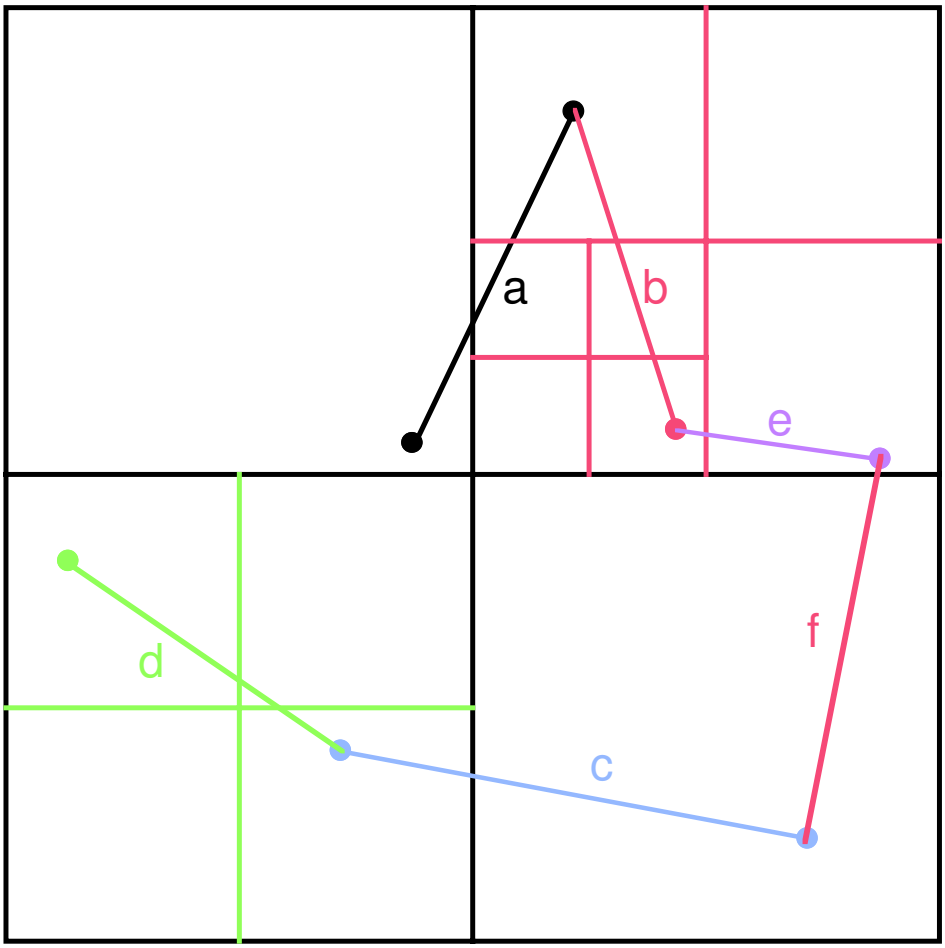
DECOMPOSITION RULE:

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

PM1 QUADTREE

- Vertex-based (one vertex per block)



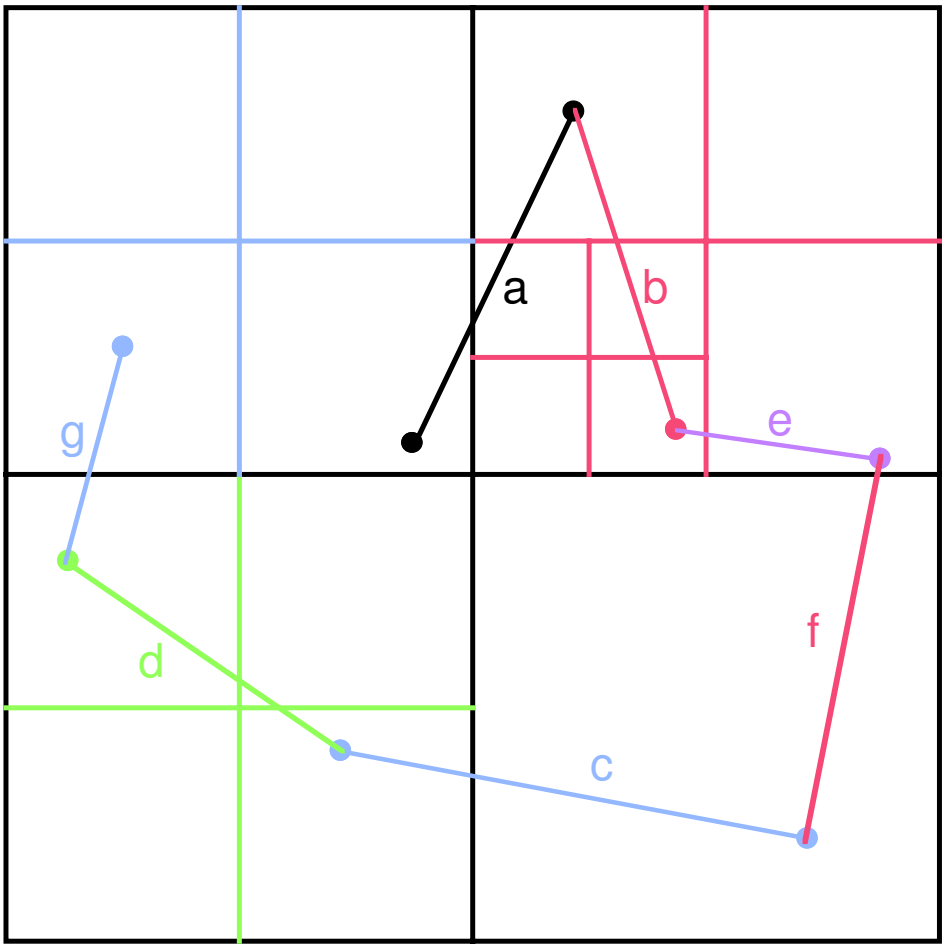
DECOMPOSITION RULE:

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

PM1 QUADTREE

- Vertex-based (one vertex per block)



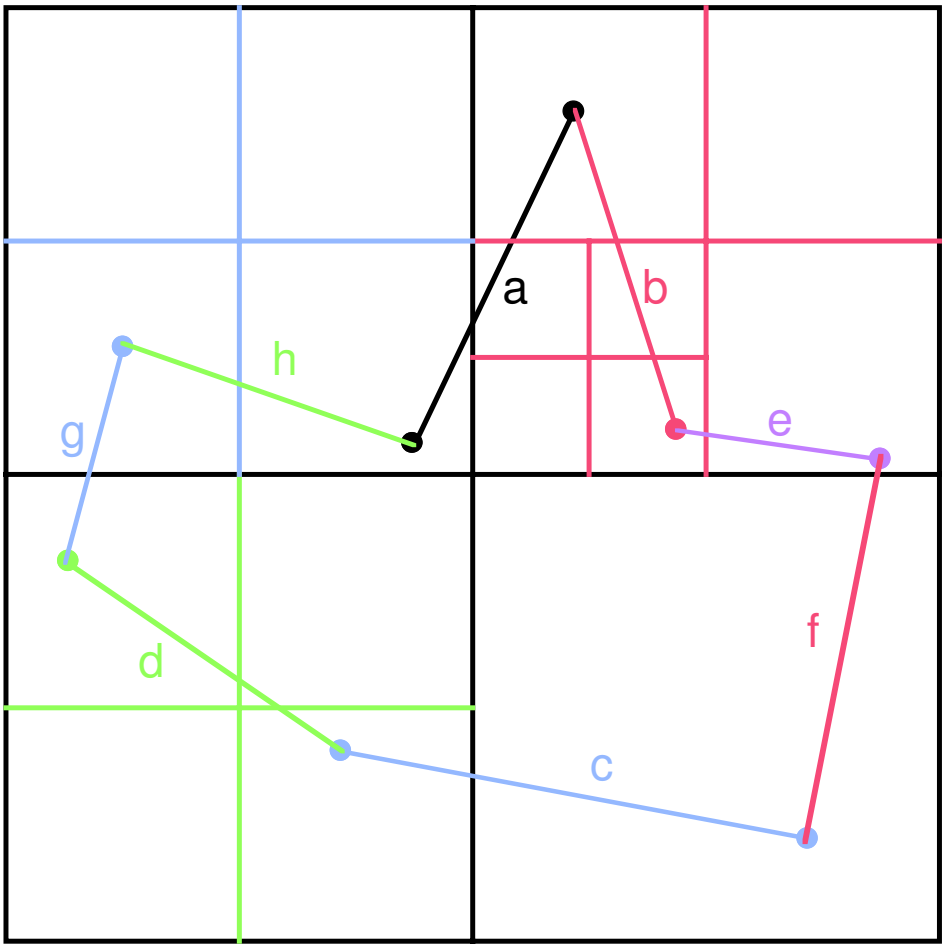
DECOMPOSITION RULE:

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

PM1 QUADTREE

- Vertex-based (one vertex per block)



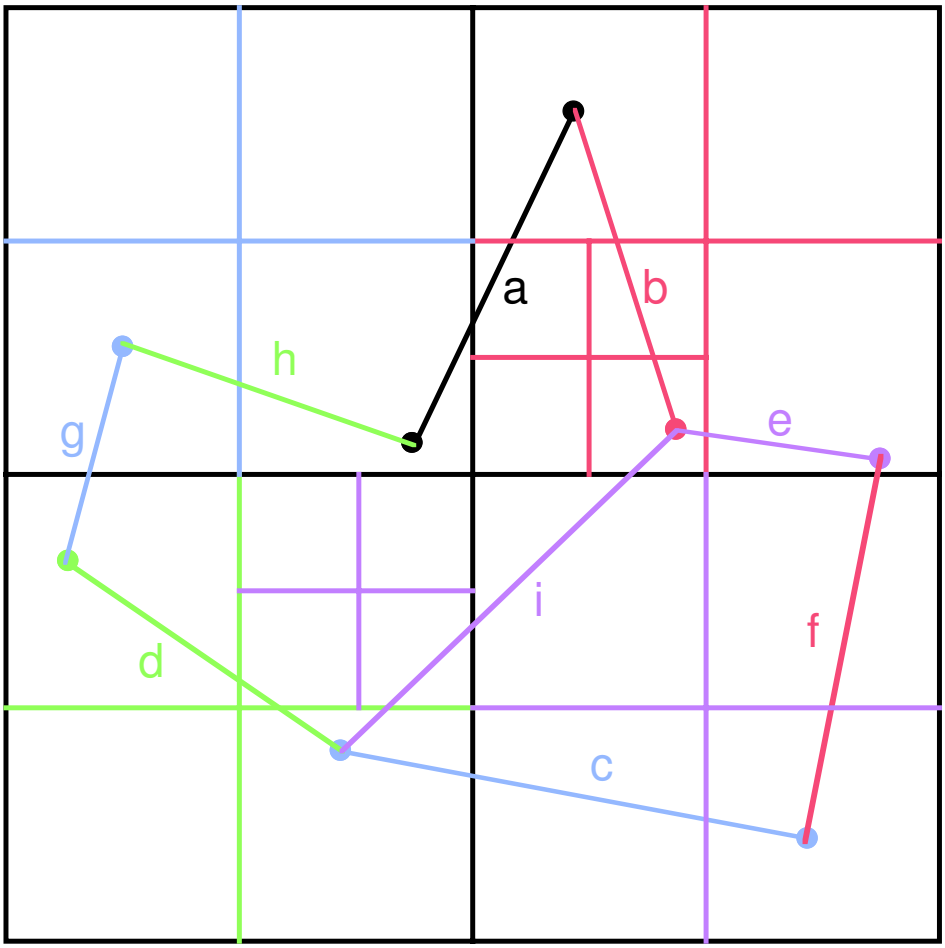
DECOMPOSITION RULE:

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

PM1 QUADTREE

- Vertex-based (one vertex per block)



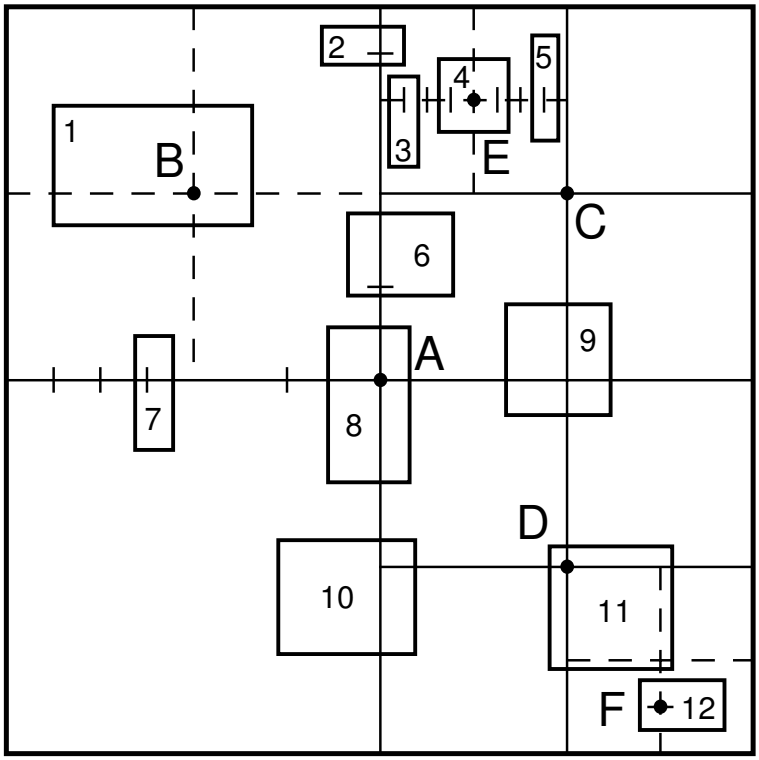
DECOMPOSITION RULE:

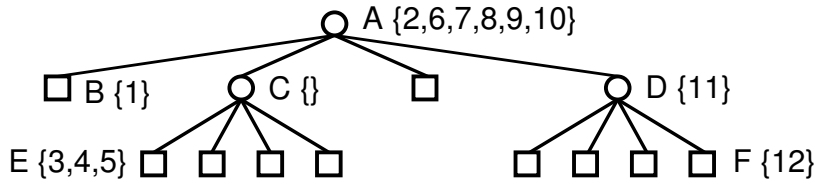
Partitioning occurs when a block contains more than one segment unless all the segments are incident at the same vertex which is also in the same block

- Shape independent of order of insertion

MX-CIF QUADTREE (Kedem)

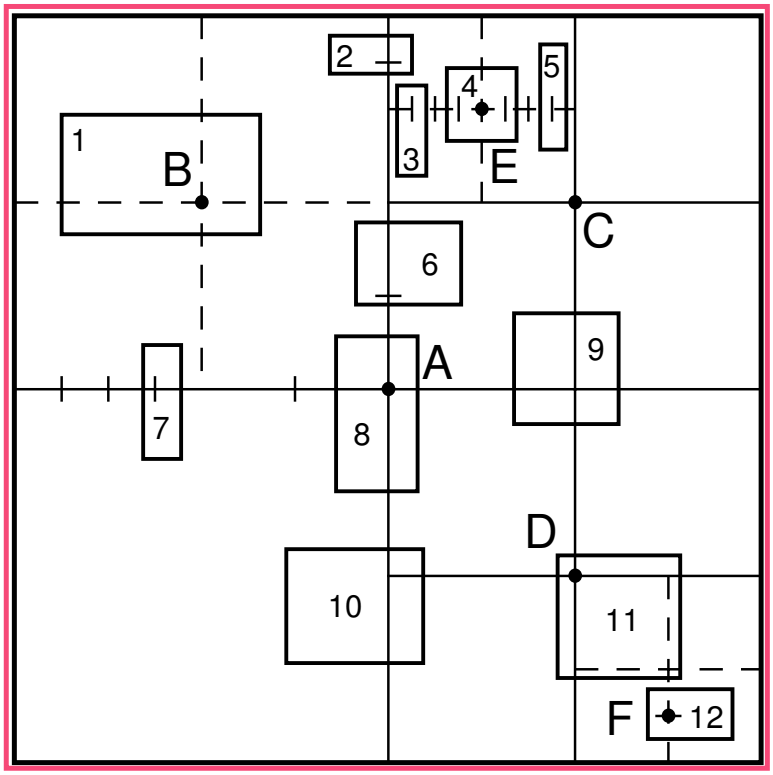
1. Collections of small rectangles for VLSI applications
2. Each rectangle is associated with its minimum enclosing quadtree block
3. Like hashing: quadtree blocks serve as hash buckets

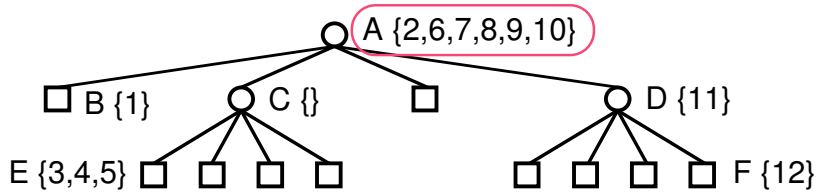




MX-CIF QUADTREE (Kedem)

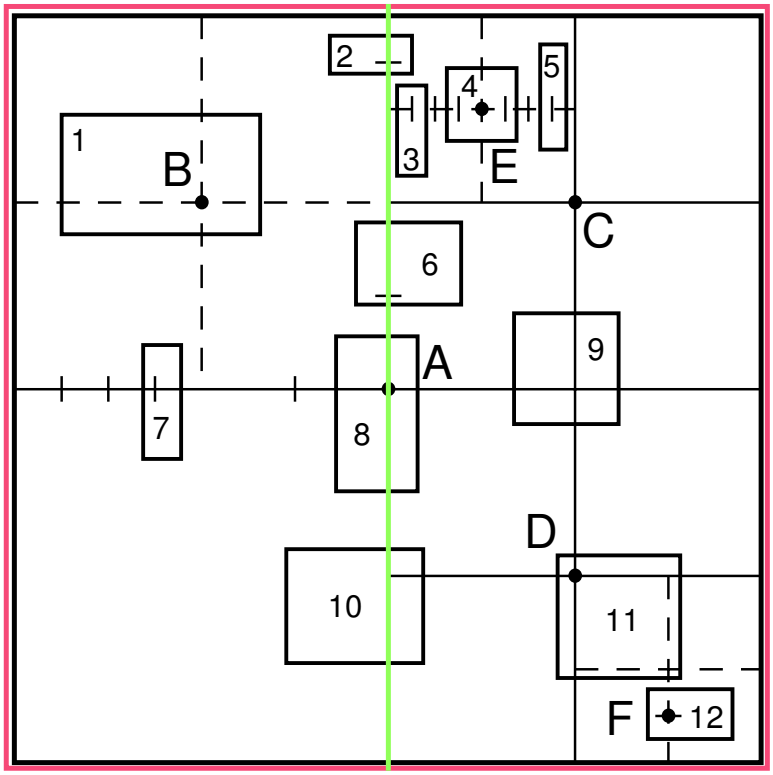
1. Collections of small rectangles for VLSI applications
2. Each rectangle is associated with its minimum enclosing quadtree block
3. Like hashing: quadtree blocks serve as hash buckets
4. Collision = more than one rectangle in a block
   - resolve by using two one-dimensional MX-CIF trees to store the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



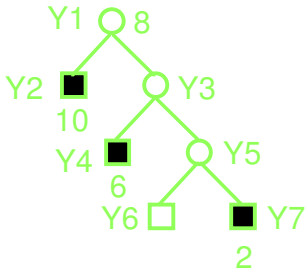


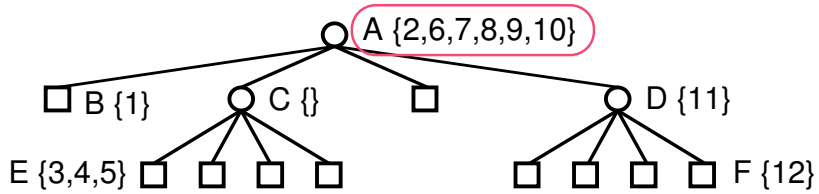
MX-CIF QUADTREE (Kedem)

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



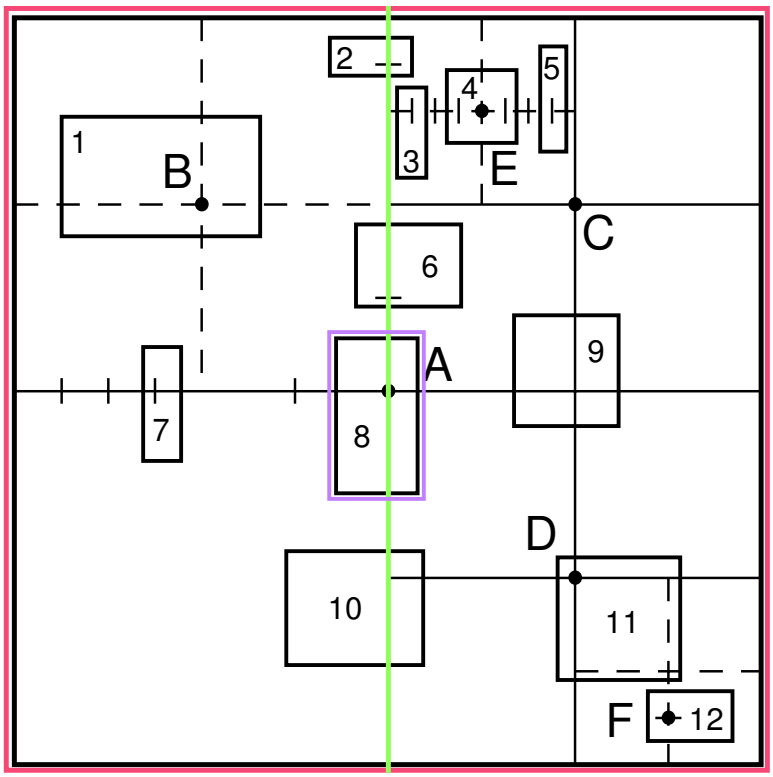
Binary tree for *y*-axis through A



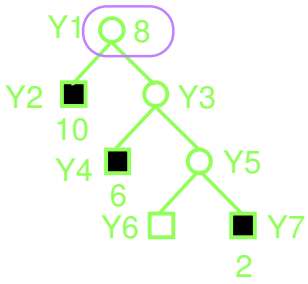


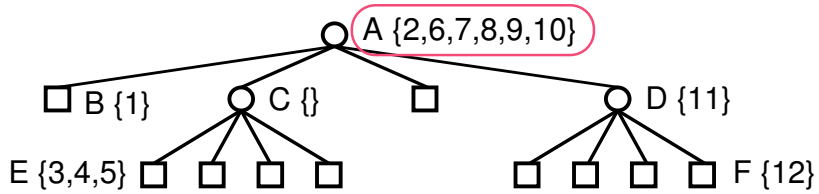
MX-CIF QUADTREE (Kedem)

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



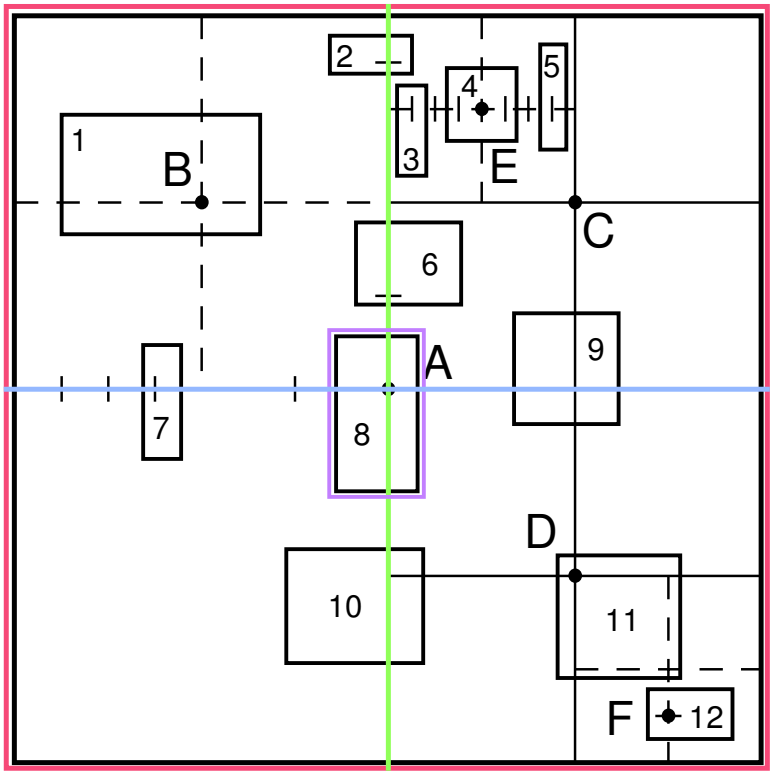
Binary tree for *y*-axis through A



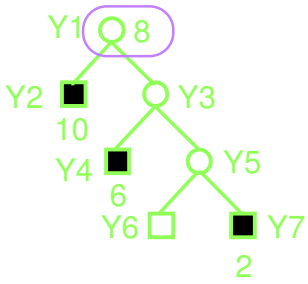


MX-CIF QUADTREE (Kedem)

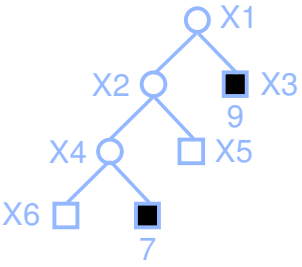
1. Collections of small rectangles for VLSI applications
2. Each rectangle is associated with its minimum enclosing quadtree block
3. Like hashing: quadtree blocks serve as hash buckets
4. Collision = more than one rectangle in a block
   - resolve by using two one-dimensional MX-CIF trees to store the 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

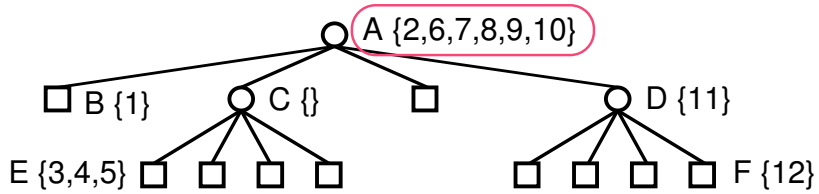


Binary tree for *y*-axis through A



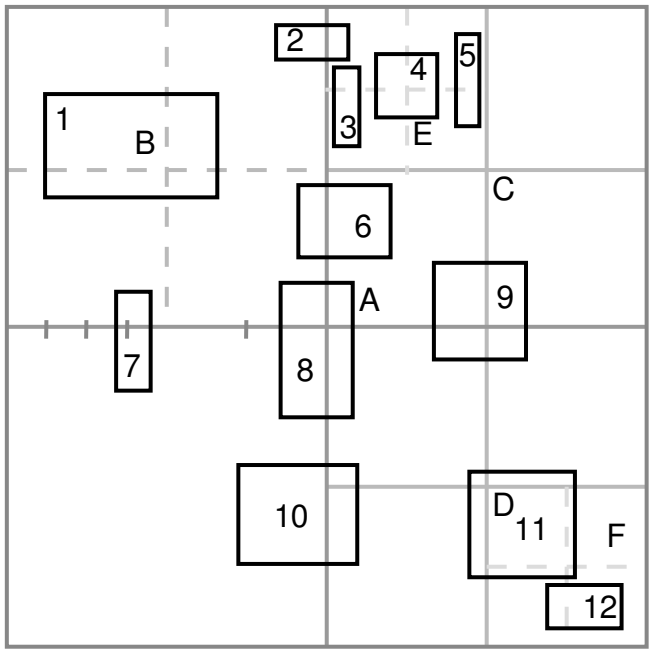
Binary tree for *x*-axis through A

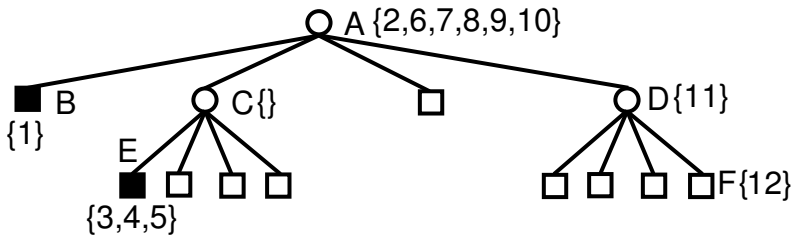




Loose Quadtree (Octree)/Cover Fieldtree

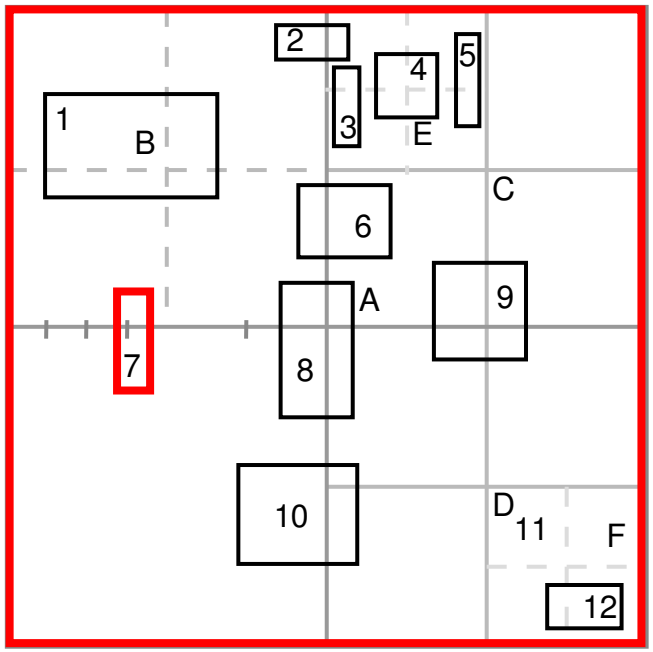
- Overcomes drawback of MX-CIF quadtree that the width w of the minimum enclosing quadtree block of a rectangle o is not a function of the size of o

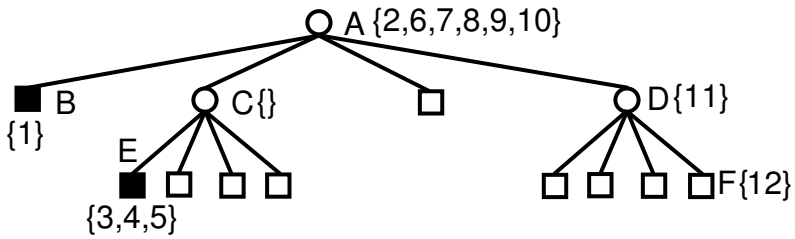




Loose Quadtree (Octree)/Cover Fieldtree

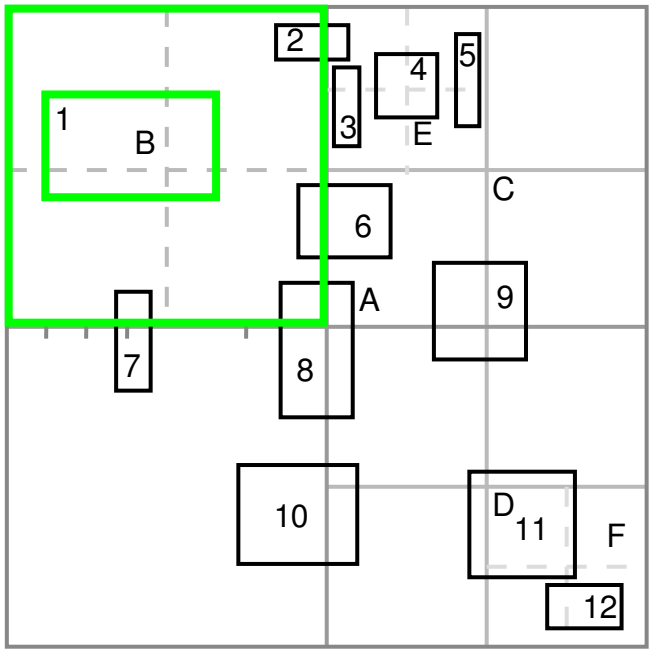
- Overcomes drawback of MX-CIF quadtree that the width w of the minimum enclosing quadtree block of a rectangle o is not a function of the size of o
- Instead, it depends on the position of the centroid of o and often considerably larger than o

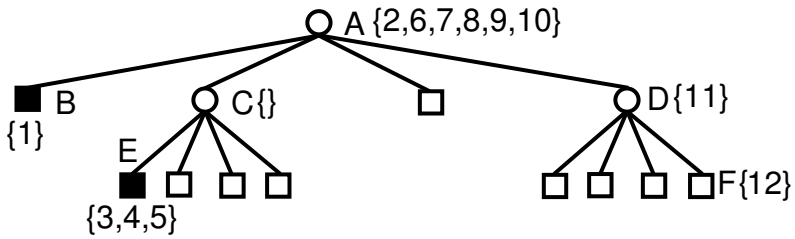




Loose Quadtree (Octree)/Cover Fieldtree

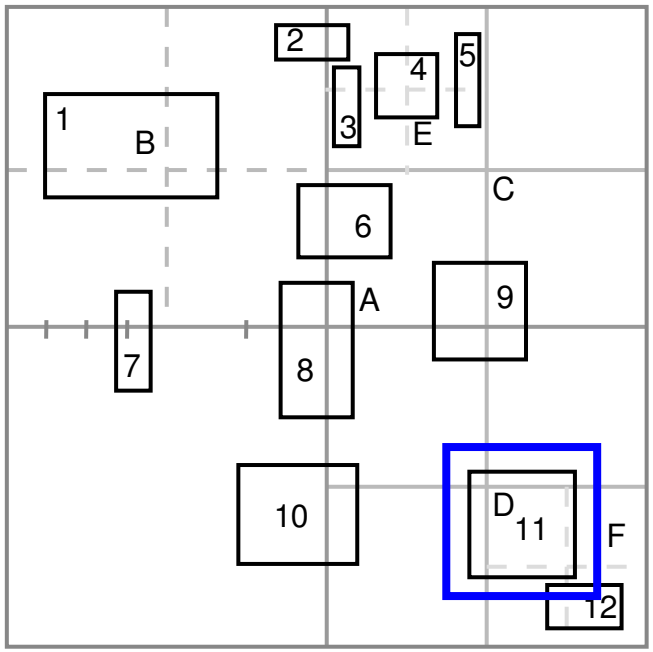
- Overcomes drawback of MX-CIF quadtree that the width w of the minimum enclosing quadtree block of a rectangle o is not a function of the size of o
- Instead, it depends on the position of the centroid of o and often considerably larger than o

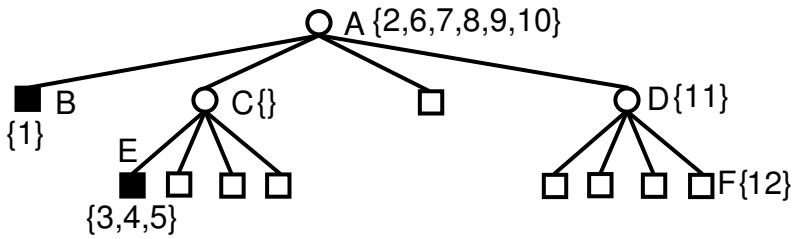




Loose Quadtree (Octree)/Cover Fieldtree

- Overcomes drawback of MX-CIF quadtree that the width w of the minimum enclosing quadtree block of a rectangle o is not a function of the size of o
- Instead, it depends on the position of the centroid of o and often considerably larger than o
- Solution: expand size of space spanned by each quadtree block of width w by expansion factor p ($p > 0$) so expanded block is of width $(1 + p)w$

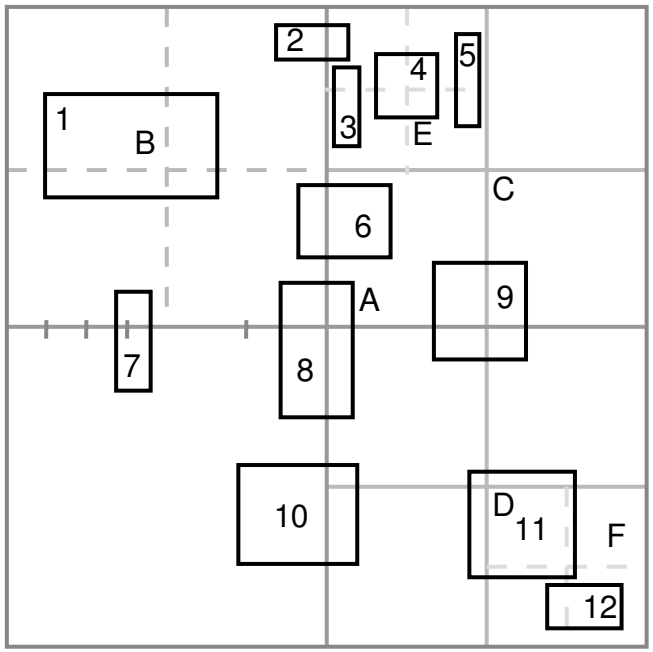


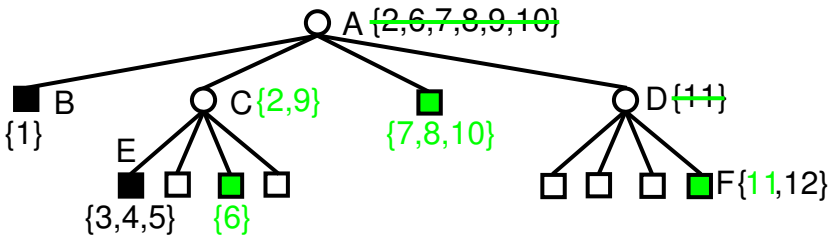


Loose Quadtree (Octree)/Cover Fieldtree

- Overcomes drawback of MX-CIF quadtree that the width w of the minimum enclosing quadtree block of a rectangle o is not a function of the size of o
- Instead, it depends on the position of the centroid of o and often considerably larger than o
- Solution: expand size of space spanned by each quadtree block of width w by expansion factor p ($p > 0$) so expanded block is of width $(1 + p)w$

  1. $p = 0.3$

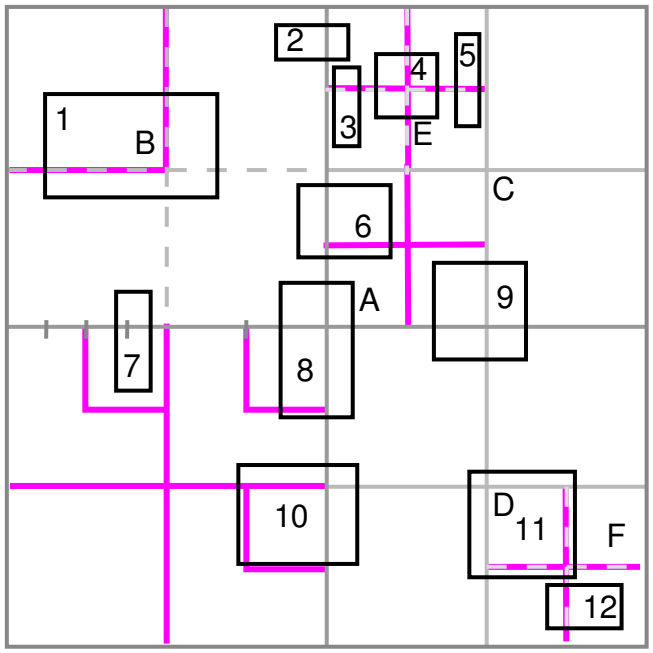


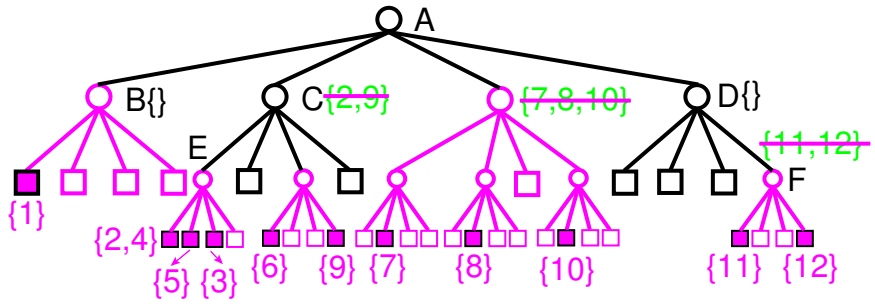


Loose Quadtree (Octree)/Cover Fieldtree

- Overcomes drawback of MX-CIF quadtree that the width w of the minimum enclosing quadtree block of a rectangle o is not a function of the size of o
- Instead, it depends on the position of the centroid of o and often considerably larger than o
- Solution: expand size of space spanned by each quadtree block of width w by expansion factor p ($p > 0$) so expanded block is of width $(1 + p)w$

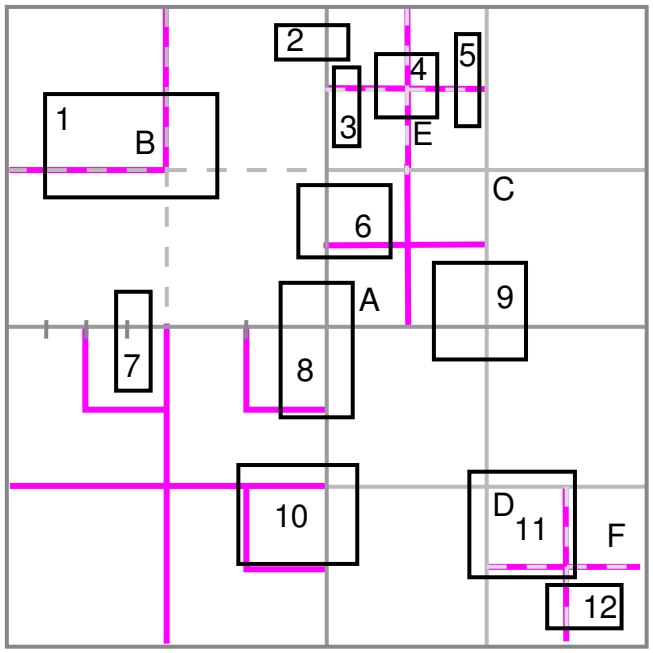
  1. $p = 0.3$
  2. $p = 1.0$

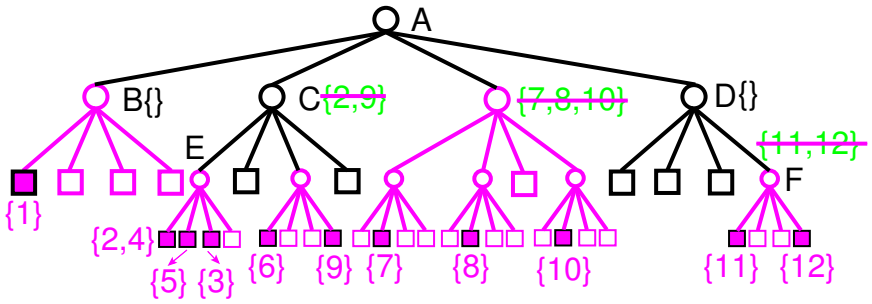




Loose Quadtree (Octree)/Cover Fieldtree

- Overcomes drawback of MX-CIF quadtree that the width w of the minimum enclosing quadtree block of a rectangle o is not a function of the size of o
- Instead, it depends on the position of the centroid of o and often considerably larger than o
- Solution: expand size of space spanned by each quadtree block of width w by expansion factor p ($p > 0$) so expanded block is of width $(1+p)w$
  1. $p = 0.3$
  2. $p = 1.0$
- Maximum w (i.e., minimum depth of minimum enclosing quadtree block) is a function of p and radius r of o and independent of position of centroid of o
  1. Range of possible ratios $w/2r$: $1/(1+p) \cdot w/2r < 2/p$
  2. For $p \geq 1$, restricting w and r to powers of 2, $w/2r$ takes on at most 2 values and usually just 1





Partition Fieldtree

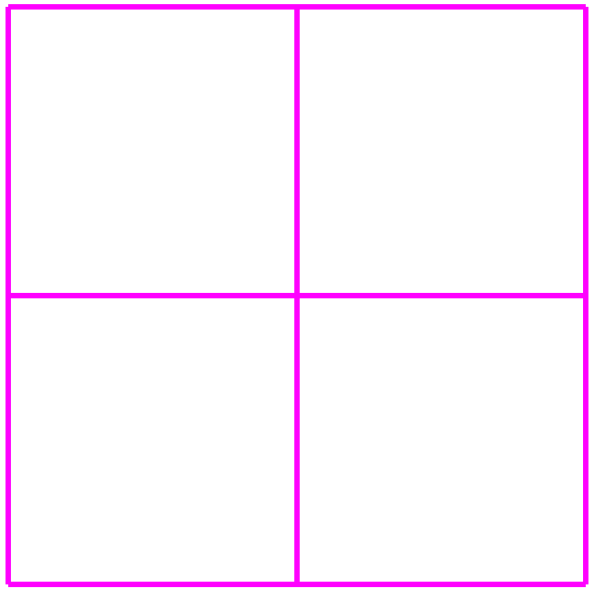
- Alternative to loose quadtree (octree)/cover fieldtree at overcoming drawback of MX-CIF quadtree that the width w of the minimum enclosing quadtree block of a rectangle o is not a function of the size of o

Partition Fieldtree

- Alternative to loose quadtree (octree)/cover fieldtree at overcoming drawback of MX-CIF quadtree that the width w of the minimum enclosing quadtree block of a rectangle o is not a function of the size of o
- Achieves similar result by shifting positions of the centroid of quadtree blocks at successive levels of the subdivision by one half of the width of the block that is being subdivided

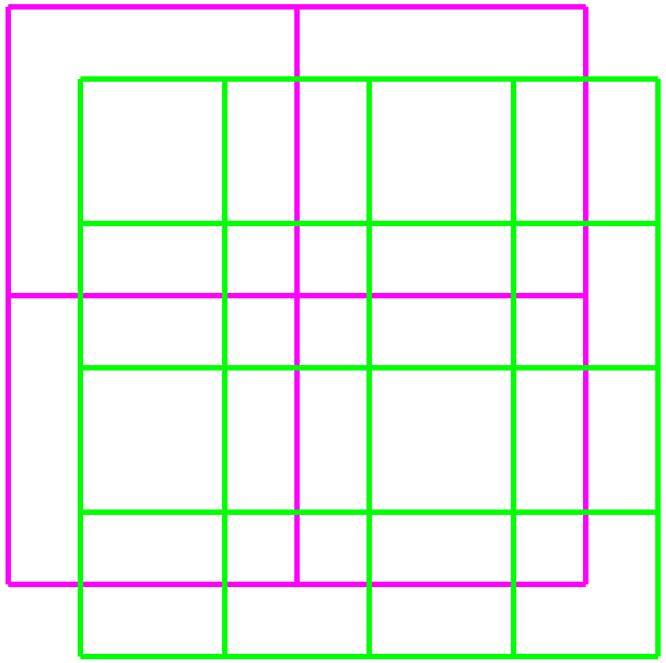
Partition Fieldtree

- Alternative to loose quadtree (octree)/cover fieldtree at overcoming drawback of MX-CIF quadtree that the width w of the minimum enclosing quadtree block of a rectangle o is not a function of the size of o
- Achieves similar result by shifting positions of the centroid of quadtree blocks at successive levels of the subdivision by one half of the width of the block that is being subdivided



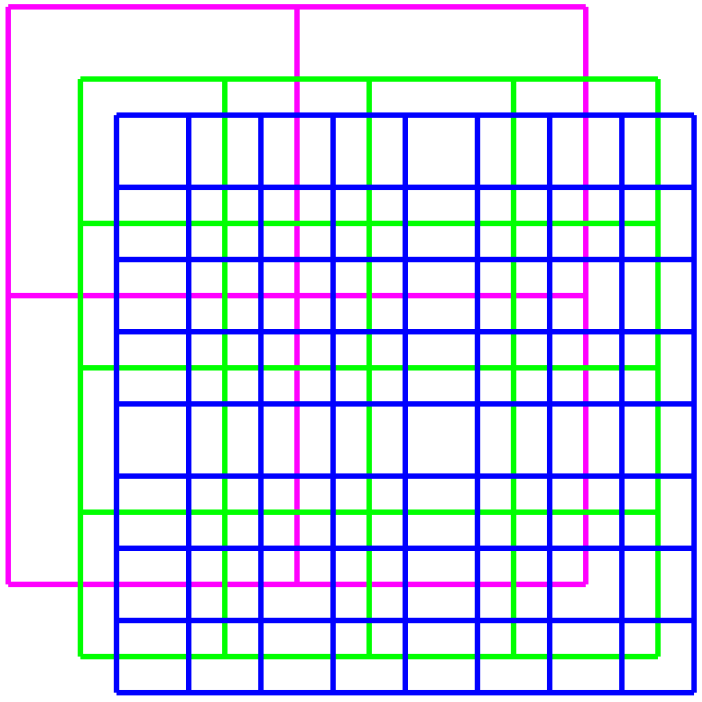
Partition Fieldtree

- Alternative to loose quadtree (octree)/cover fieldtree at overcoming drawback of MX-CIF quadtree that the width w of the minimum enclosing quadtree block of a rectangle o is not a function of the size of o
- Achieves similar result by shifting positions of the centroid of quadtree blocks at successive levels of the subdivision by one half of the width of the block that is being subdivided



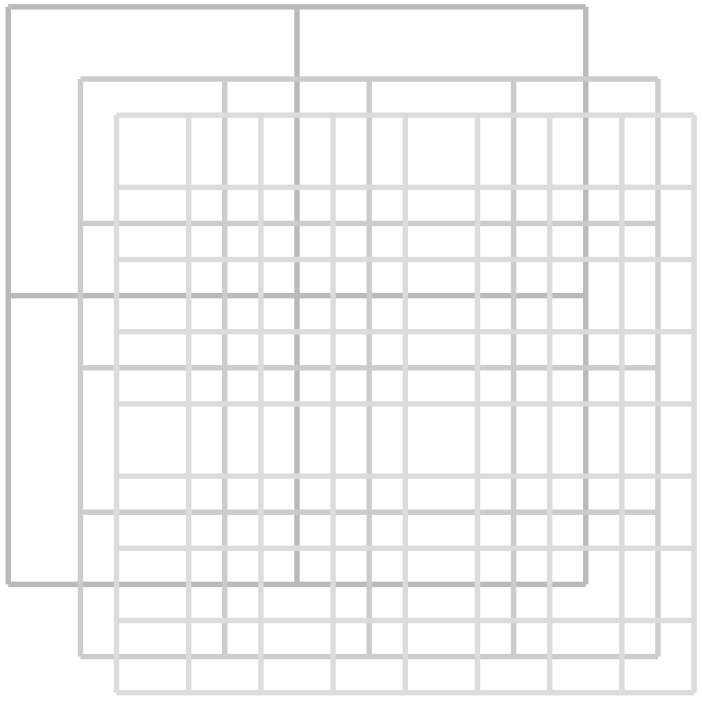
Partition Fieldtree

- Alternative to loose quadtree (octree)/cover fieldtree at overcoming drawback of MX-CIF quadtree that the width w of the minimum enclosing quadtree block of a rectangle o is not a function of the size of o
- Achieves similar result by shifting positions of the centroid of quadtree blocks at successive levels of the subdivision by one half of the width of the block that is being subdivided



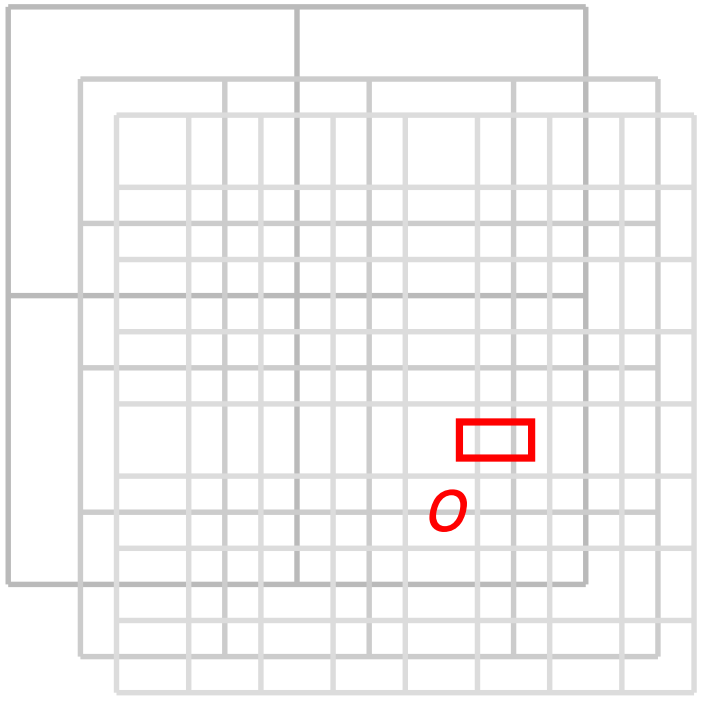
Partition Fieldtree

- Alternative to loose quadtree (octree)/cover fieldtree at overcoming drawback of MX-CIF quadtree that the width w of the minimum enclosing quadtree block of a rectangle o is not a function of the size of o
- Achieves similar result by shifting positions of the centroid of quadtree blocks at successive levels of the subdivision by one half of the width of the block that is being subdivided
- Subdivision rule guarantees that width of minimum enclosing quadtree block for rectangle o is bounded by 8 times the maximum extent r of o



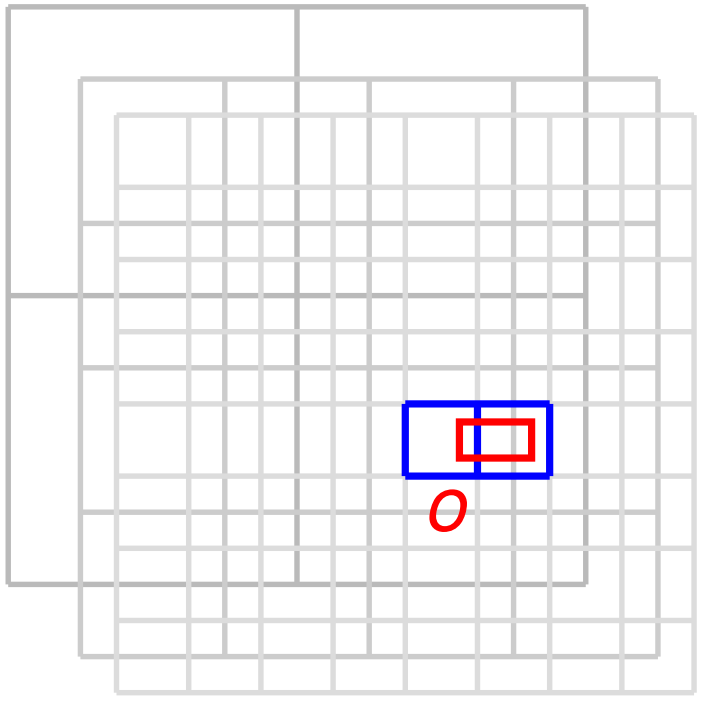
Partition Fieldtree

- Alternative to loose quadtree (octree)/cover fieldtree at overcoming drawback of MX-CIF quadtree that the width w of the minimum enclosing quadtree block of a rectangle o is not a function of the size of o
- Achieves similar result by shifting positions of the centroid of quadtree blocks at successive levels of the subdivision by one half of the width of the block that is being subdivided
- Subdivision rule guarantees that width of minimum enclosing quadtree block for rectangle o is bounded by 8 times the maximum extent r of o



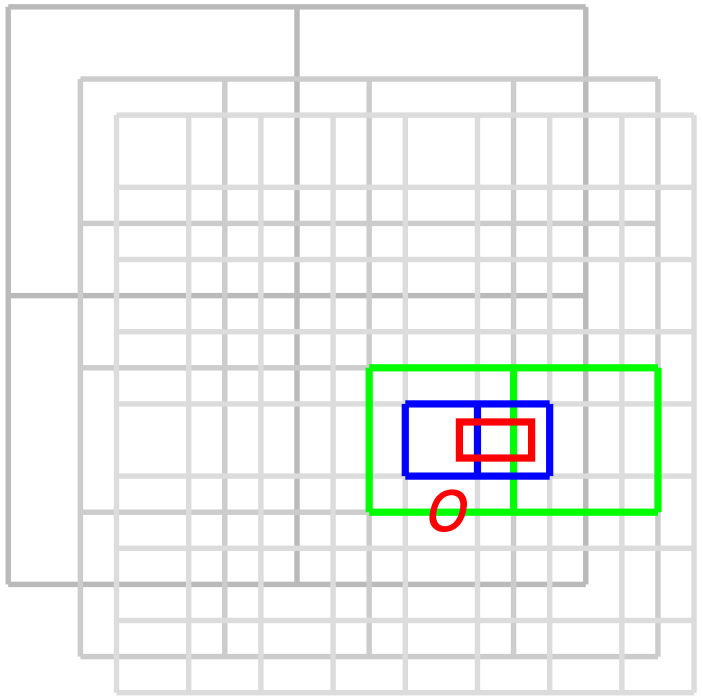
Partition Fieldtree

- Alternative to loose quadtree (octree)/cover fieldtree at overcoming drawback of MX-CIF quadtree that the width w of the minimum enclosing quadtree block of a rectangle o is not a function of the size of o
- Achieves similar result by shifting positions of the centroid of quadtree blocks at successive levels of the subdivision by one half of the width of the block that is being subdivided
- Subdivision rule guarantees that width of minimum enclosing quadtree block for rectangle o is bounded by 8 times the maximum extent r of o



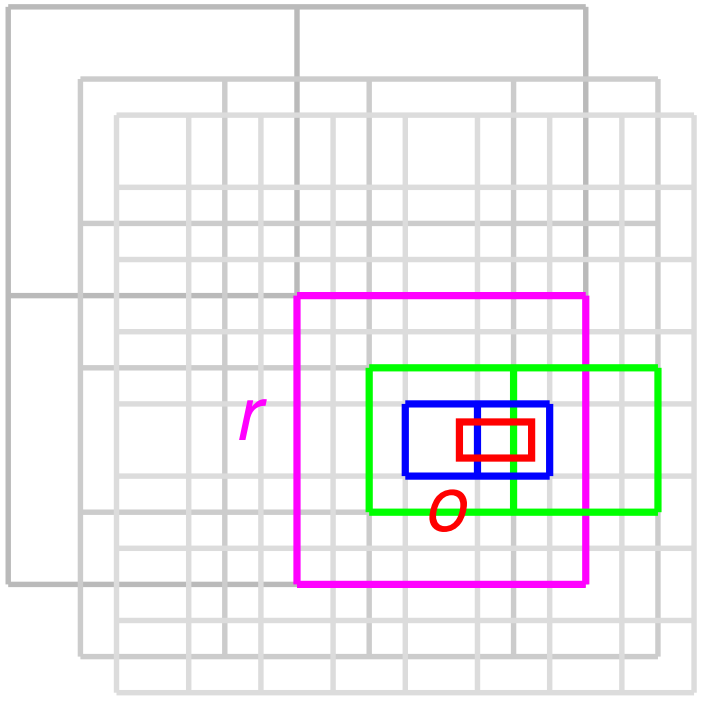
Partition Fieldtree

- Alternative to loose quadtree (octree)/cover fieldtree at overcoming drawback of MX-CIF quadtree that the width w of the minimum enclosing quadtree block of a rectangle o is not a function of the size of o
- Achieves similar result by shifting positions of the centroid of quadtree blocks at successive levels of the subdivision by one half of the width of the block that is being subdivided
- Subdivision rule guarantees that width of minimum enclosing quadtree block for rectangle o is bounded by 8 times the maximum extent r of o



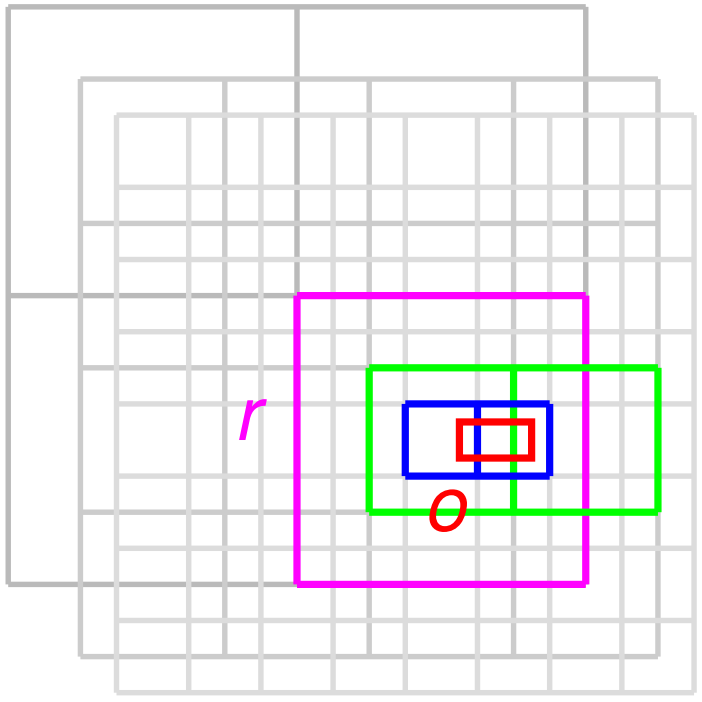
Partition Fieldtree

- Alternative to loose quadtree (octree)/cover fieldtree at overcoming drawback of MX-CIF quadtree that the width w of the minimum enclosing quadtree block of a rectangle o is not a function of the size of o
- Achieves similar result by shifting positions of the centroid of quadtree blocks at successive levels of the subdivision by one half of the width of the block that is being subdivided
- Subdivision rule guarantees that width of minimum enclosing quadtree block for rectangle o is bounded by 8 times the maximum extent r of o



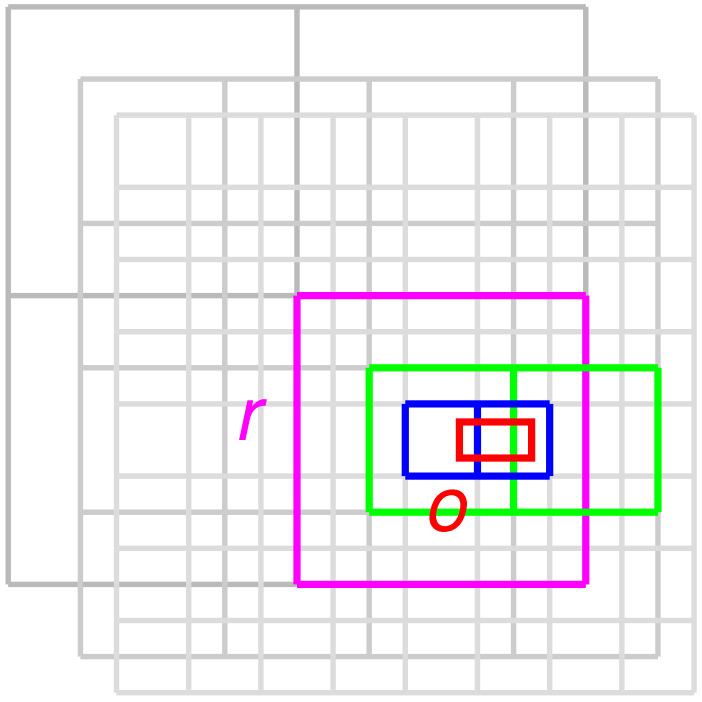
Partition Fieldtree

- Alternative to loose quadtree (octree)/cover fieldtree at overcoming drawback of MX-CIF quadtree that the width w of the minimum enclosing quadtree block of a rectangle o is not a function of the size of o
- Achieves similar result by shifting positions of the centroid of quadtree blocks at successive levels of the subdivision by one half of the width of the block that is being subdivided
- Subdivision rule guarantees that width of minimum enclosing quadtree block for rectangle o is bounded by 8 times the maximum extent r of o
- Same ratio is obtained for the loose quadtree (octree)/cover fieldtree when p =1/4, and thus partition fieldtree is superior to the cover fieldtree when p <1/4



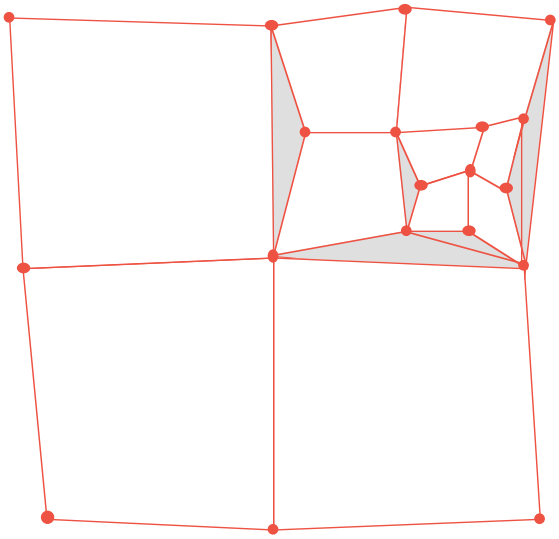
Partition Fieldtree

- Alternative to loose quadtree (octree)/cover fieldtree at overcoming drawback of MX-CIF quadtree that the width w of the minimum enclosing quadtree block of a rectangle o is not a function of the size of o
- Achieves similar result by shifting positions of the centroid of quadtree blocks at successive levels of the subdivision by one half of the width of the block that is being subdivided
- Subdivision rule guarantees that width of minimum enclosing quadtree block for rectangle o is bounded by 8 times the maximum extent r of o
- Same ratio is obtained for the loose quadtree (octree)/cover fieldtree when p =1/4, and thus partition fieldtree is superior to the cover fieldtree when p <1/4
- Summary: cover fieldtree expands the width of the quadtree blocks while the partition fieldtree shifts the positions of their centroids



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*)

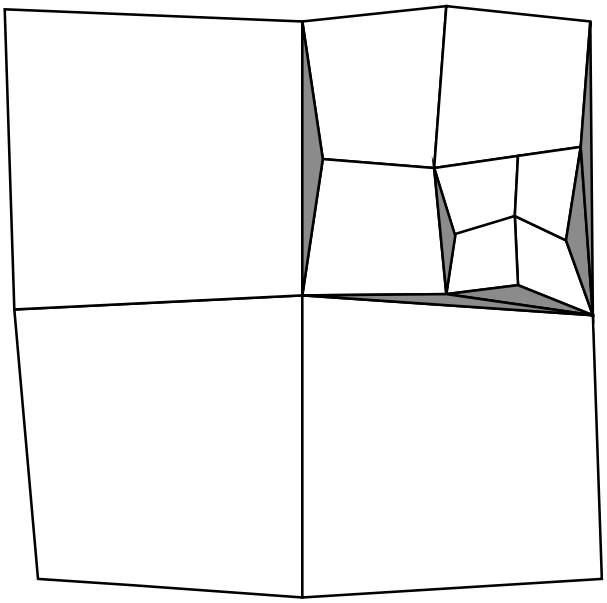


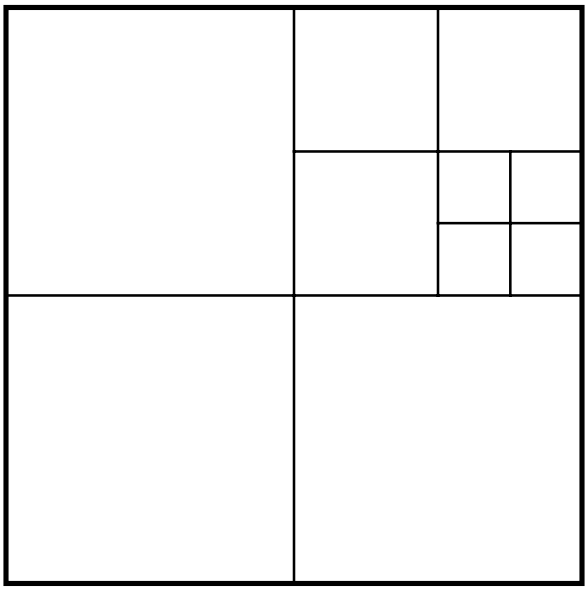
- Overcoming the presence of cracks
  1. use the interpolated point instead of the true point (Barrera and Hinjosa)
  2. triangulate the squares (Von Herzen and Barr)
     - can split into 2, 4, or 8 triangles depending on how many lines are drawn through the midpoint
     - if split into 2 triangles, 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)

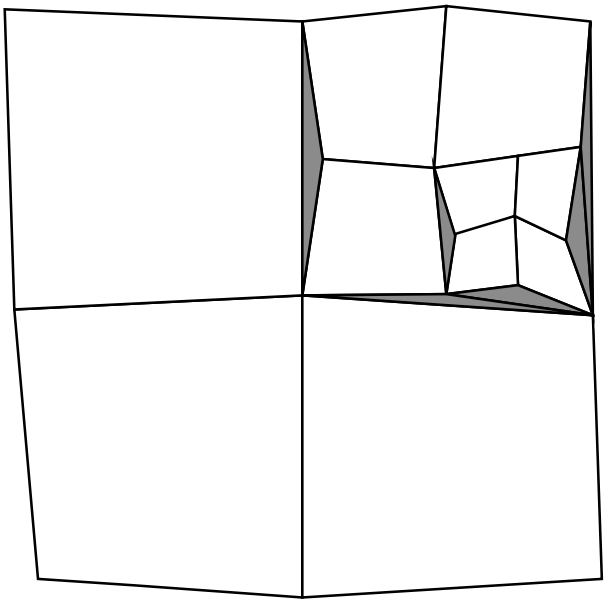


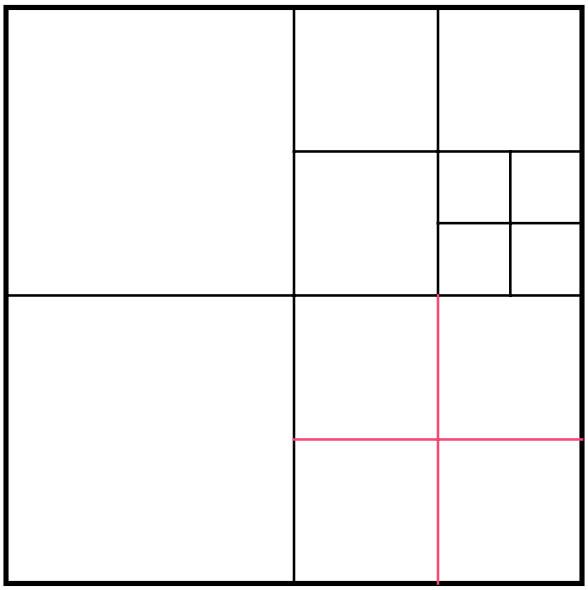


RESTRICTED QUADTREE (VON HERZEN/BARR)

- 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)

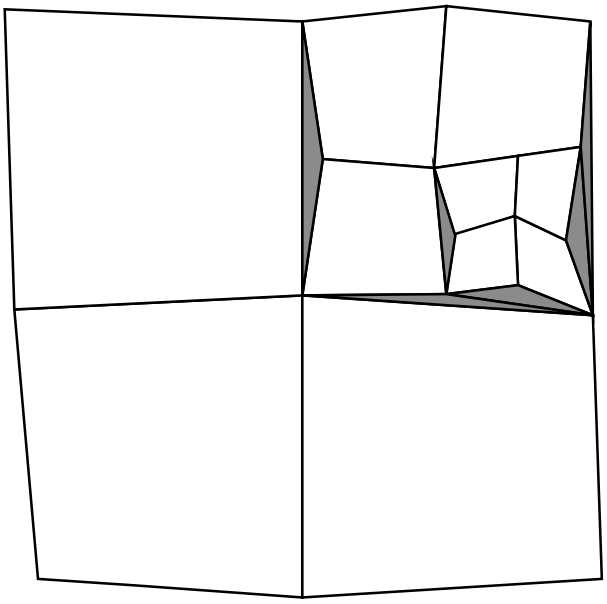


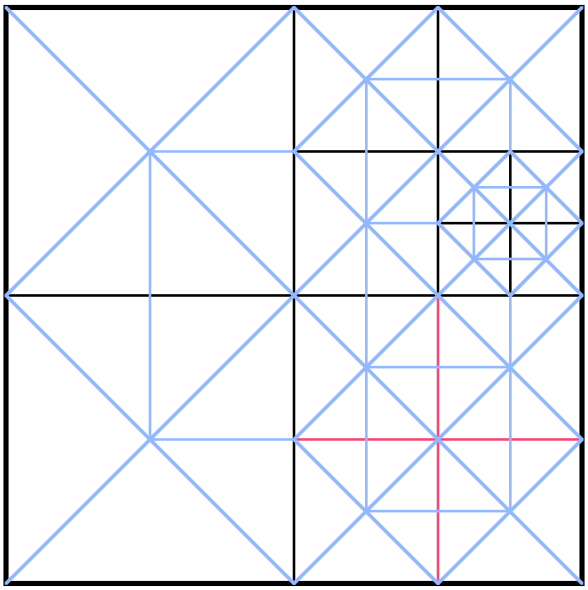


RESTRICTED QUADTREE (VON HERZEN/BARR)

- 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)



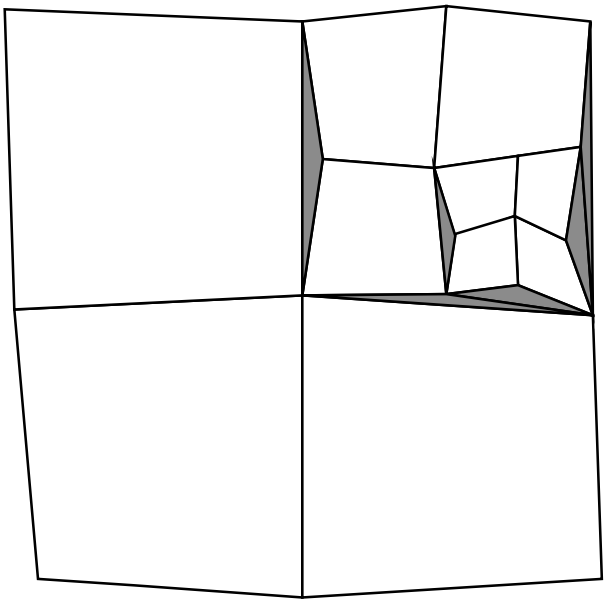


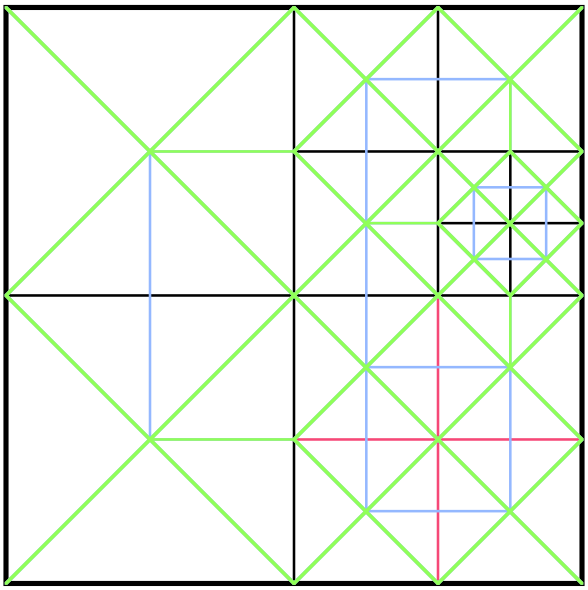
- 8-triangle decomposition rule
  1. decompose each block into 8 triangles (i.e., 2 triangles per edge)
  2. unless the edge is shared by a larger block
  3. in which case only 1 triangle is formed

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)



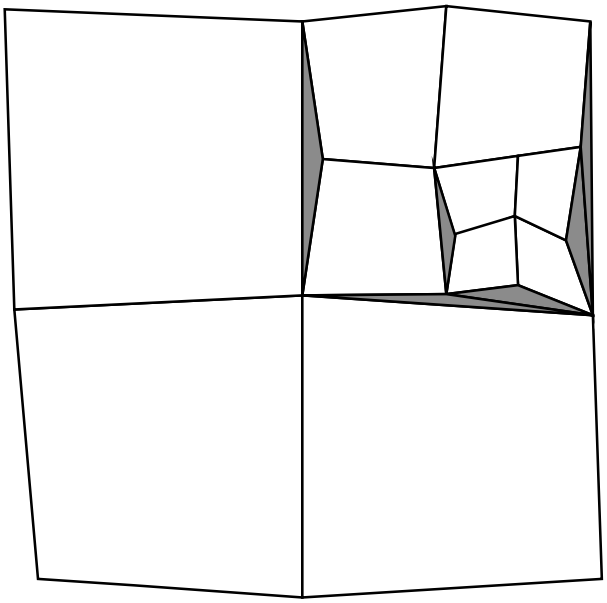


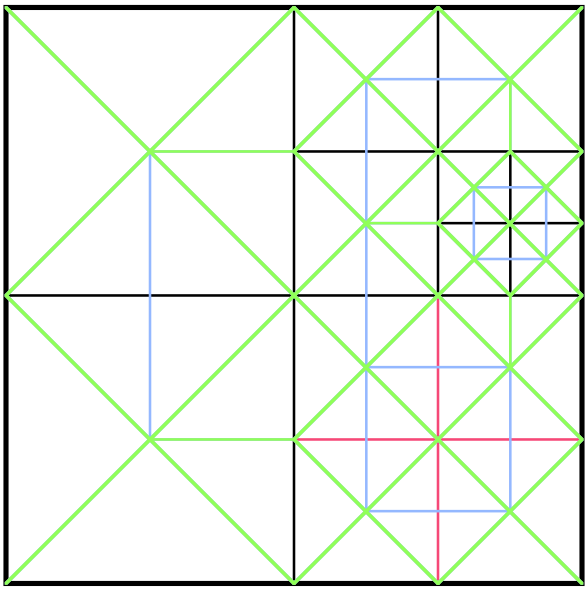
- 8-triangle decomposition rule
  1. decompose each block into 8 triangles (i.e., 2 triangles per edge)
  2. unless the edge is shared by a larger block
  3. in which case only 1 triangle is formed
- 4-triangle decomposition rule
  1. decompose each block into 4 triangles (i.e., 1 triangle per edge)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

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)

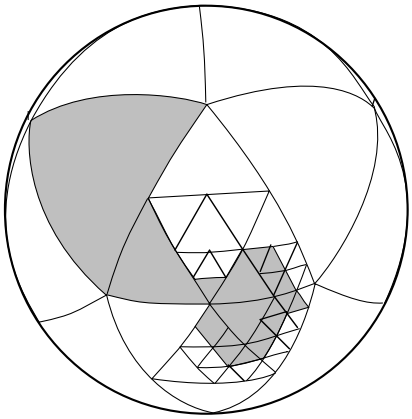


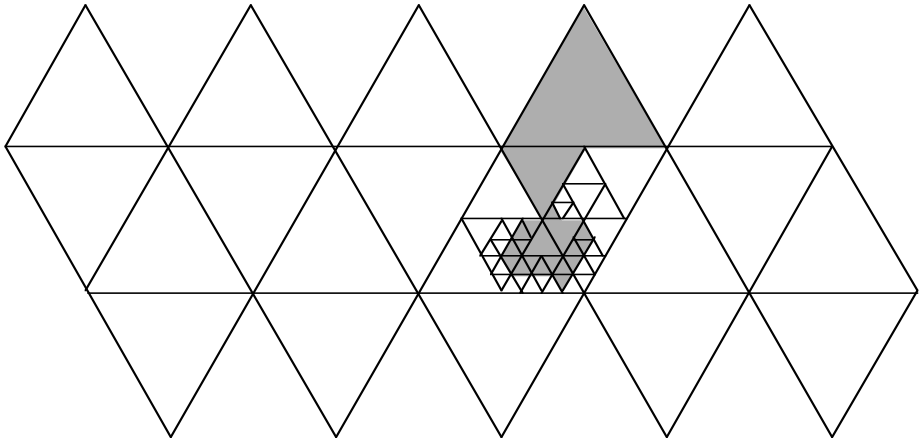


- 8-triangle decomposition rule
  1. decompose each block into 8 triangles (i.e., 2 triangles per edge)
  2. unless the edge is shared by a larger block
  3. in which case only 1 triangle is formed
- 4-triangle decomposition rule
  1. decompose each block into 4 triangles (i.e., 1 triangle per edge)
  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)

PROPERTY SPHERES (FEKETE)

- Approximation of spherical data
- Uses icosahedron which is a Platonic solid
  1. 20 faces—each is a regular triangle
  2. largest possible regular polyhedron





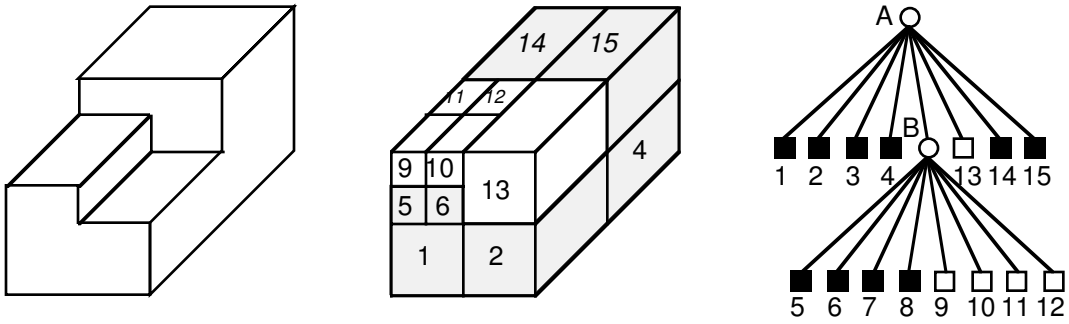
ALTERNATIVE SPHERICAL APPROXIMATIONS

- Could use other Platonic solids
  1. all have faces that are regular polygons
     - tetrahedron: 4 equilateral triangular faces
     - hexahedron: 6 square faces
     - octahedron: 8 equilateral triangular faces
     - dodecahedron: 12 pentagonal faces
  2. octahedron is nice for modeling the globe
     - it can be aligned so that the poles are at opposite vertices
     - the prime meridian and the equator intersect at another vertex
     - one subdivision line of each face is parallel to the equator
- Decompose on the basis of latitude and longitude values
  1. not so good if want a partition into units of equal area as great problems around the poles
  2. project sphere onto plane using Lambert's cylindrical projection which is locally area preserving
- Instead of approximating sphere with the solids, project the faces of the solids on the sphere (Scott)
  1. all edges become sub-arcs of a great circle
  2. use regular decomposition on triangular, square, or pentagonal spherical surface patches

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)

EXAMPLE QUADTREE-BASED QUERY

- Query: find all cities with population in excess of 5,000 in wheat growing regions within 10 miles of the Mississippi River
  1. assume river is a linear feature
     - use a line map
     - could be a region if asked for sandbars in the river
  2. region map for the wheat
  3. assume cities are points
     - point map for cities
     - could be region is asked for high income areas
- Combines spatial and non-spatial (i.e., attribute) data
- Many possible execution plans - e.g.,
  1. compute buffer or corridor around river
  2. extract wheat area
  3. intersect 1 with 2
  4. intersect city map with 3
  5. retrieve value of population attribute for cities in 4 from the nonspatial database (e.g., relational)
- Regular decomposition hierarchical data structures such as the quadtree
  1. all maps are in registration
     - all blocks are in the same positions
     - not true for R+-trees and BSP trees
     - disjoint decomposition of space - unlike R-tree
  2. can perform set-theoretic operations on different feature types (e.g., 3 and 4)

FURTHER READING

1. F. Brabec and H. Samet, Client–based spatial browsing on the world wide web. IEEE Internet Computing, 11(1):52–59, Jan/Feb 2007.

2. H. Samet, Foundations of Multidimensional and Metric Data Structures, Morgan–Kaufmann, San Francisco, 2006. [http://www.cs.umd.edu/~hjs/multidimensional–book–flyer.pdf]

3. H. Samet, Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS, Addison–Wesley, Reading, MA, 1990.

4. H. Samet, Design and Analysis of Spatial Data Structures, Addison–Wesley, Reading, MA, 1990.

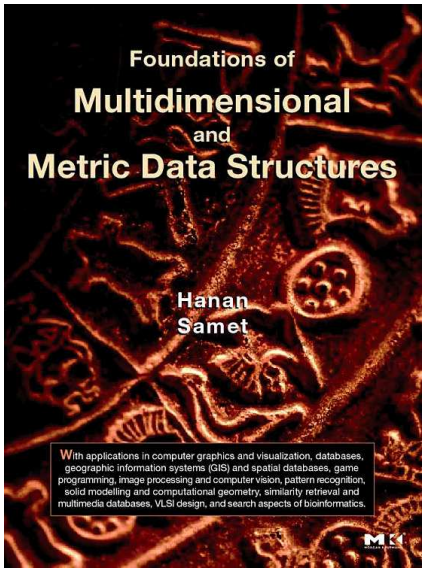
5. Spatial Data Applets at http://www.cs.umd.edu/~hjs/quadtree

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~~ ~~£29.99~~ ~~44.95~~ $47.96 £23.99 35.16



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.

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.

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