Sorting in Space

Multidimensional, Spatial, and Metric Data Structures for Applications in Spatial Databases, Geographic Information Systems (GIS), and Location-Based Services

Hanan Samet
hjs@cs.umd.edu

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

Department of Computer Science

University of Maryland

College Park, MD 20742, USA

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

Speaker Biography

Hanan Samet (http://www.cs.umd.edu/~hjs/) received the B.S. degree in engineering from UCLA, and the M.S. Degree in operations research and the M.S. and Ph.D. degrees in computer science from Stanford University. At Stanford, he was a member of the Stanford Artificial Intelligence Lab where he was one of the developers of the SAIL programming language compiler. His doctoral dissertation dealt with proving the correctness of translations of LISP programs which was the first work in translation validation as well as the related proof-carrying code.

In 1975 he joined the Computer Science Department at the University of Maryland, College Park, where he is a Professor. He is a member of the Computer Vision Laboratory and leads a number of research projects on the use of hierarchical data structures for geographic information systems, computer graphics, image processing, and search. His research group has developed the QUILT system which is a GIS based on hierarchical spatial data structures such as quadtrees and octrees, the SAND system which integrates spatial and non-spatial data, the SAND Browser (http://www.cs.umd.edu/~brabec/sandjava) which enables browsing through a spatial database using a graphical user interface, the VASCO spatial indexing applet (found at http://www.cs.umd.edu/~hjs/quadtree/index.html), the MARCO system for map retrieval by content which consists of a sophisticated pictorial query specification method, the STEWARD system for identifying the geographic focus of documents thereby facilitating the performance of spatio-textual search to enable searches that rank the results by spatial proximity rather than by exact match, and the NewsStand and TwitterStand systems that apply these ideas to a database of news articles and Tweets, respectively, that are continuously updated and that enable them to be accessed using a map query interface.

He is the founding editor-in-chief of the ACM Transactions on Spatial Algorithms and System (TSAS), an associate editor of Graphical Models, an advisory editor of the Journal of Visual Languages and Computing, and on the editorial boards of GeoInformatica and Image Understanding. He is the founding chair of the ACM SIG on Spatial Information. He has served as the co-general chair of the 2007 and 2008 ACM SIGSPATIAL Conference on Geographic Information Systems (ACM GIS).

His research interests include data structures, computer graphics, geographic information systems, computer vision, robotics, database management systems, and programming languages, and is the author of over 300 publications on these topics. He is the author of the recent book titled "Foundations of Multidimensional and Metric Data Structures" (http://www.cs.umd.edu/~hjs/multidimensional-book-flyer.pdf) published by Morgan-Kaufmann, an imprint of Elsevier, in 2006, an award winner in the 2006 best book in Computer and Information Science competition of the Professional and Scholarly Publishers (PSP) Group of the American Publishers Association (AAP), and of the first two books on spatial data structures titled "Design and Analysis of Spatial Data Structures", and "Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS", both published by Addison-Wesley in 1990.

He is a Fellow of the IEEE, ACM, AAAS, and IAPR (International Association for Pattern Recognition), and was also elected to the ACM Council in 1989-1991 where he served as the Capital Region Representative. He received the 2009 UCGIS research award, the 2011 ACM Paris Kanellakis theory and practice award, and best paper awards in the 2008 ACM SIGMOD and SIGSPATIAL Conferences, and in the 2012 SIGSPATIAL MobiGIS Workshop.

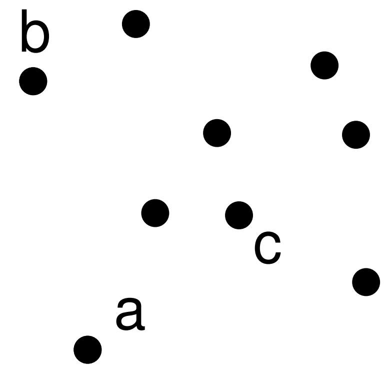
Outline

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

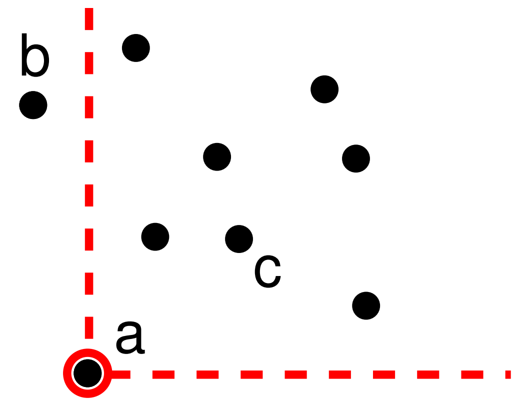
1. Fine for one-dimensional data
   - sort people by weight and find closest in weight to Bill and can also find closest in weight to Larry
   - sort cities by distance from Chicago and find closest to Chicago but cannot find closest to New York unless resort



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

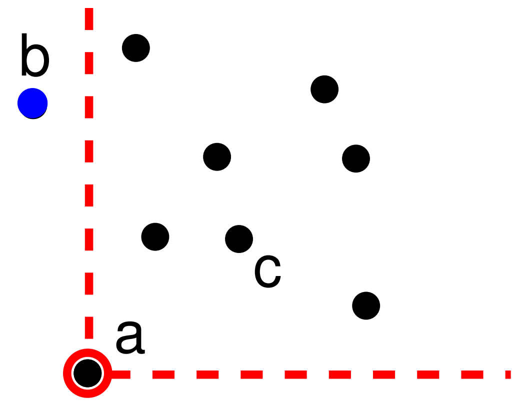
1. Fine for one-dimensional data
   - sort people by weight and find closest in weight to Bill and can also find closest in weight to Larry
   - sort cities by distance from Chicago and find closest to Chicago but cannot find closest to New York unless resort



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

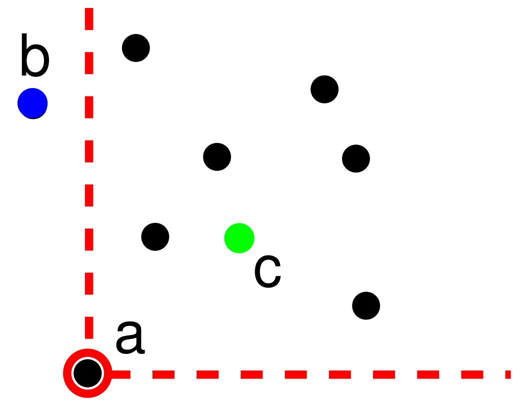
1. Fine for one-dimensional data
   - sort people by weight and find closest in weight to Bill and can also find closest in weight to Larry
   - sort cities by distance from Chicago and find closest to Chicago but cannot find closest to New York unless resort



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

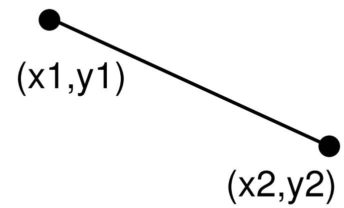
Map of Prince George's County

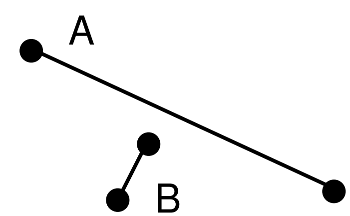
Example Queries in Line Segment Databases

1. Queries about line segments
   - All segments that intersect a given point or set of points
   - All segments that have a given set of endpoints
   - All segments that intersect a given line segment
   - All segments that are coincident with a given line segment
2. Proximity queries
   - The nearest line segment to a given point
   - All segments within a given distance from a given point (also known as a range or window query)
3. Queries involving attributes of line segments
   - Given a point, find the closest line segment of a particular type
   - Given a point, find the minimum enclosing polygon whose constituent line segments are all of a given type
   - 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 (x1 , y1 , x2 , y2)
   - 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)
   - 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

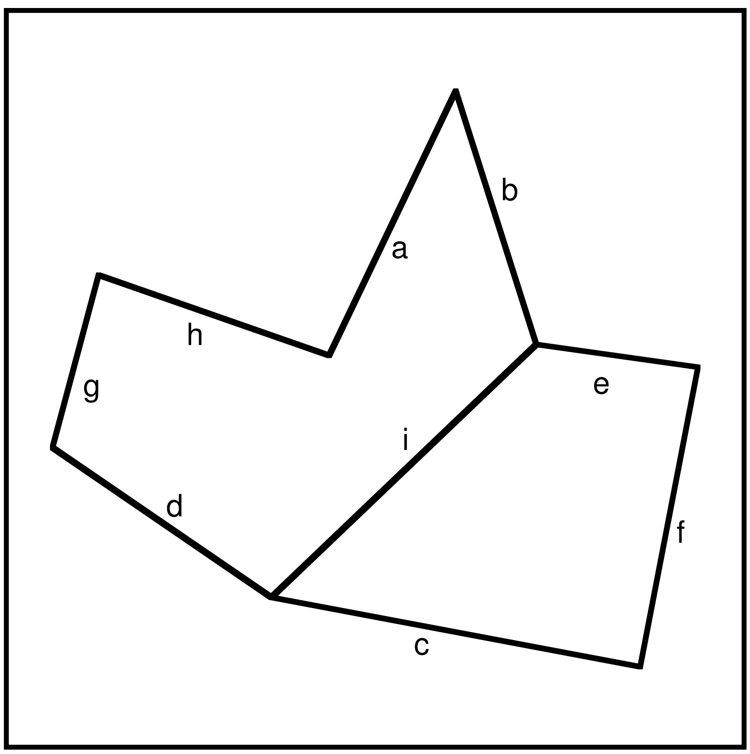
1. Compatibility with the data being stored
2. Choose an appropriate zero or reference point
3. Need an implicit rather than an explicit index
    a. impossible to foresee all possible queries in advance
    b. cannot have an attribute for every possible spatial relationship
        i. derive adjacency relations
        ii. 2-d strings capture a subset of adjacencies
            A. all rows
            B. all columns
    c. 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
    d. 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 or AABB (axis-aligned) and OBB (arbitrary orientation)
     - 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

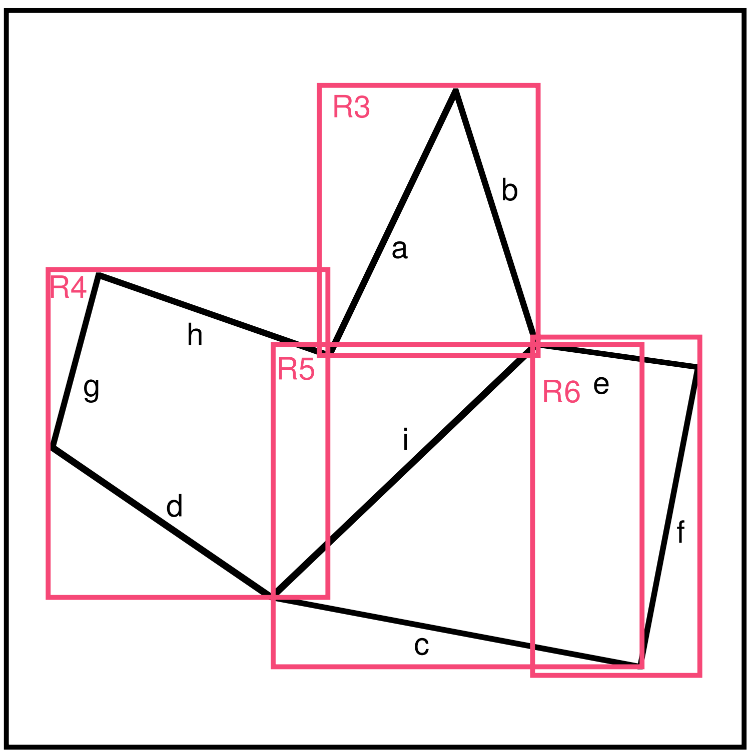
MINIMUM BOUNDING RECTANGLES

- 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

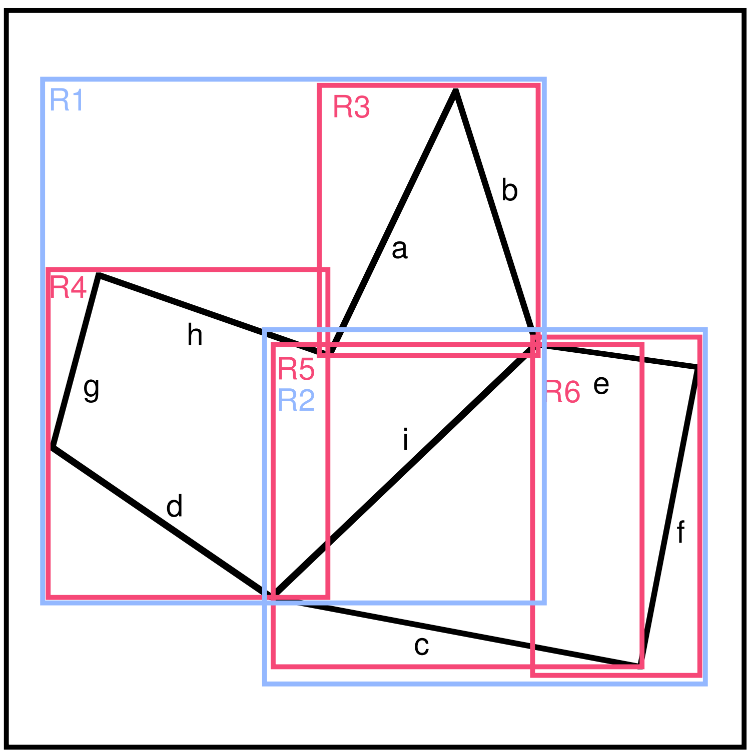
- Objects grouped into hierarchies, stored in a structure similar to a B-tree
- Drawback: not a disjoint decomposition of space
- Object has single bounding rectangle, yet area that it spans may be included in several bounding rectangles
- Examples include the R-tree and the R*-tree
- Order (*m*,*M*)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

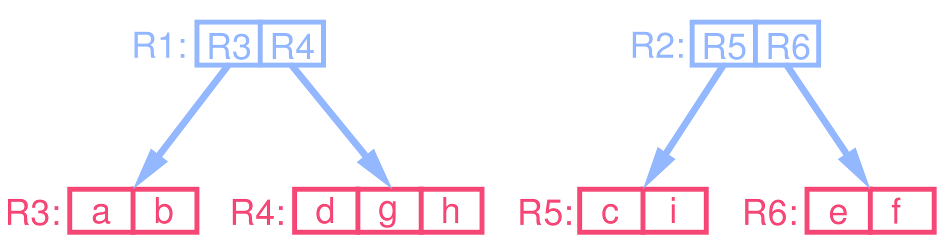


R3: | a | b | R4: | d | g | h | R5: | c | i | R6: | e | f |

MINIMUM BOUNDING RECTANGLES

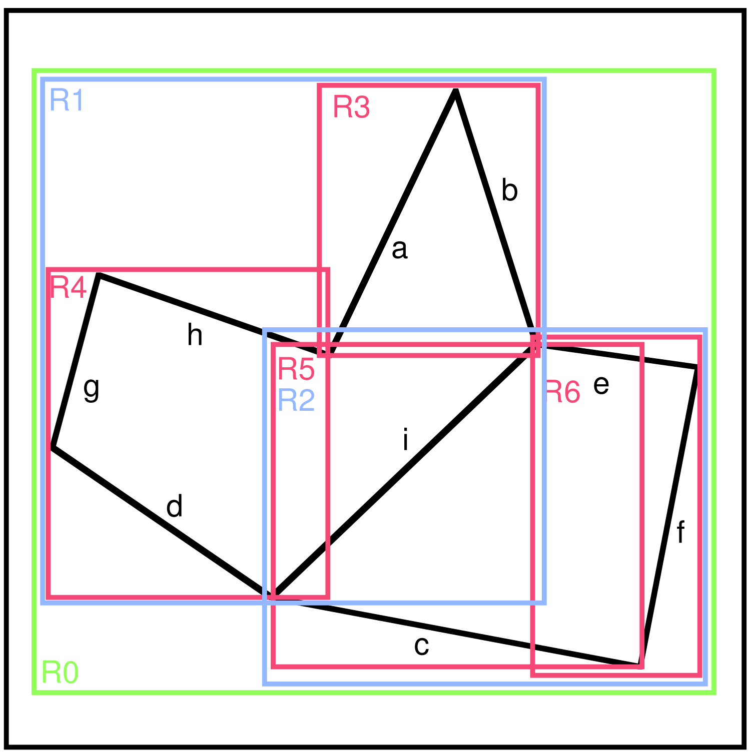
- Objects grouped into hierarchies, stored in a structure similar to a B-tree
- Drawback: not a disjoint decomposition of space
- Object has single bounding rectangle, yet area that it spans may be included in several bounding rectangles
- Examples include the R-tree and the R*-tree
- Order (m,M) R-tree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

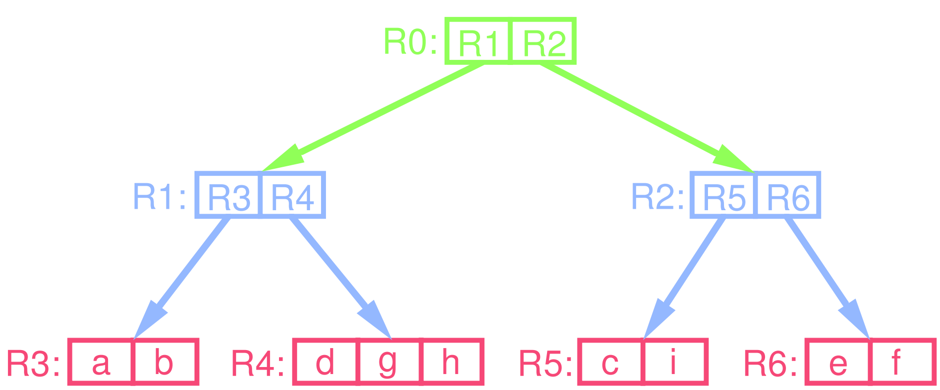




MINIMUM BOUNDING RECTANGLES

- 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

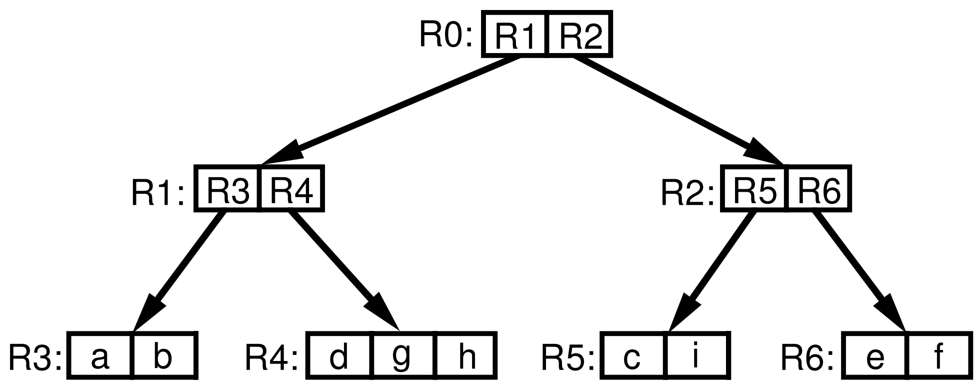


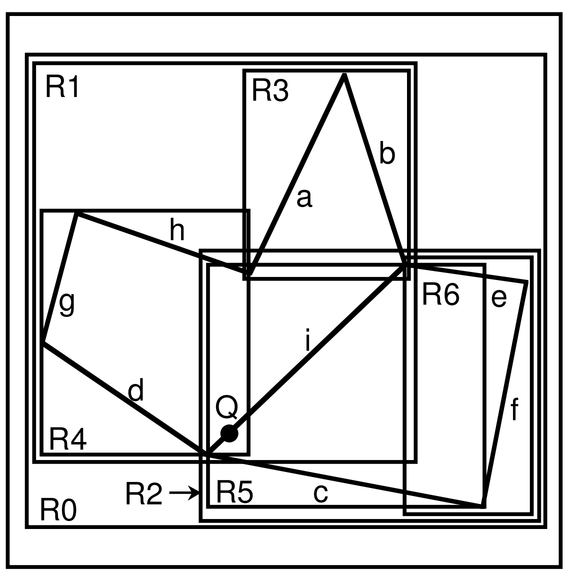


SEARCHING FOR A POINT OR LINE SEGMENT IN AN R-TREE

- 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

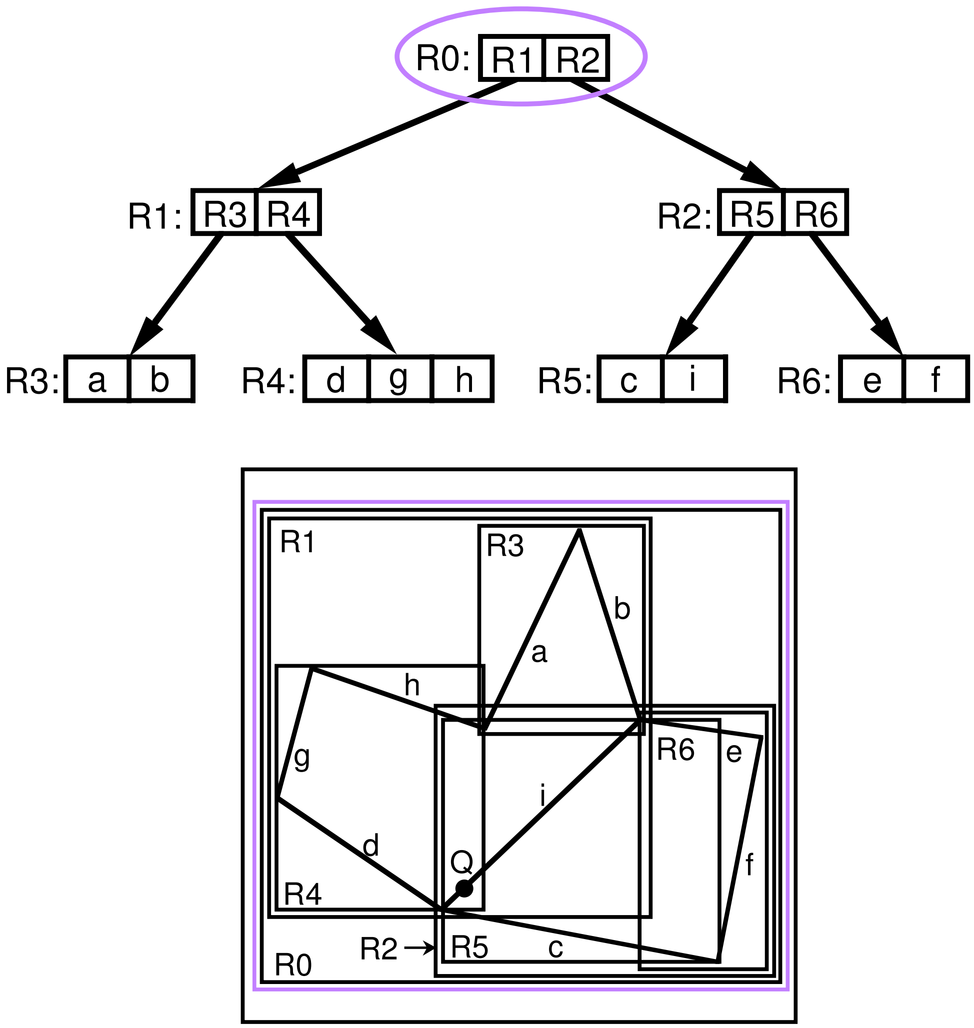




SEARCHING FOR A POINT OR LINE SEGMENT IN AN R-TREE

- 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

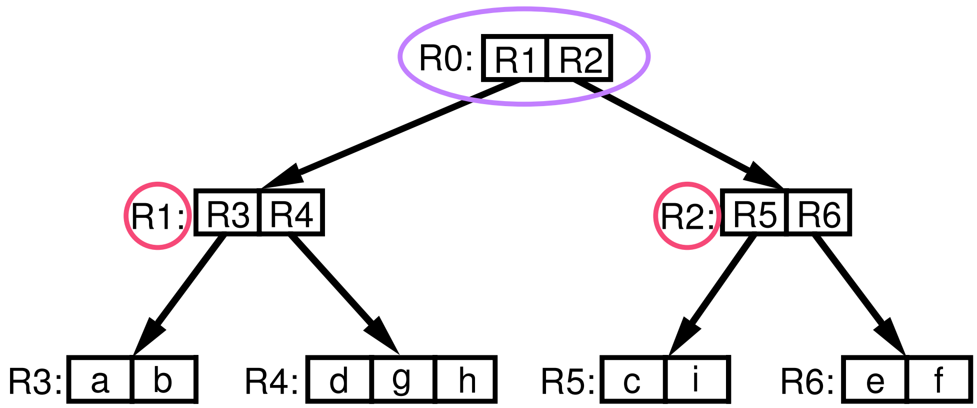


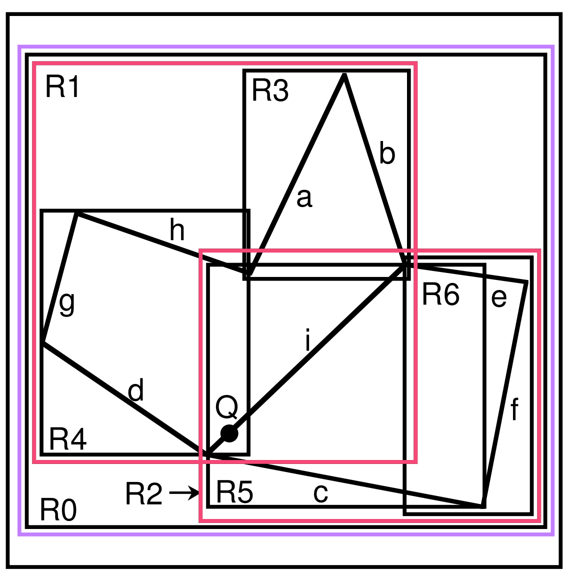
- Q is in R0

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



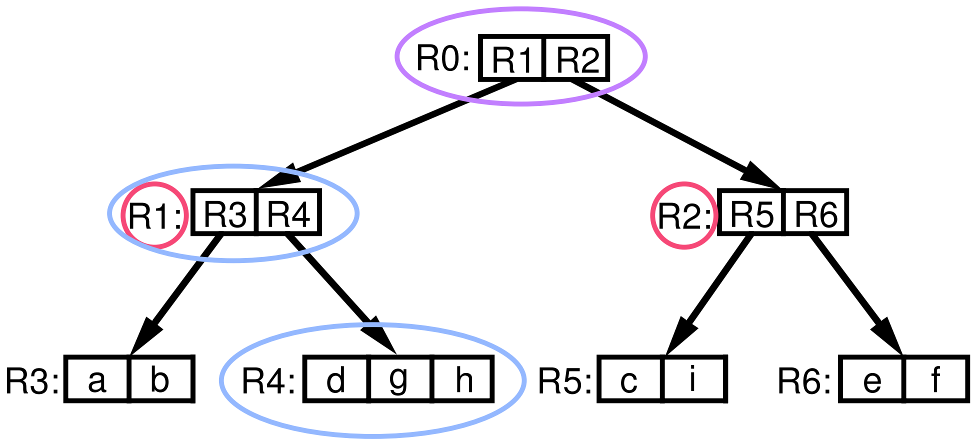


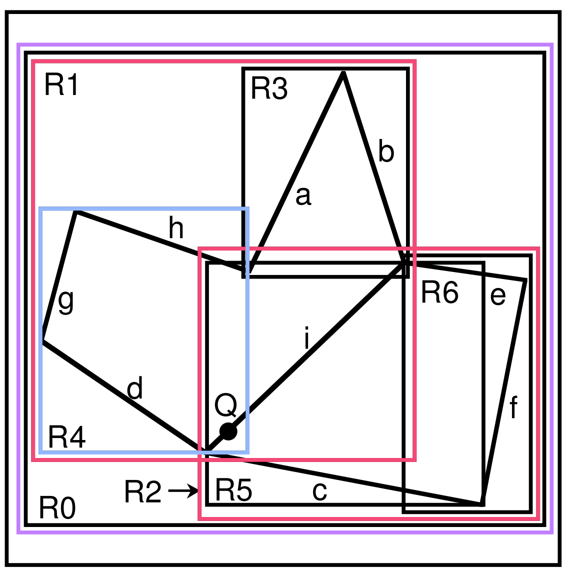
- Q is in R0
- Q can be in both R1 and R2

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



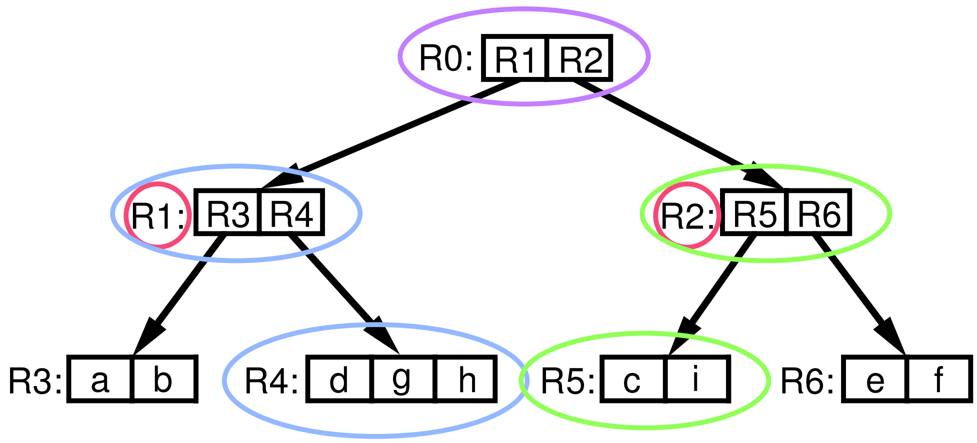


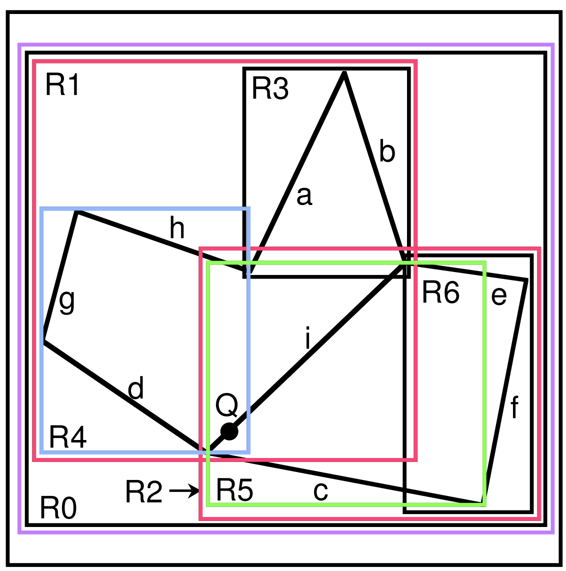
- Q is in R0
- Q can be in both R1 and R2
- Searching R1 first means that R4 is searched but this leads to failure even though Q is part of i which is in R4

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

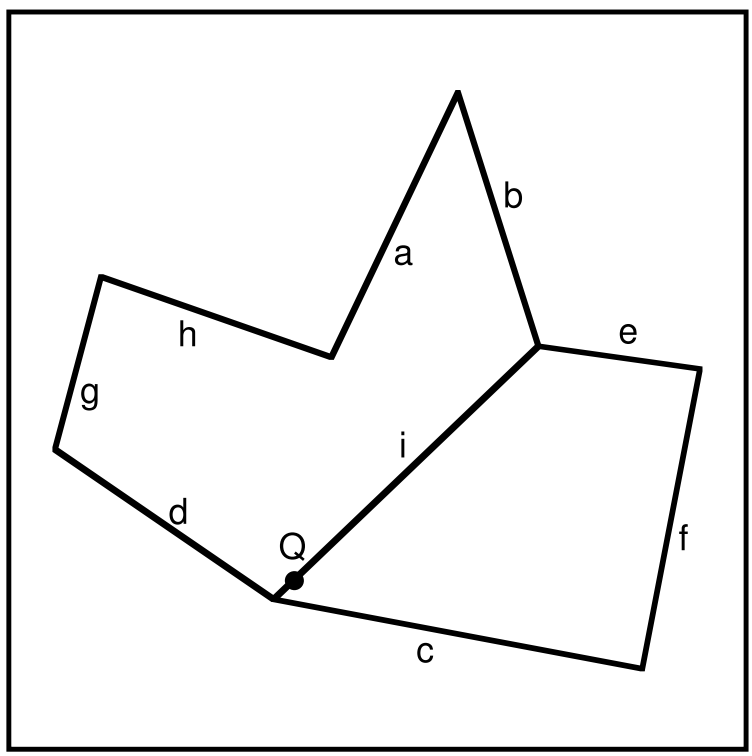




- Q is in R0
- Q can be in both R1 and R2
- Searching R1 first means that R4 is searched but this leads to failure even though Q is part of i which is in R4
- Searching R2 finds that Q can only be in R5

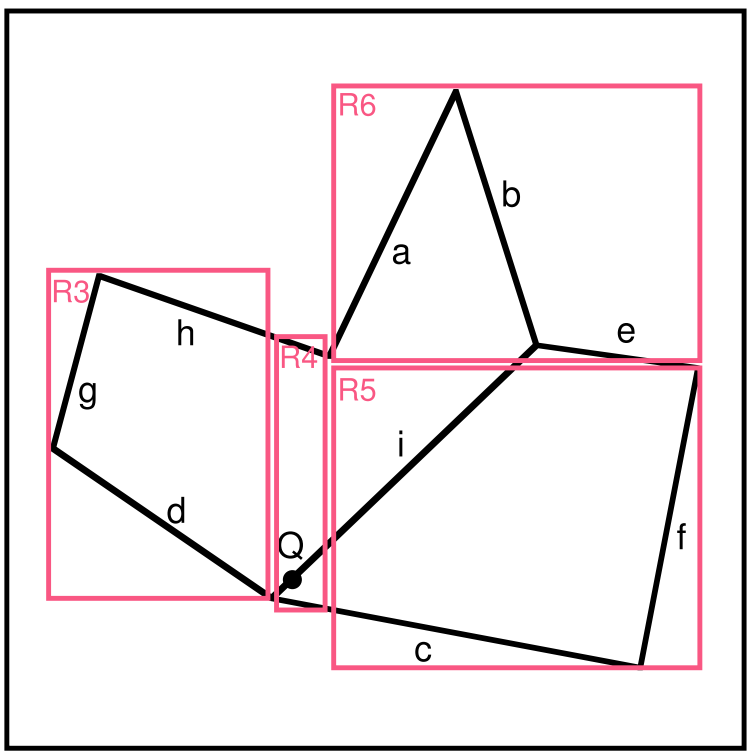
DISJOINT CELLS

- 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

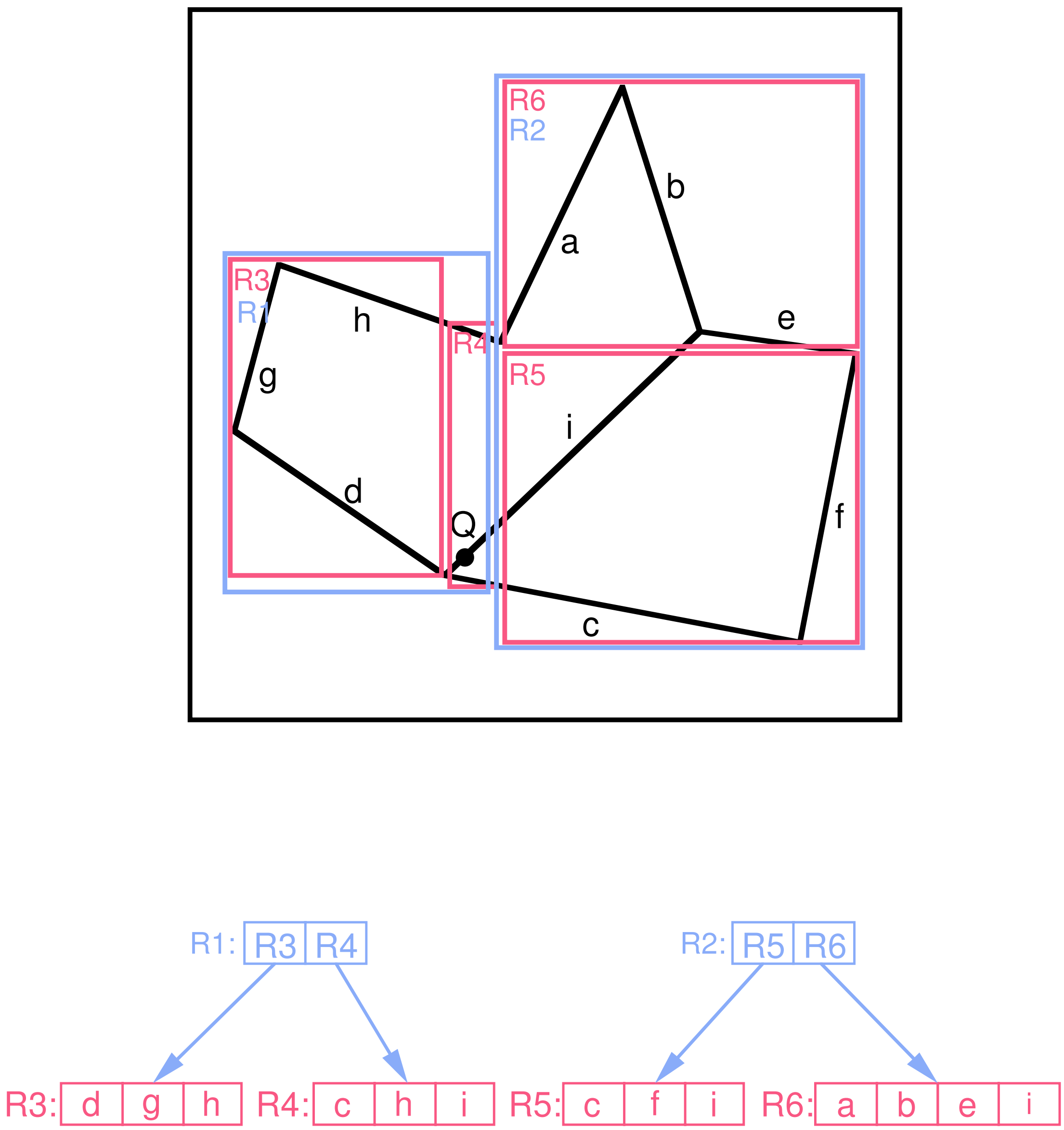
- Objects decomposed into disjoint subobjects; each subobject in different cell
- Techniques differ in degree of regularity
- Drawback: in order to determine area covered by object, must retrieve all cells that it occupies
- R+-tree (also k-d-B-tree) and cell tree are examples of this technique



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

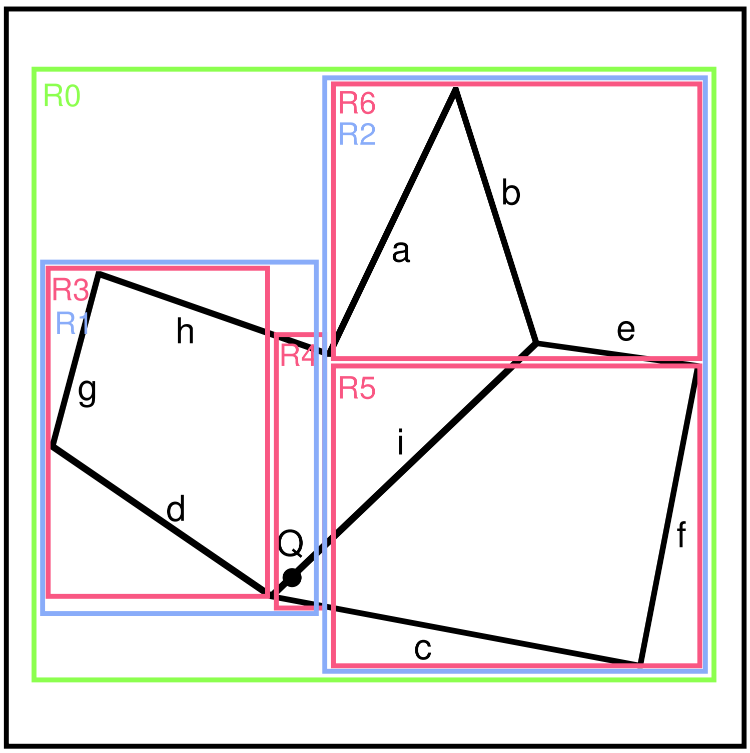
DISJOINT CELLS

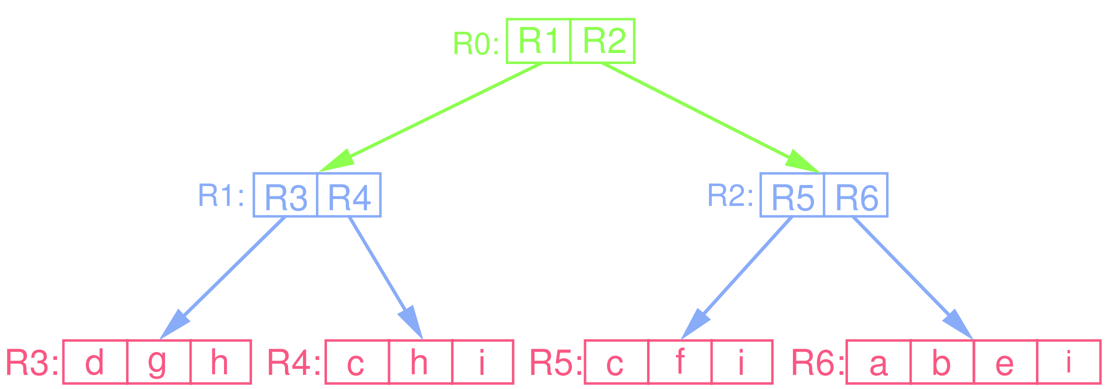
- Objects decomposed into disjoint subobjects; each subobject in different cell
- Techniques differ in degree of regularity
- Drawback: in order to determine area covered by object, must retrieve all cells that it occupies
- R+-tree (also k-d-B-tree) and cell tree are examples of this technique



DISJOINT CELLS

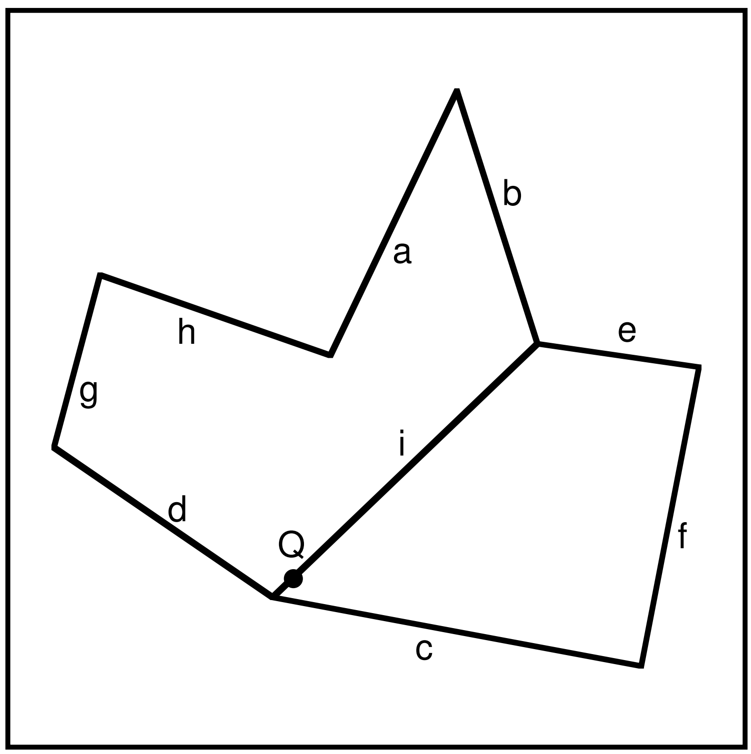
- Objects decomposed into disjoint subobjects; each subobject in different cell
- Techniques differ in degree of regularity
- Drawback: in order to determine area covered by object, must retrieve all cells that it occupies
- R+-tree (also k-d-B-tree) and cell tree are examples of this technique





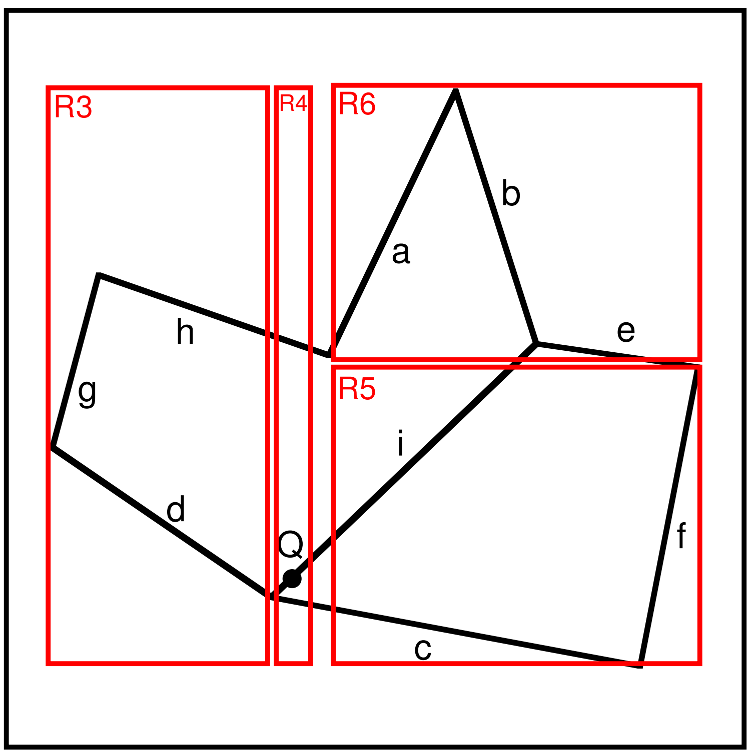
K-D-B-TREES

- 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

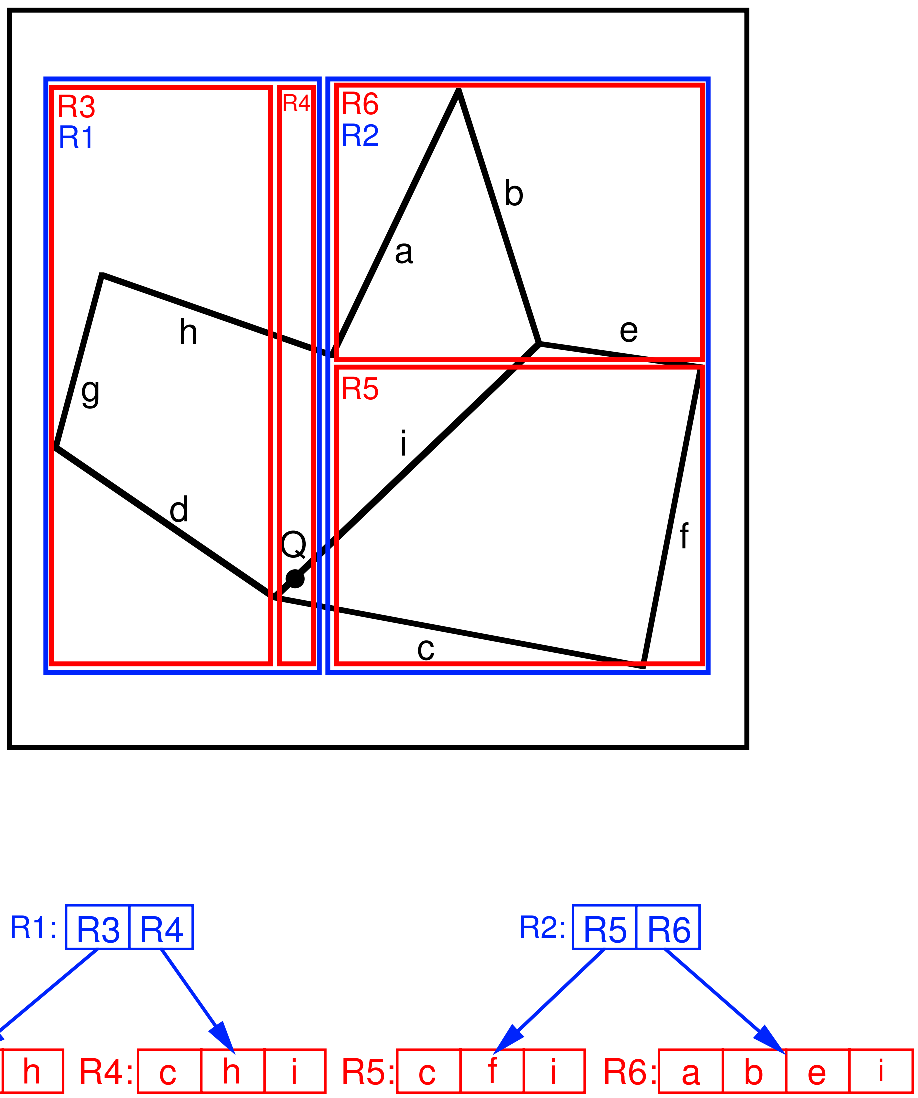
- Rectangular embedding space is hierarchically decomposed into disjoint rectangular regions
- No dead space in the sense that at any level of the tree, entire embedding space is covered by one of the nodes
- Blocks of k-d tree partition of space are aggregated into nodes of a finite capacity
- When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

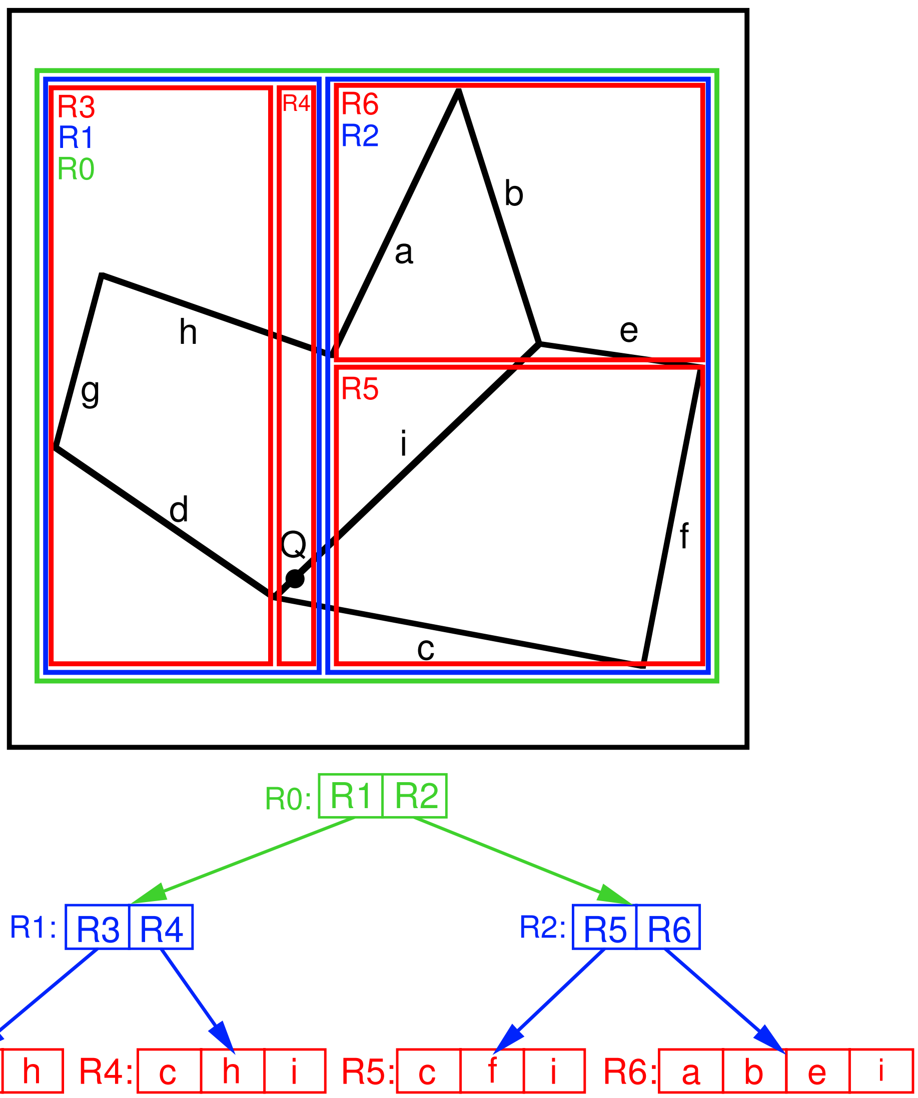
K-D-B-TREES

- 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



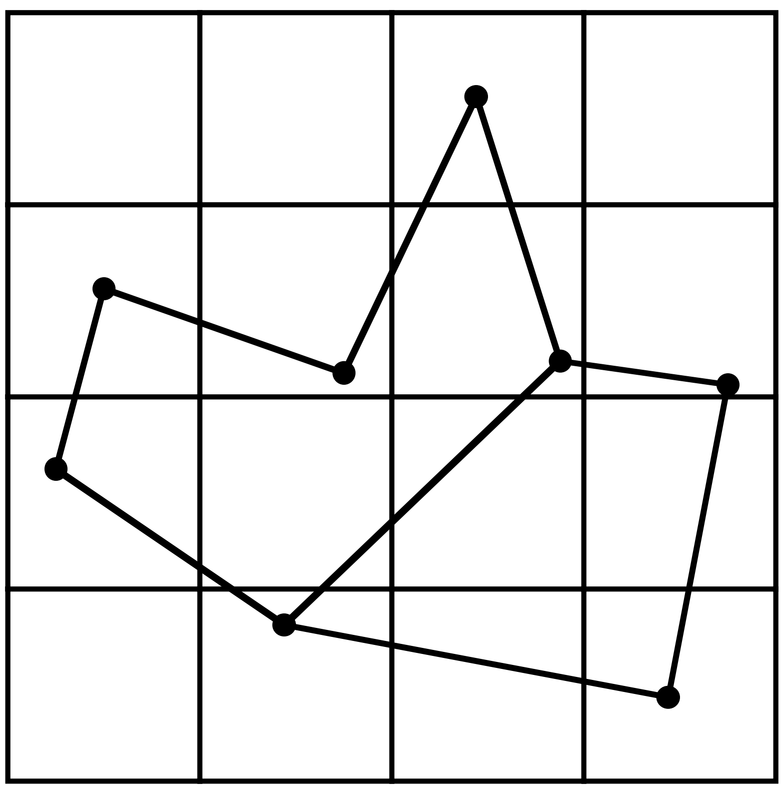
K-D-B-TREES

- 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



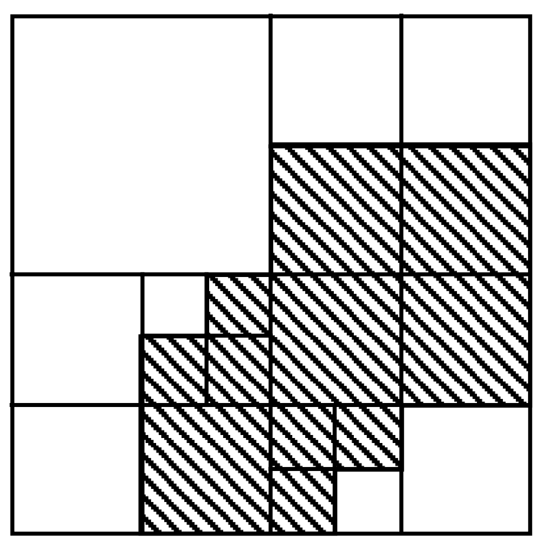
UNIFORM GRID

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



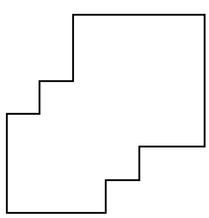
QUADTREES

- 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!

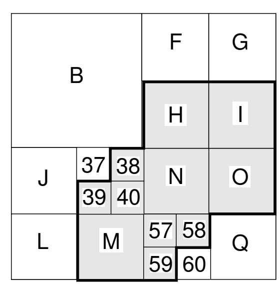


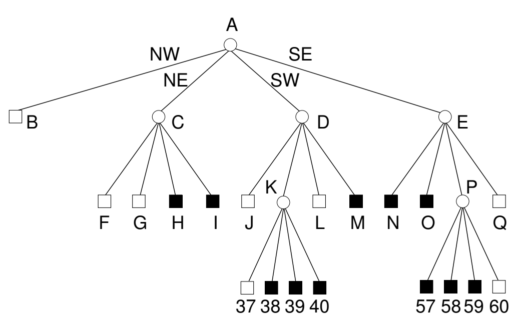
REGION QUADTREE

- 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



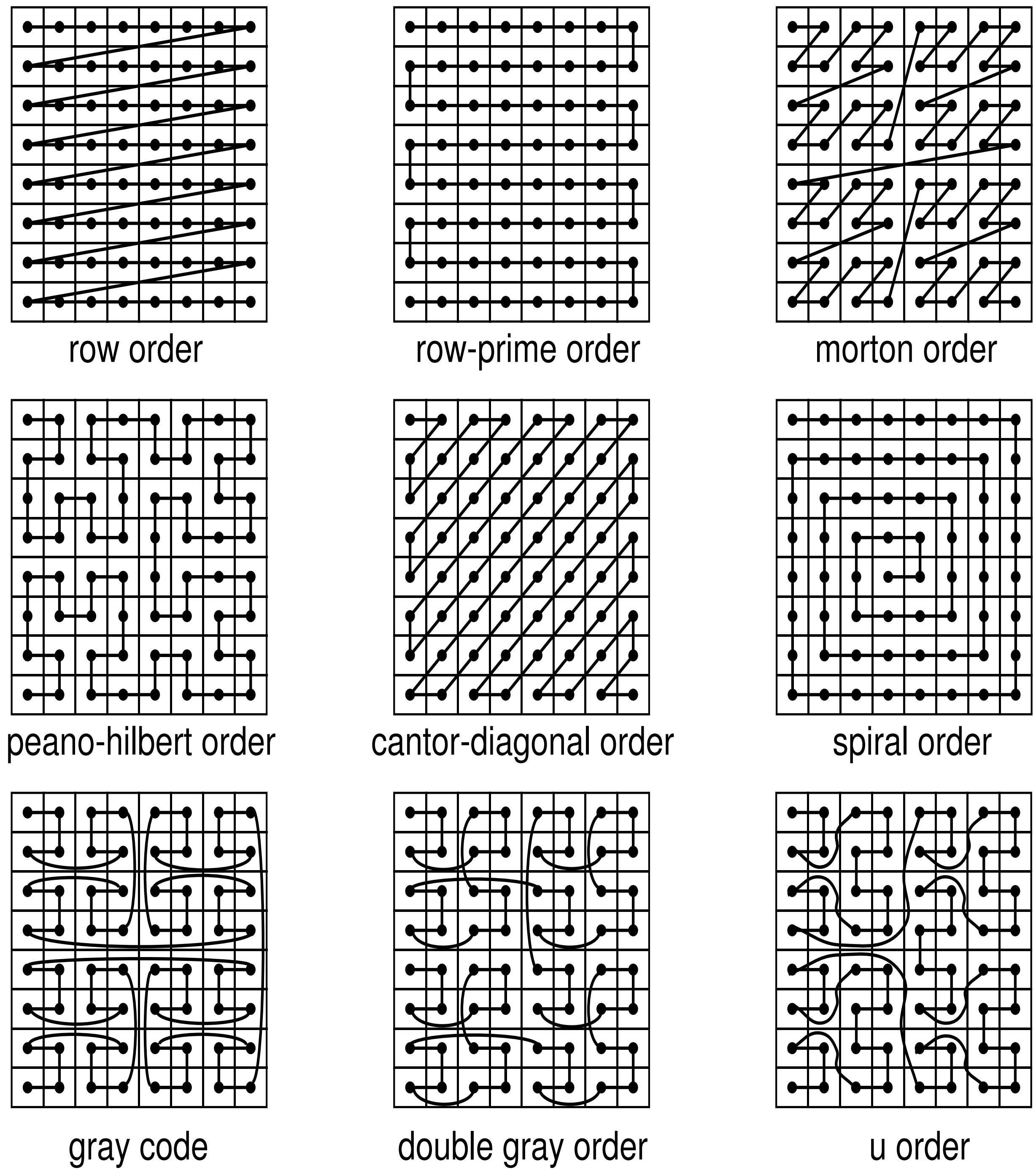
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	1	1	1	1
0	0	0	0	1	1	1	1
0	0	0	1	1	1	1	1
0	0	1	1	1	1	1	1
0	0	1	1	1	1	0	0
0	0	1	1	1	0	0	0





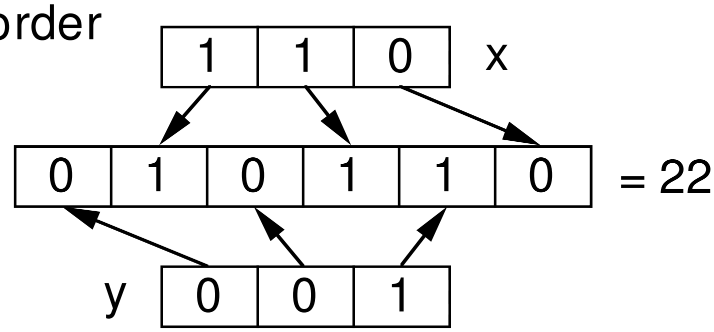
Ordering Space

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



CONVERTING BETWEEN POINTS AND CURVES

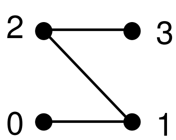
- Need to know size of image for all but the Morton order
- Relatively easy for all but the Peano-Hilbert order which is difficult (although possible) to decode and encode to obtain the corresponding *x* and *y* coordinate values
- Morton order
  1. use bit interleaving of binary representation of the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



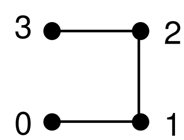
STABILITY OF SPACE ORDERING METHODS

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

Morton:



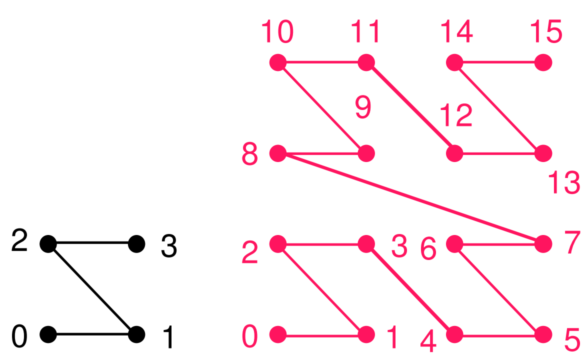
Peano-Hilbert:



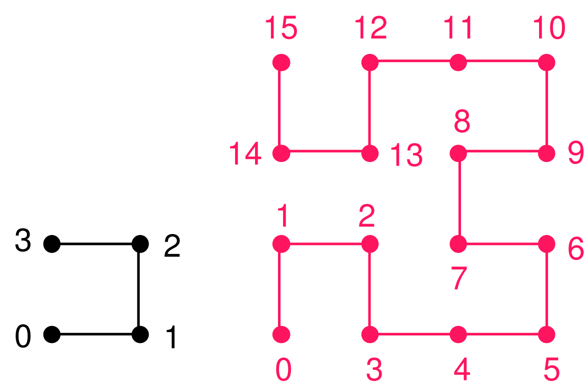
STABILITY OF SPACE ORDERING METHODS

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

Morton:



Peano-Hilbert:

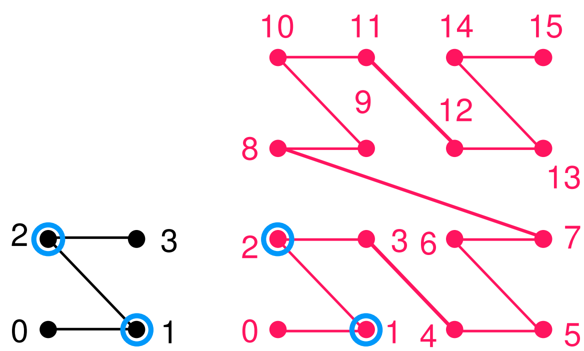


- Result of doubling the resolution (i.e., the coverage)

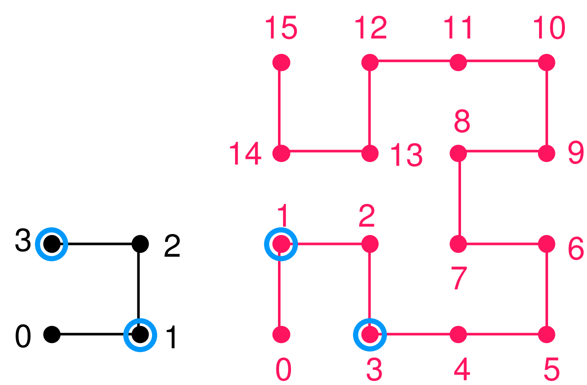
STABILITY OF SPACE ORDERING METHODS

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

Morton:



Peano-Hilbert:



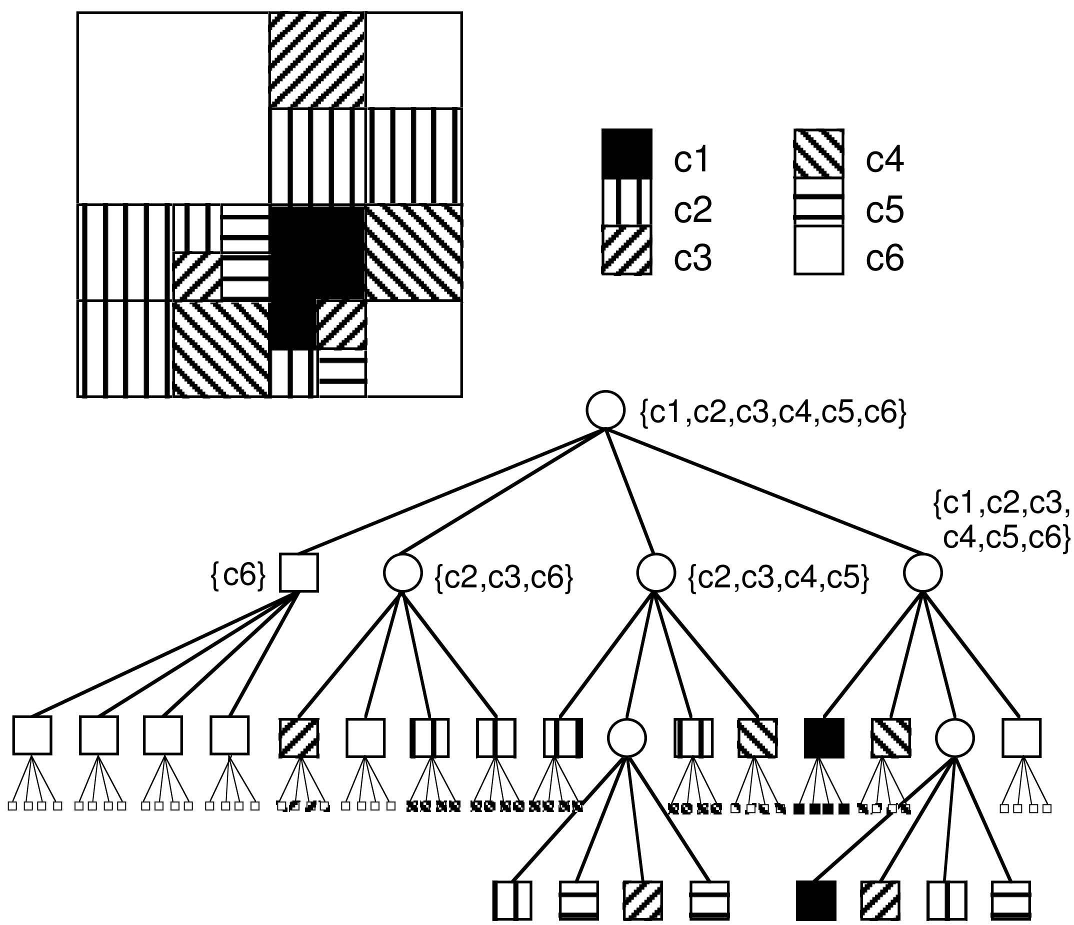
- Result of doubling the resolution (i.e., the coverage) in which case the circled points do not maintain the same relative order in the Peano-Hilbert order while they do in the Morton order

DESIRABLE PROPERTIES OF SPACE FILLING CURVES

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

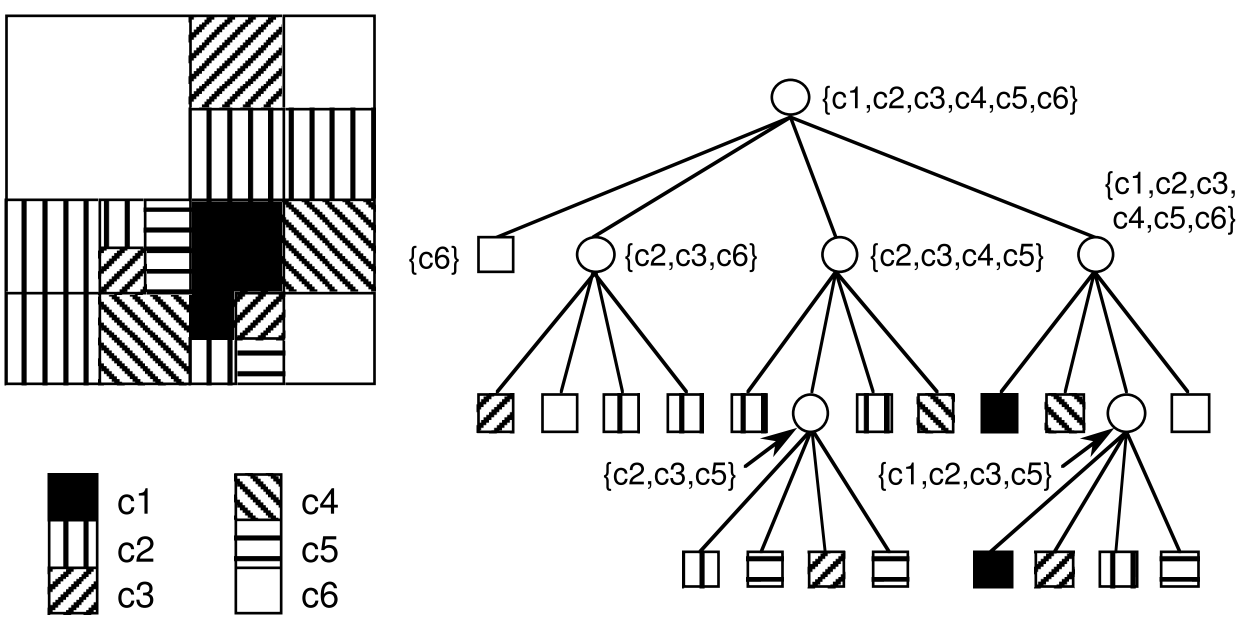
PYRAMID

- 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



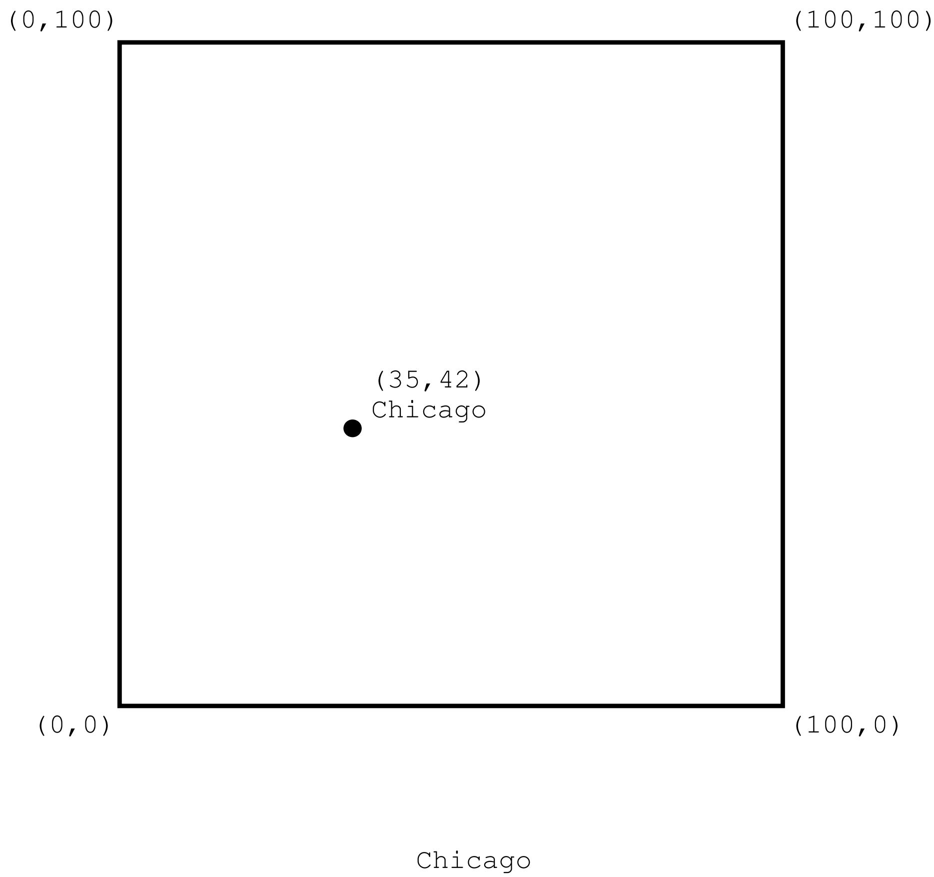
- Can represent as a list of leaf and nonleaf blocks (e.g., as a linear quadtree)

Outline

1. Introduction
2. Points
3. Lines
4. Regions, Volumes, and Surfaces
5. Bounding Box Hierarchies
6. Rectangles
7. Surfaces and Volumes
8. Metric Data
9. Operations
10. Indexing Spatiotextual Data
11. Example system

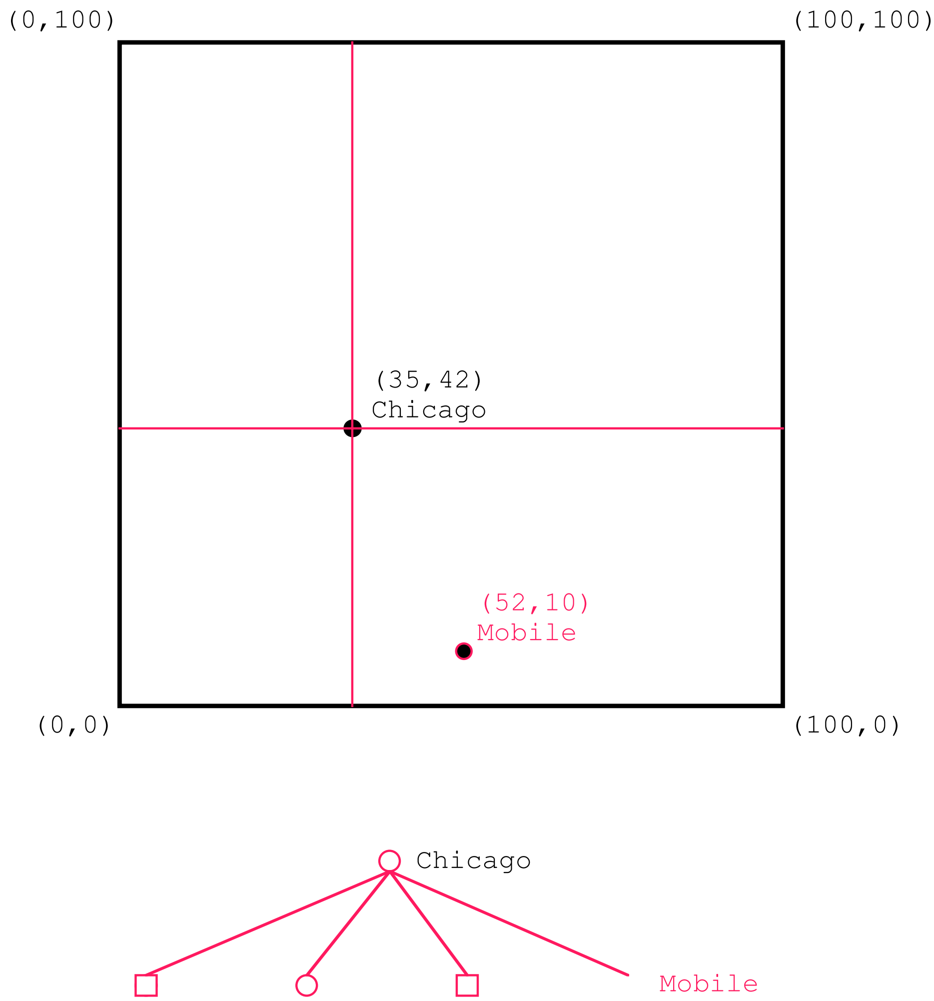
POINT QUADTREE (Finkel/Bentley)

- Marriage between a uniform grid and a binary search tree



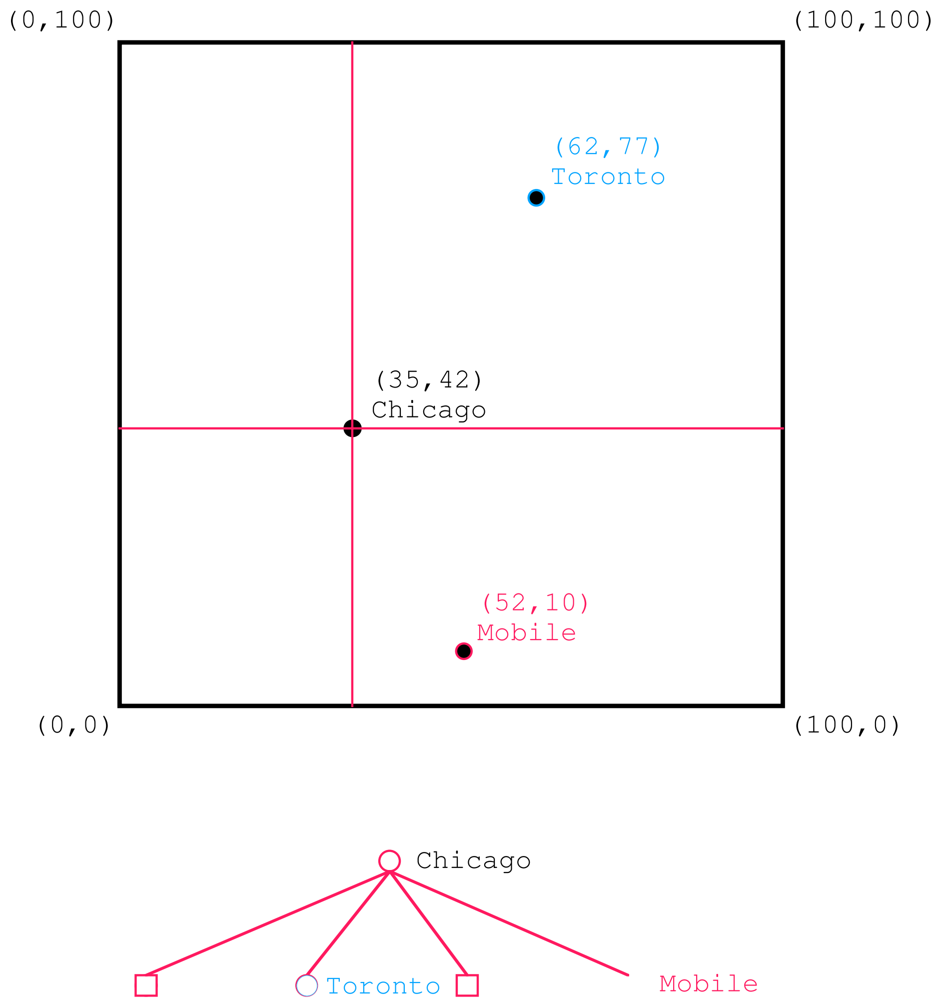
POINT QUADTREE (Finkel/Bentley)

- Marriage between a uniform grid and a binary search tree



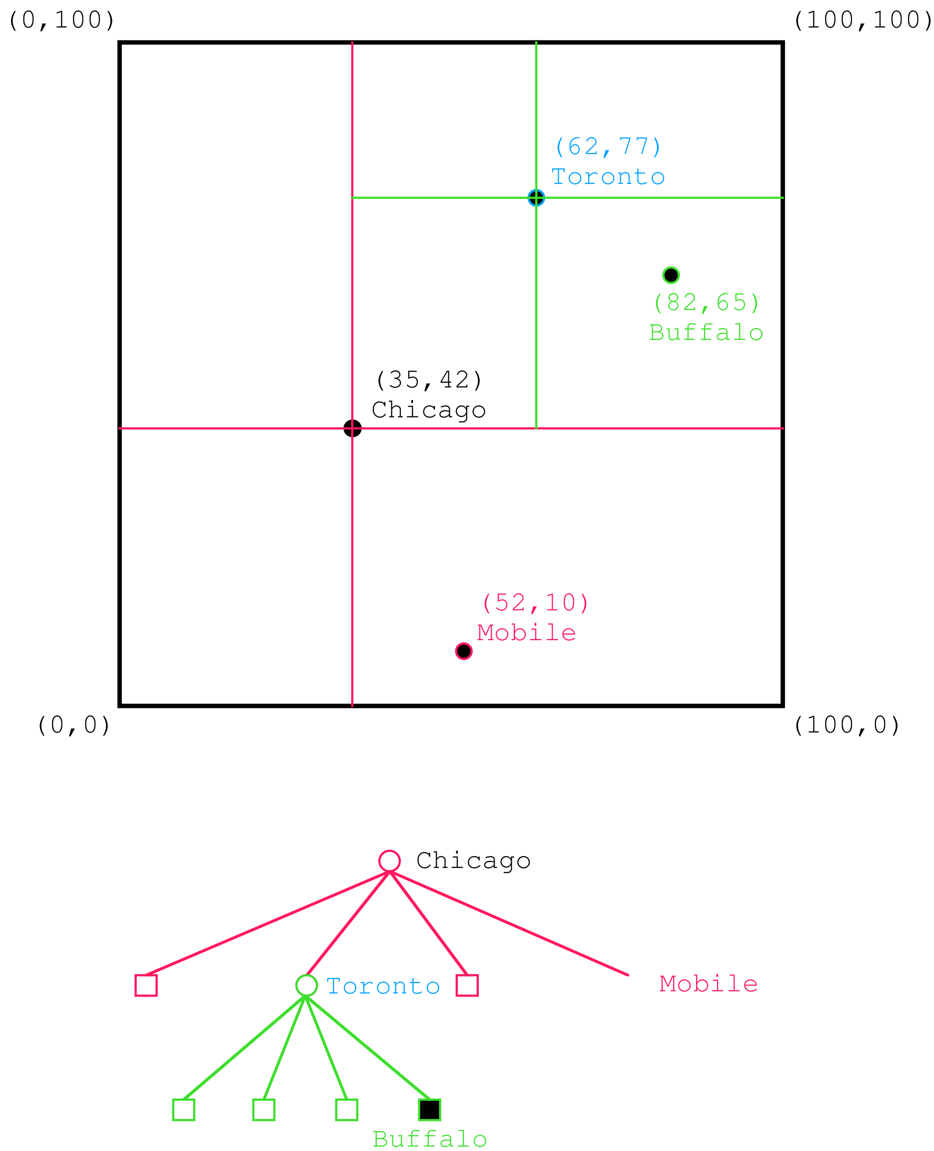
POINT QUADTREE (Finkel/Bentley)

- Marriage between a uniform grid and a binary search tree



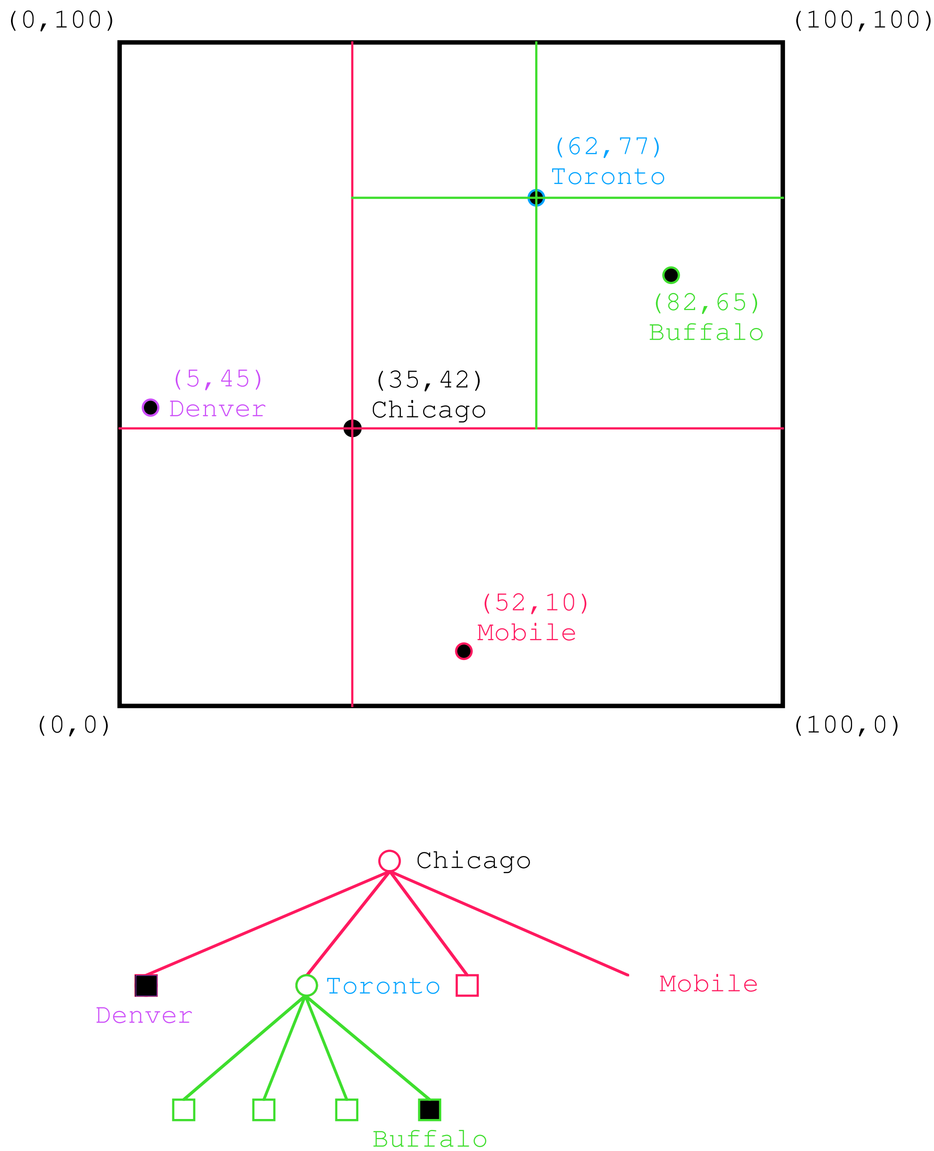
POINT QUADTREE (Finkel/Bentley)

- Marriage between a uniform grid and a binary search tree



POINT QUADTREE (Finkel/Bentley)

- Marriage between a uniform grid and a binary search tree

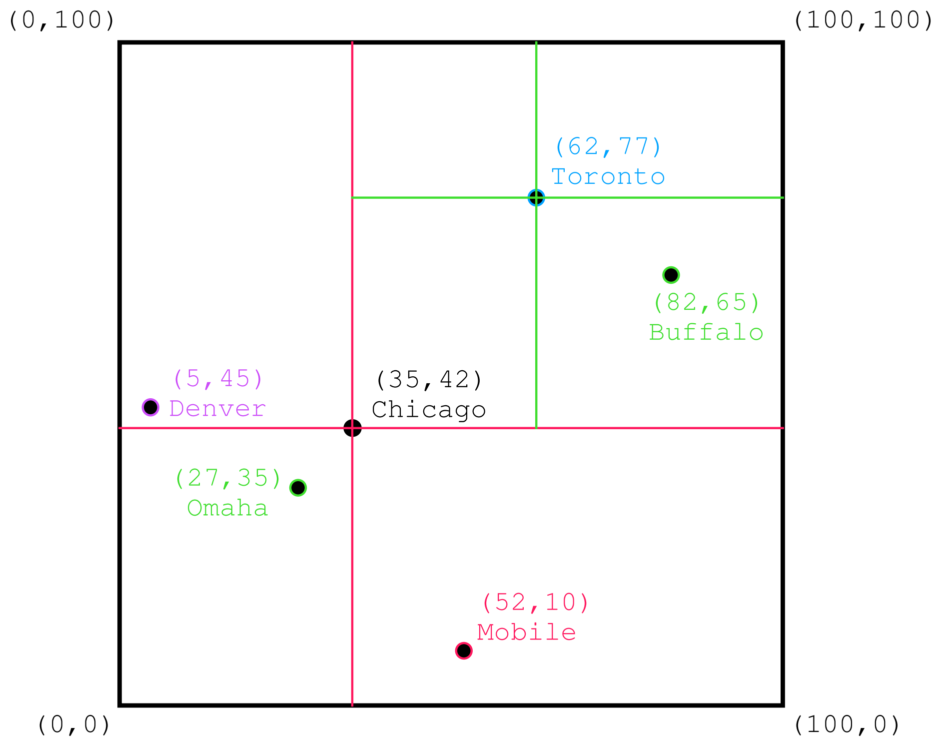


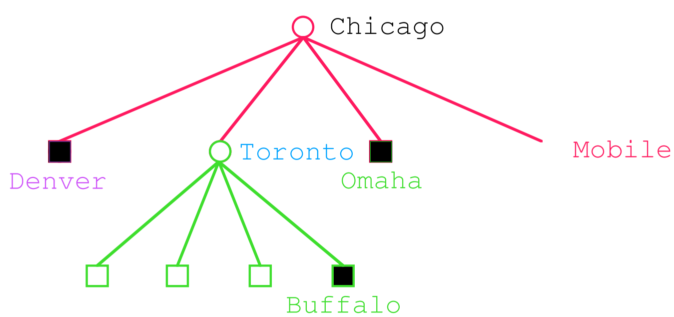
POINT QUADTREE (Finkel/Bentley)

6 5 4 3 2 1
g v g z r b

hp4

- Marriage between a uniform grid and a binary search tree



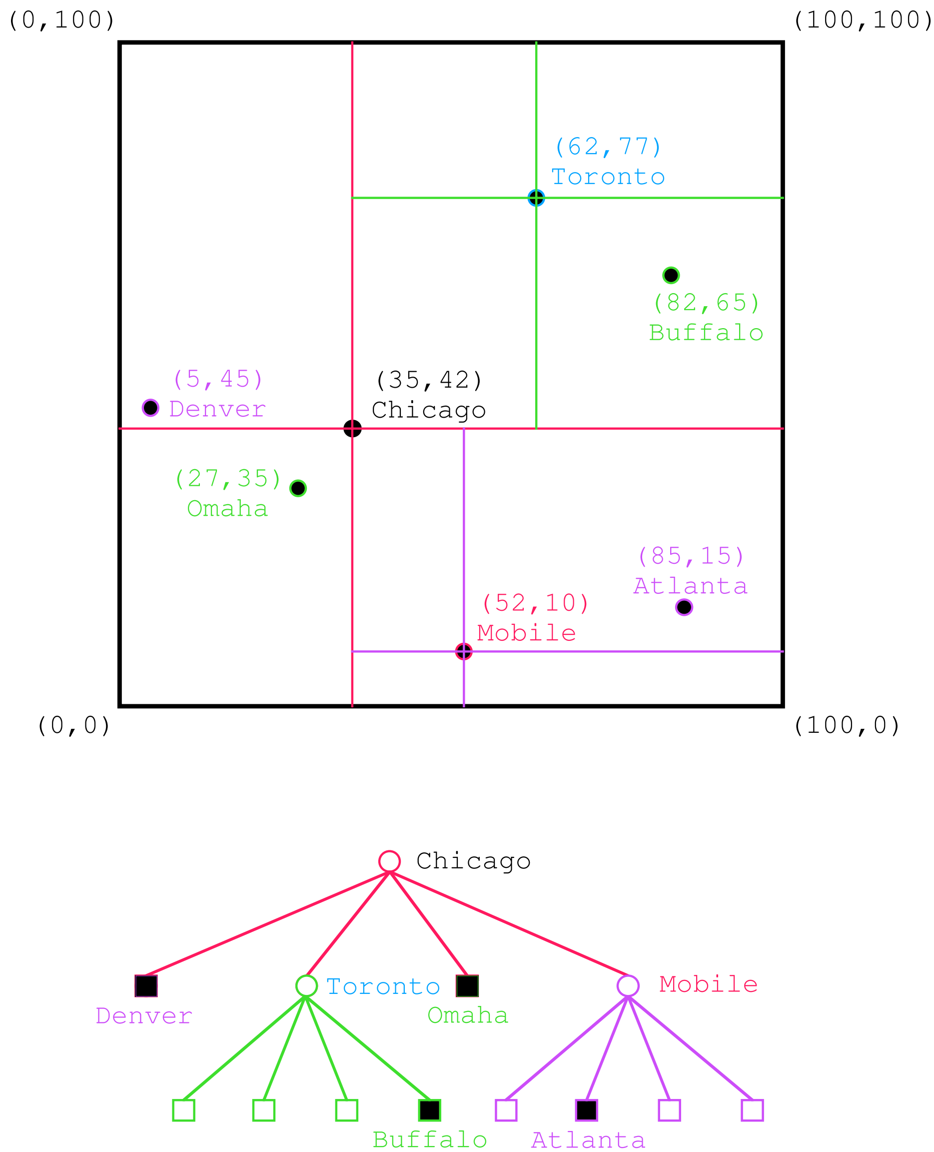


POINT QUADTREE (Finkel/Bentley)

7 6 5 4 3 2 1
v g v g z r b

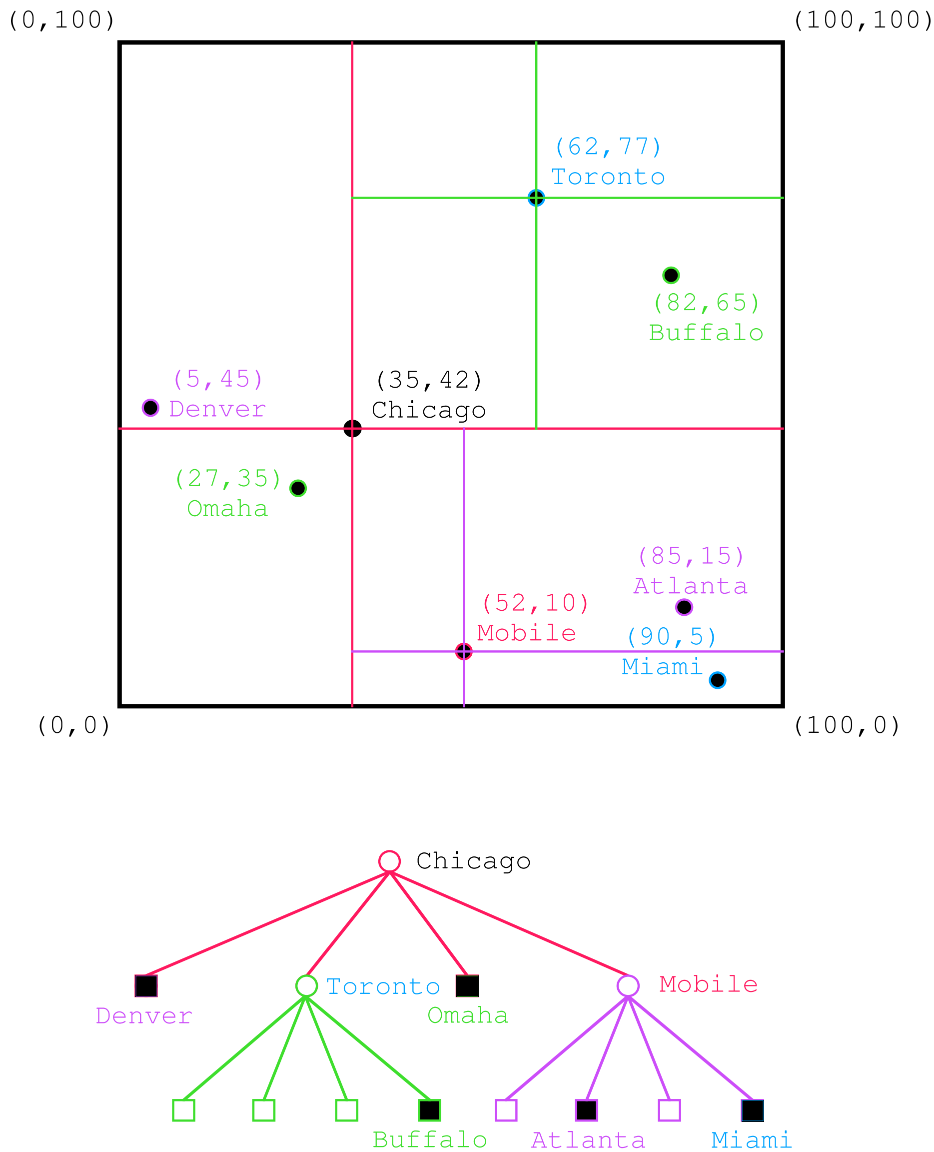
hp4

- Marriage between a uniform grid and a binary search tree



POINT QUADTREE (Finkel/Bentley)

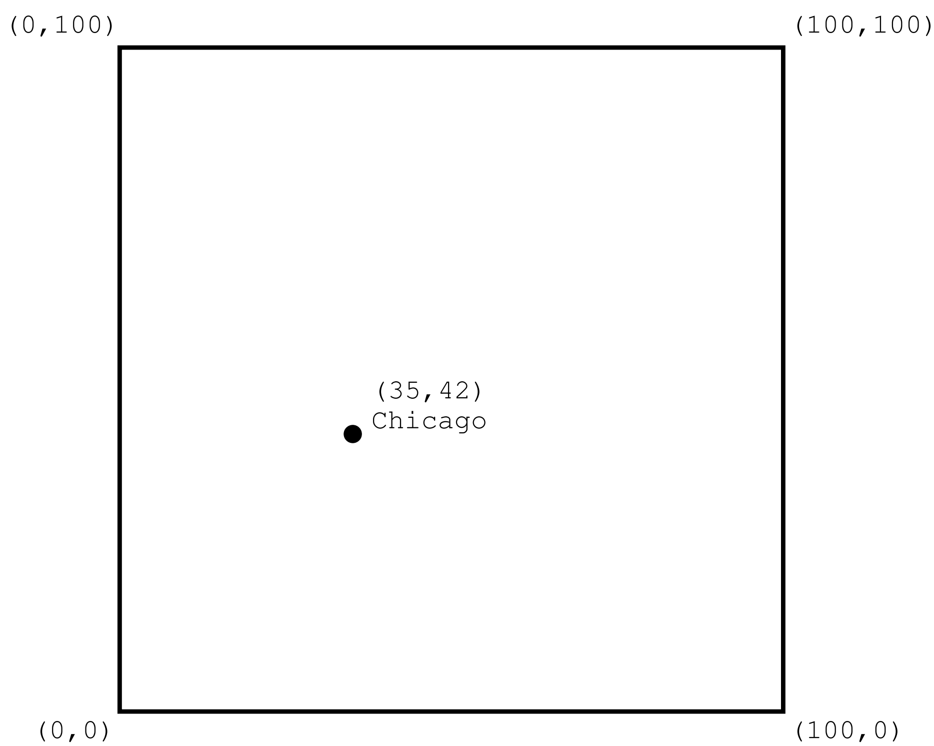
- Marriage between a uniform grid and a binary search tree



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

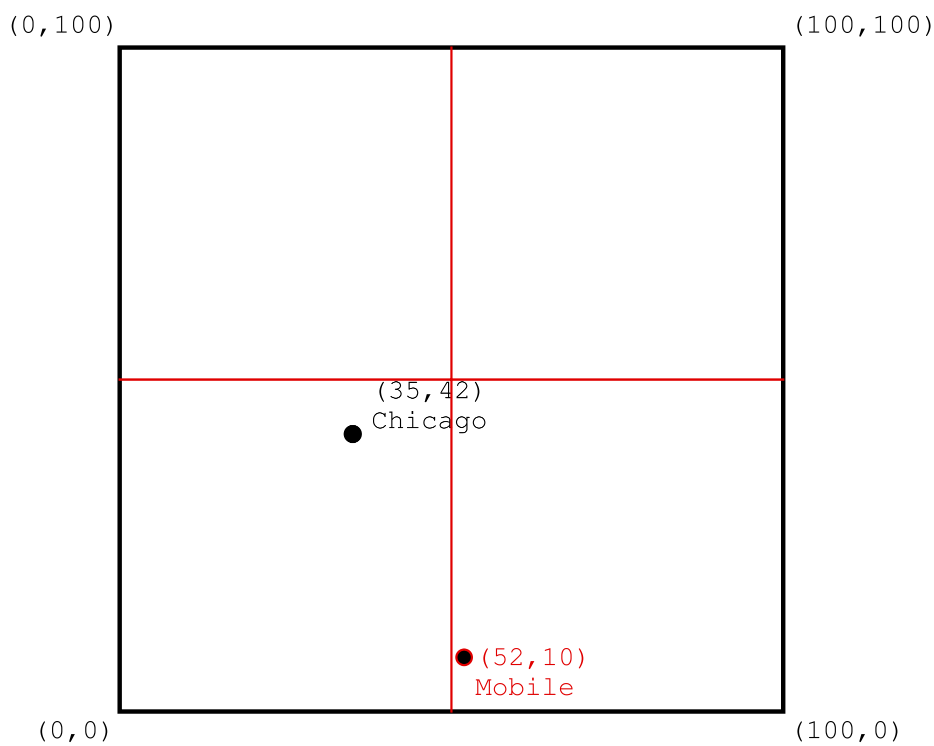
Ex: $c = 1$



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

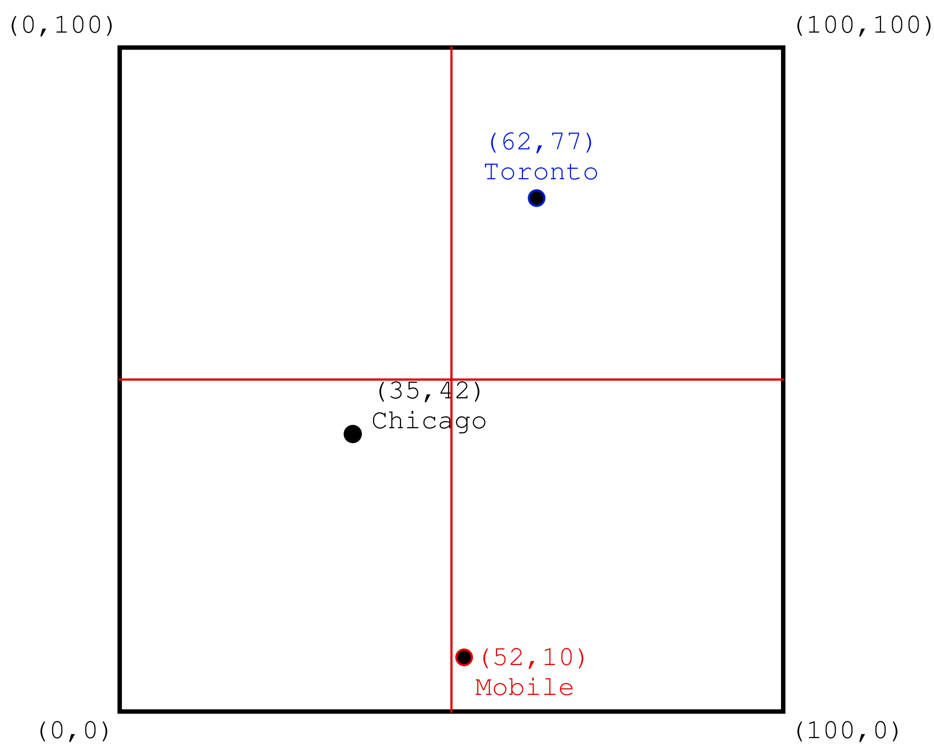
Ex: $c = 1$



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

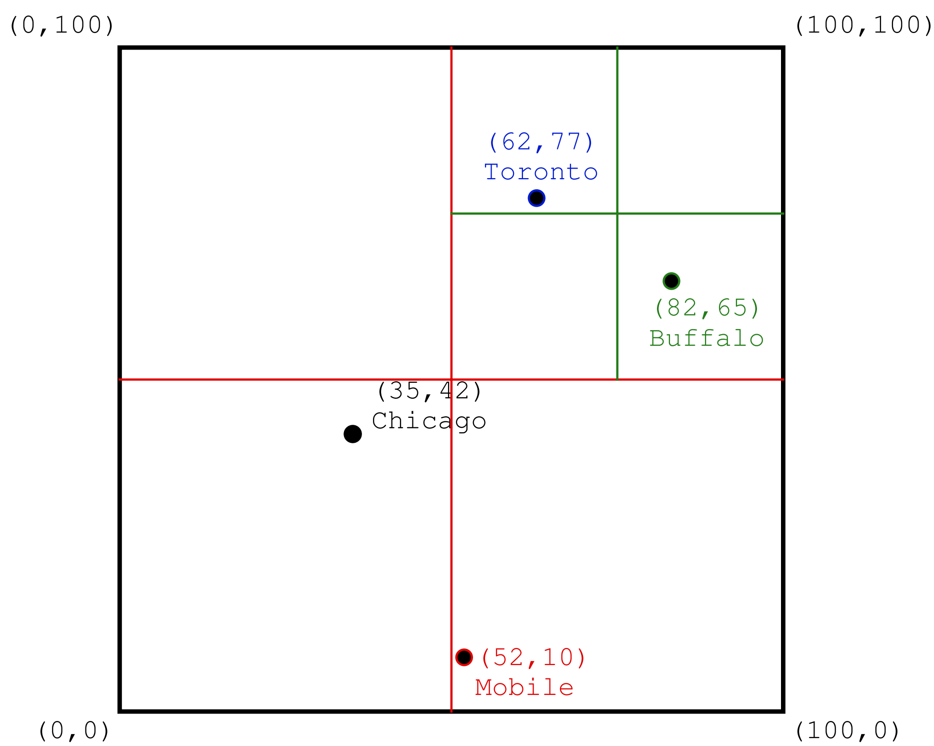
Ex: $c = 1$



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

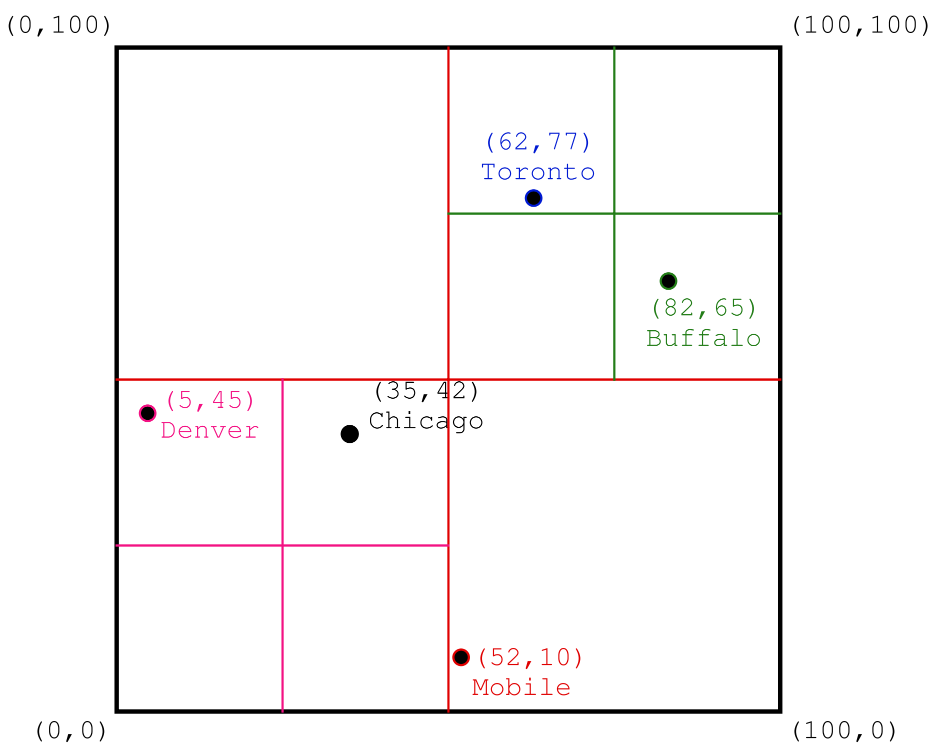
Ex: $c = 1$



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

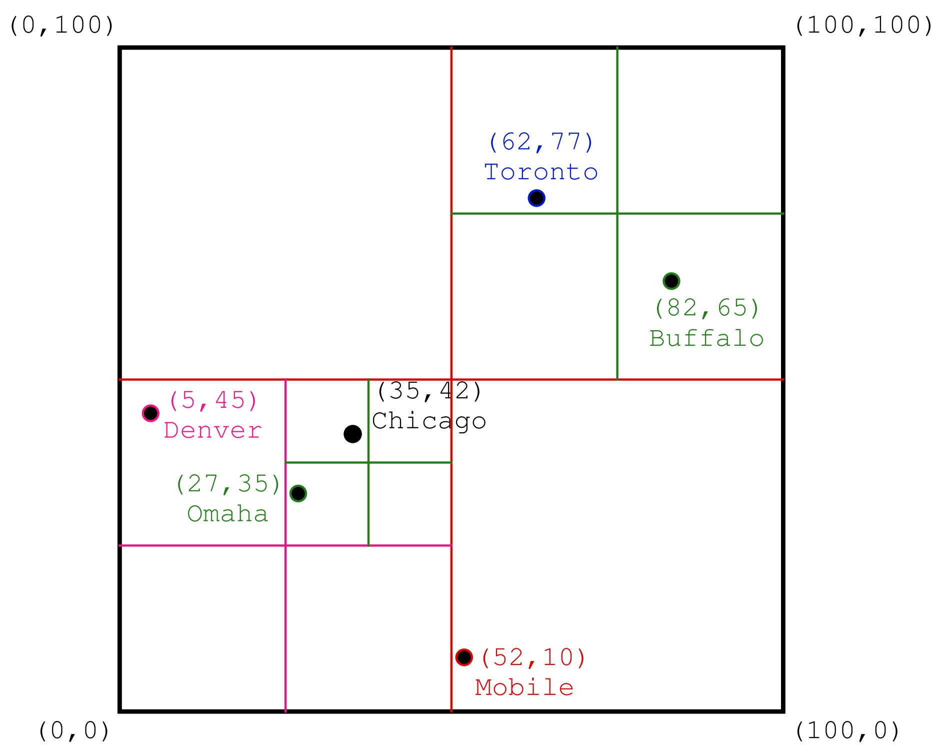
Ex: $c = 1$



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

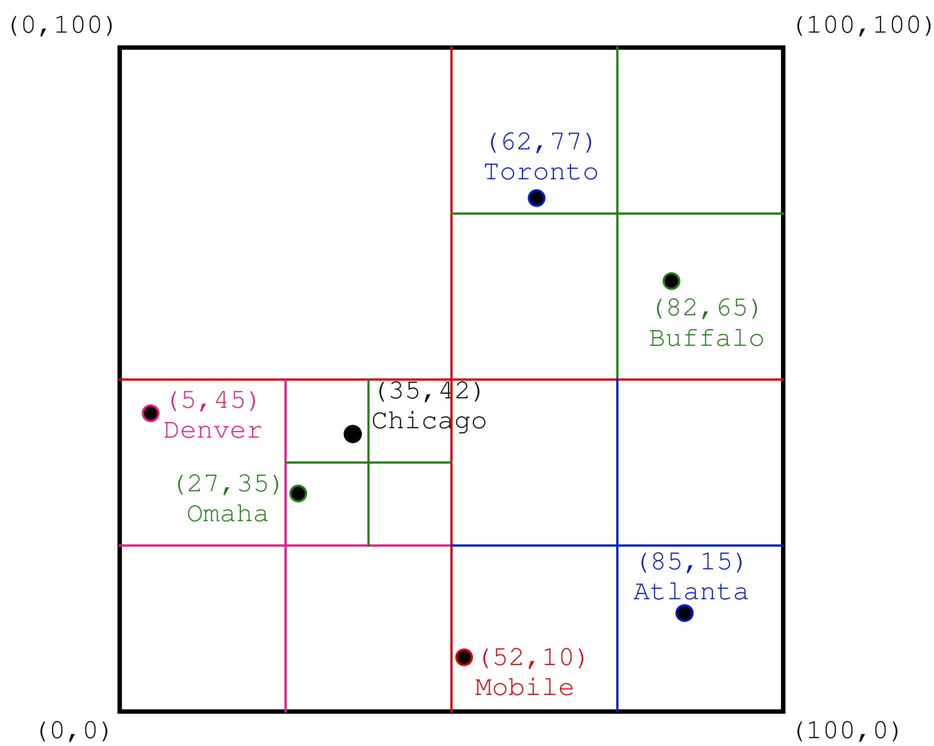
Ex: $c = 1$



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

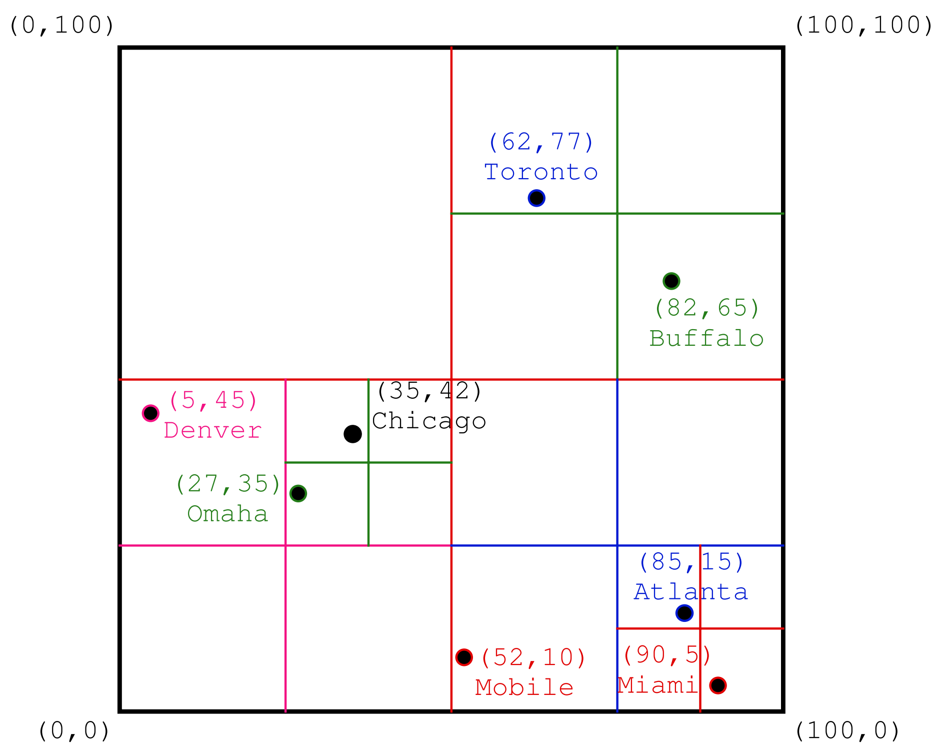
Ex: $c = 1$



PR QUADTREE (Orenstein)

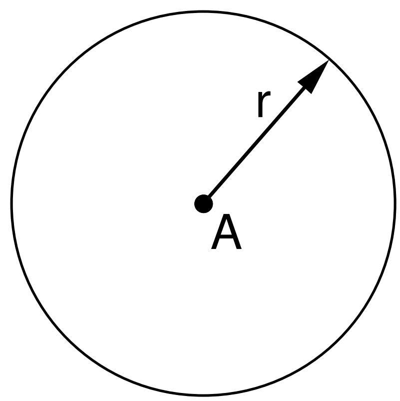
1. Regular decomposition point representation
2. Decomposition occurs whenever a block contains more than one point
3. Useful when the domain of data points is not discrete but finite
4. Maximum level of decomposition depends on the minimum separation between two points
   - if two points are very close,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

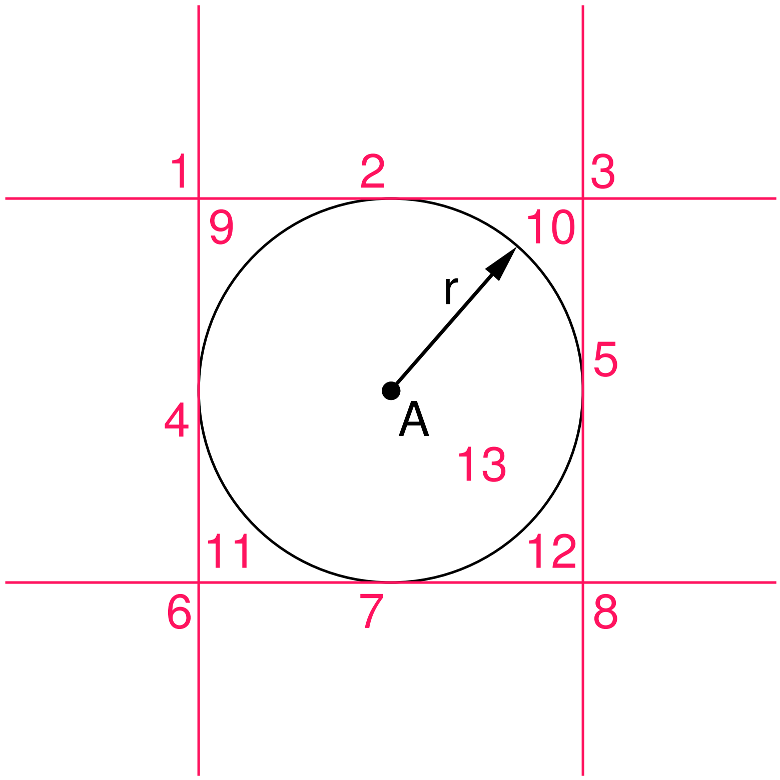
- Ex: Find all points within radius *r* of point A



- Use of quadtree results in pruning the search space

REGION SEARCH

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

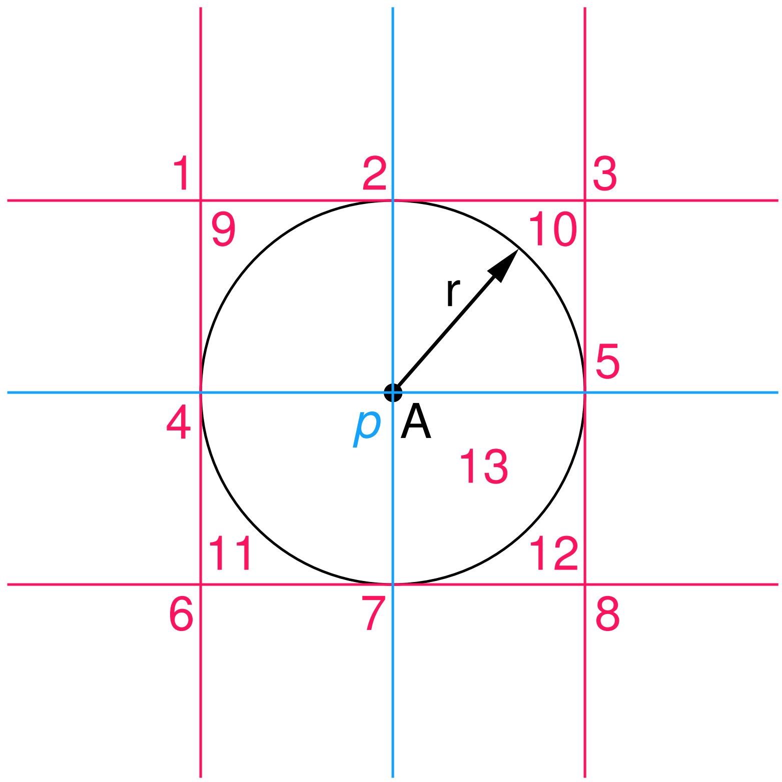


- Use of quadtree results in pruning the search space
- If a quadrant subdivision point *p* lies in a region *l*, then search the quadrants of *p* specified by *l*

1. SE
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

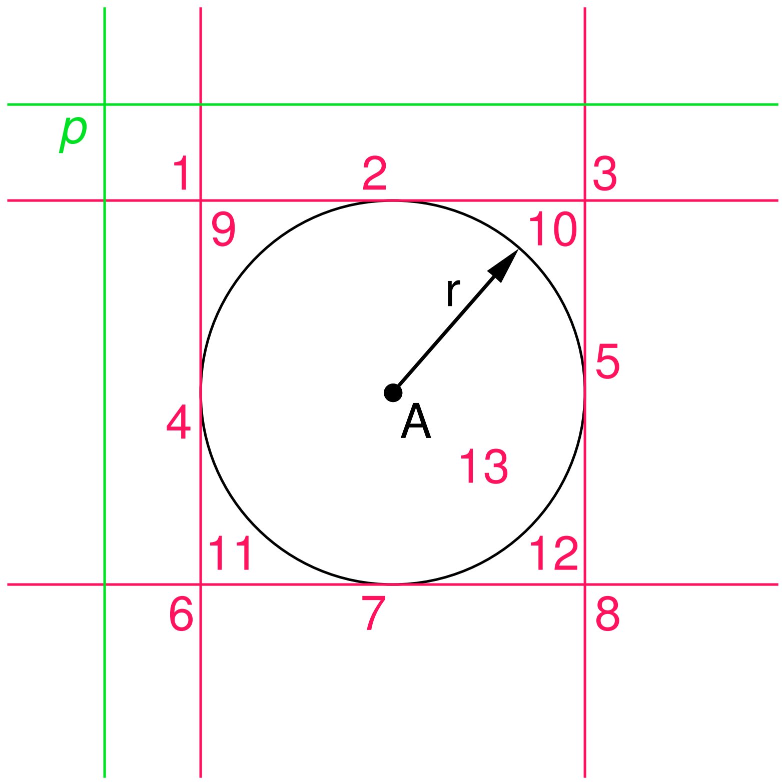


- Use of quadtree results in pruning the search space
- If a quadrant subdivision point *p* lies in a region *l*, then search the quadrants of *p* specified by *l*

1. SE
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

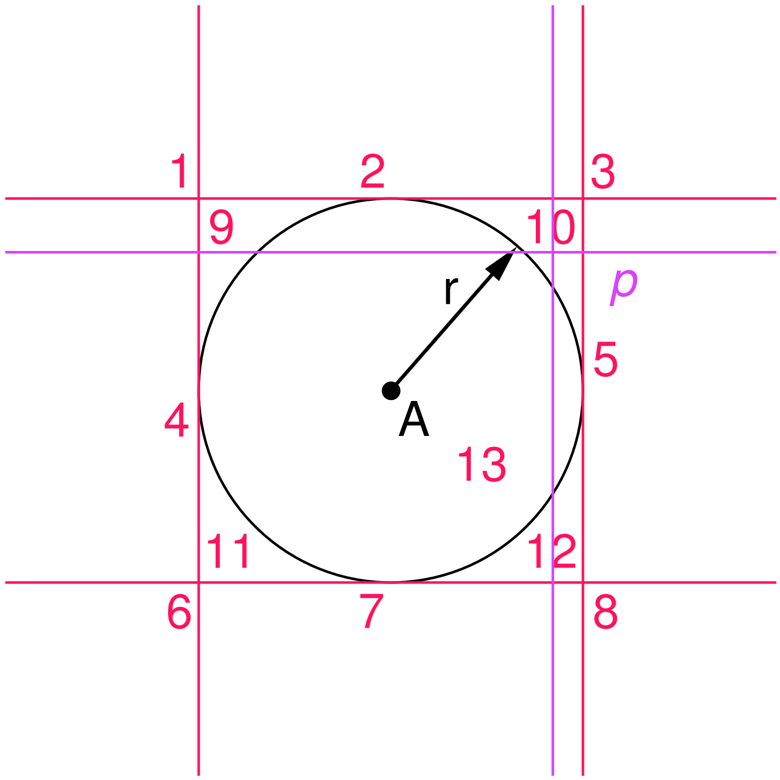


- Use of quadtree results in pruning the search space
- If a quadrant subdivision point *p* lies in a region *l*, then search the quadrants of *p* specified by *l*

1. SE
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

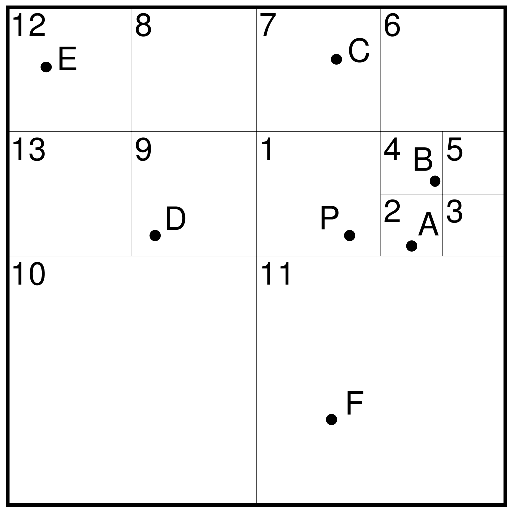


- Use of quadtree results in pruning the search space
- If a quadrant subdivision point *p* lies in a region *l*, then search the quadrants of *p* specified by *l*

1. SE
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

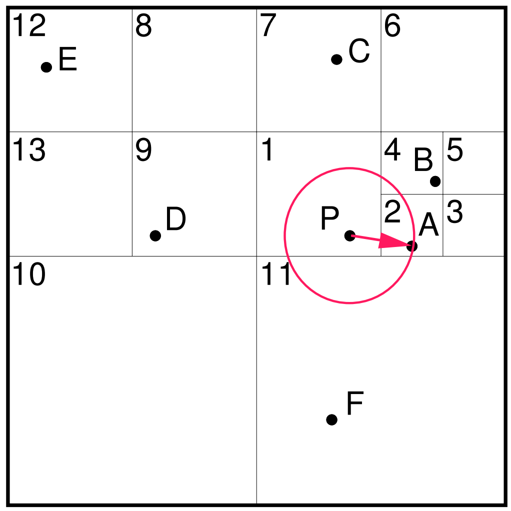
- Ex: find the nearest object to P



- 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

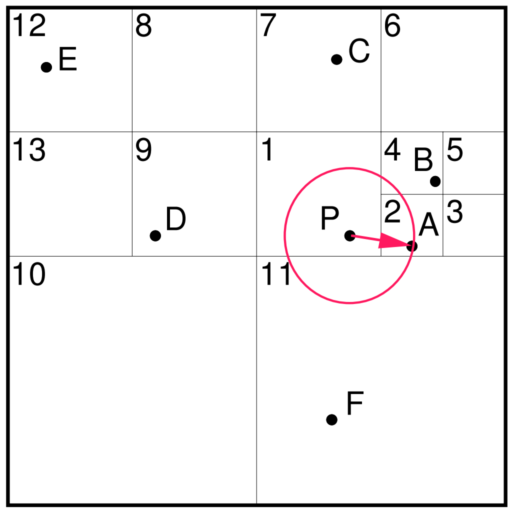


- Assume PR quadtree for points (i.e., at most one point per block)
- Search neighbors of block 1 in counterclockwise order
- Points are sorted with respect to the space they occupy which enables pruning the search space
- Algorithm:

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

FINDING THE NEAREST OBJECT

- Ex: find the nearest object to P

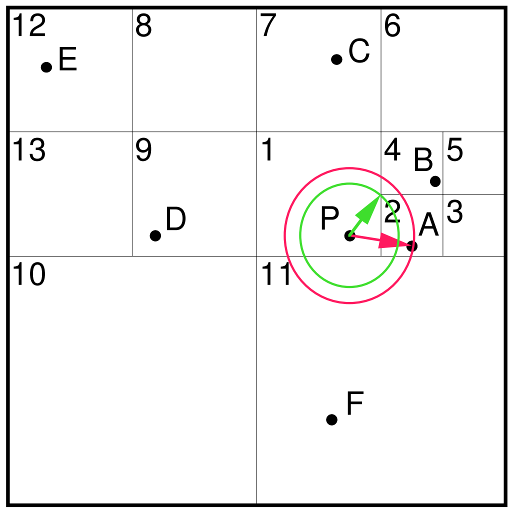


- Assume PR quadtree for points (i.e., at most one point per block)
- Search neighbors of block 1 in counterclockwise order
- Points are sorted with respect to the space they occupy which enables pruning the search space
- Algorithm:

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

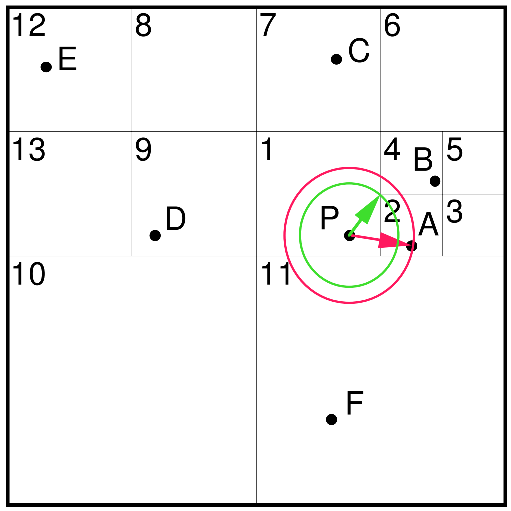
- Ex: find the nearest object to P



- 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

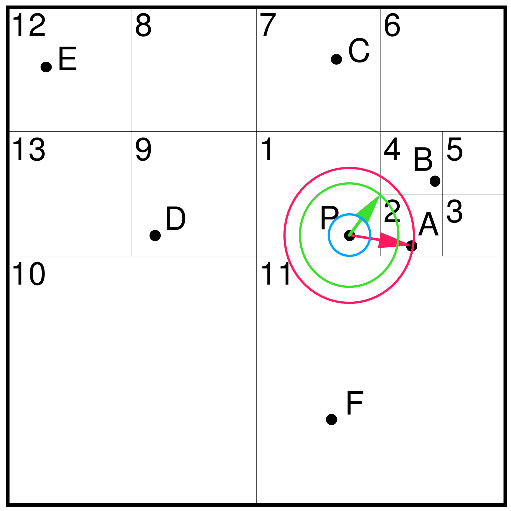
- Ex: find the nearest object to P



- 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

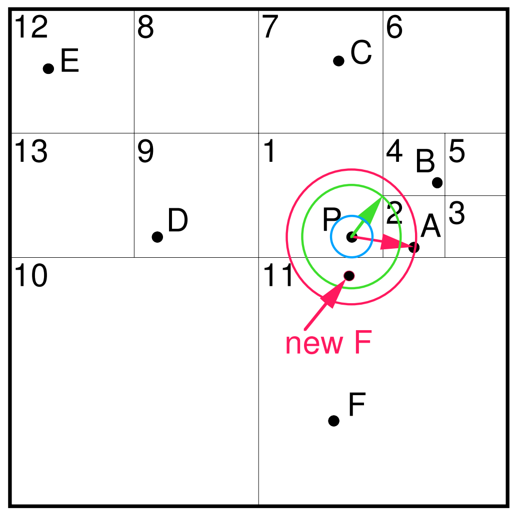


- Assume PR quadtree for points (i.e., at most one point per block)
- Search neighbors of block 1 in counterclockwise order
- Points are sorted with respect to the space they occupy which enables pruning the search space
- Algorithm:

  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

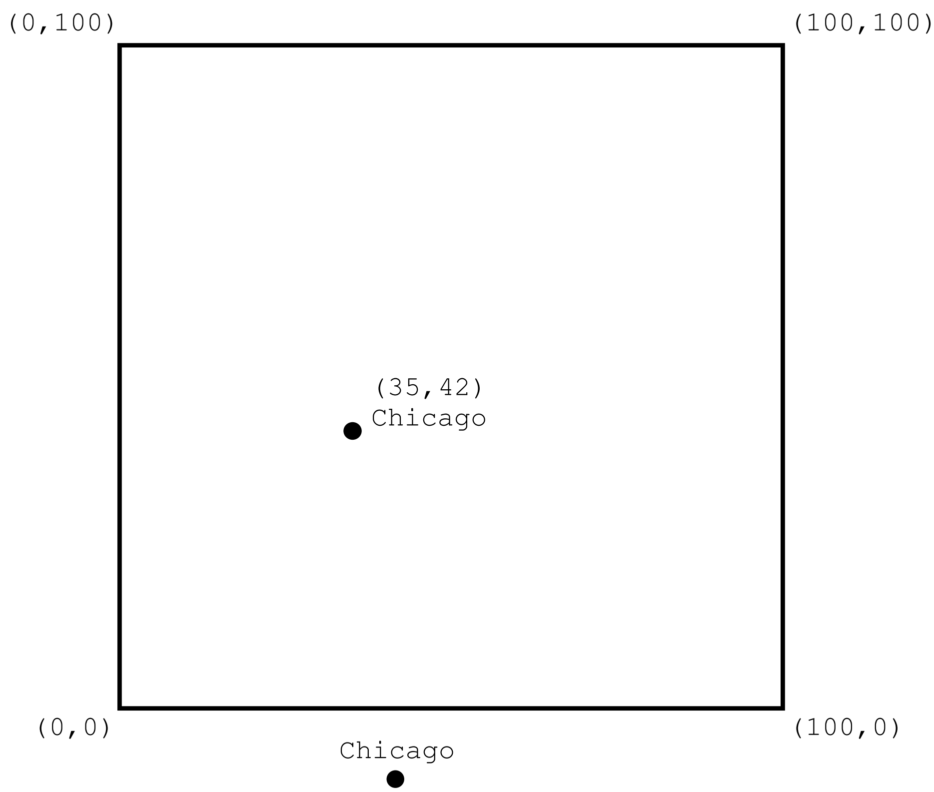
- Ex: find the nearest object to P



- 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

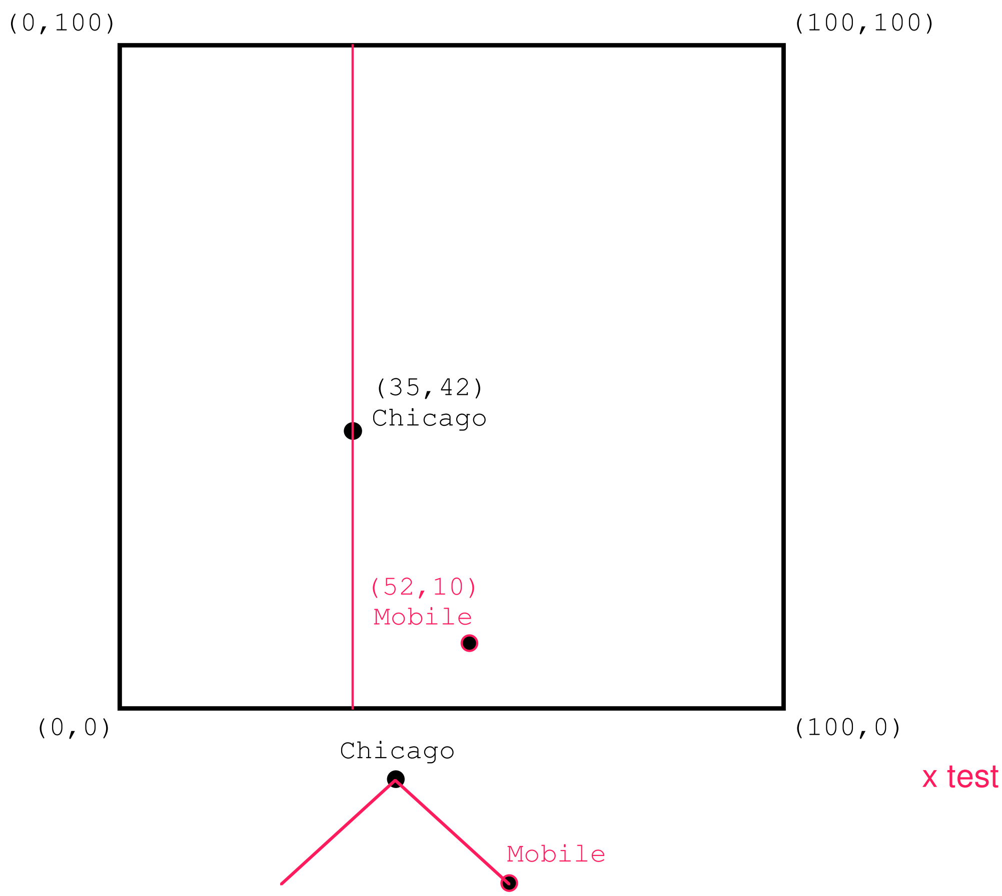
K-D TREE (Bentley)

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



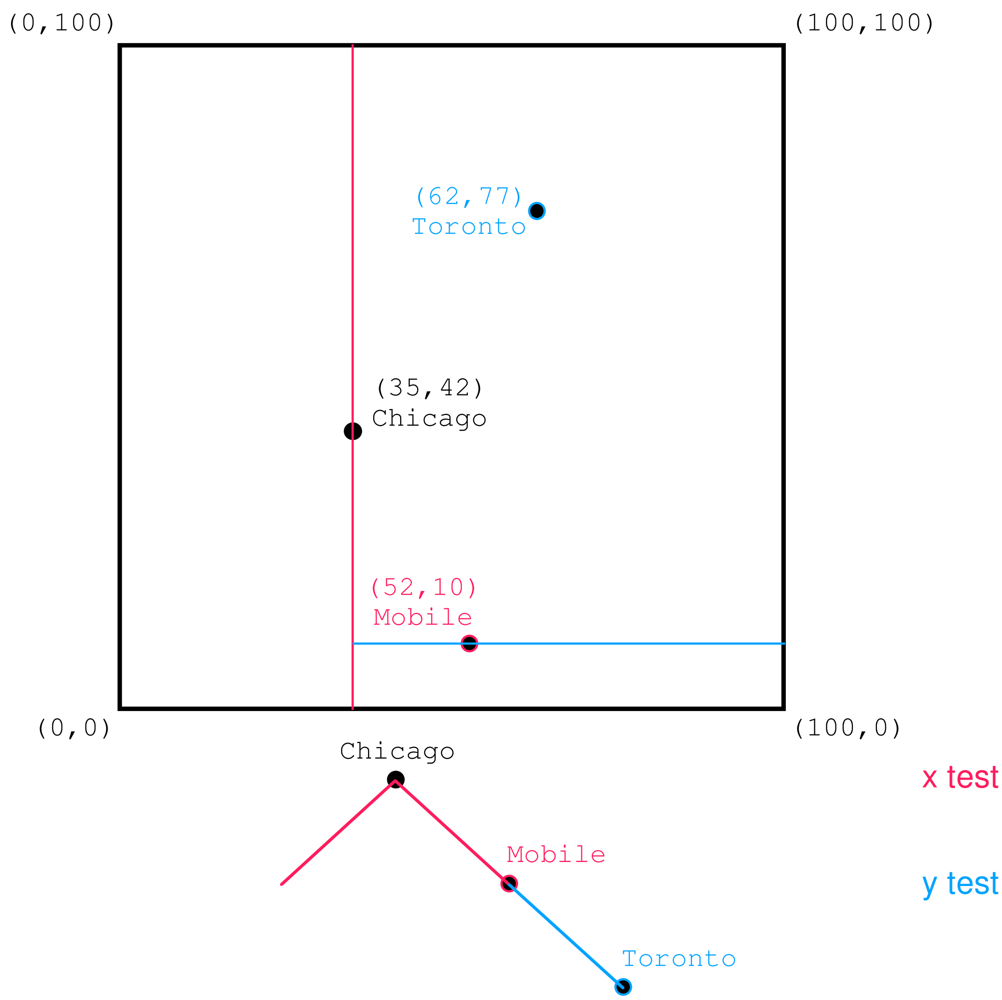
K-D TREE (Bentley)

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



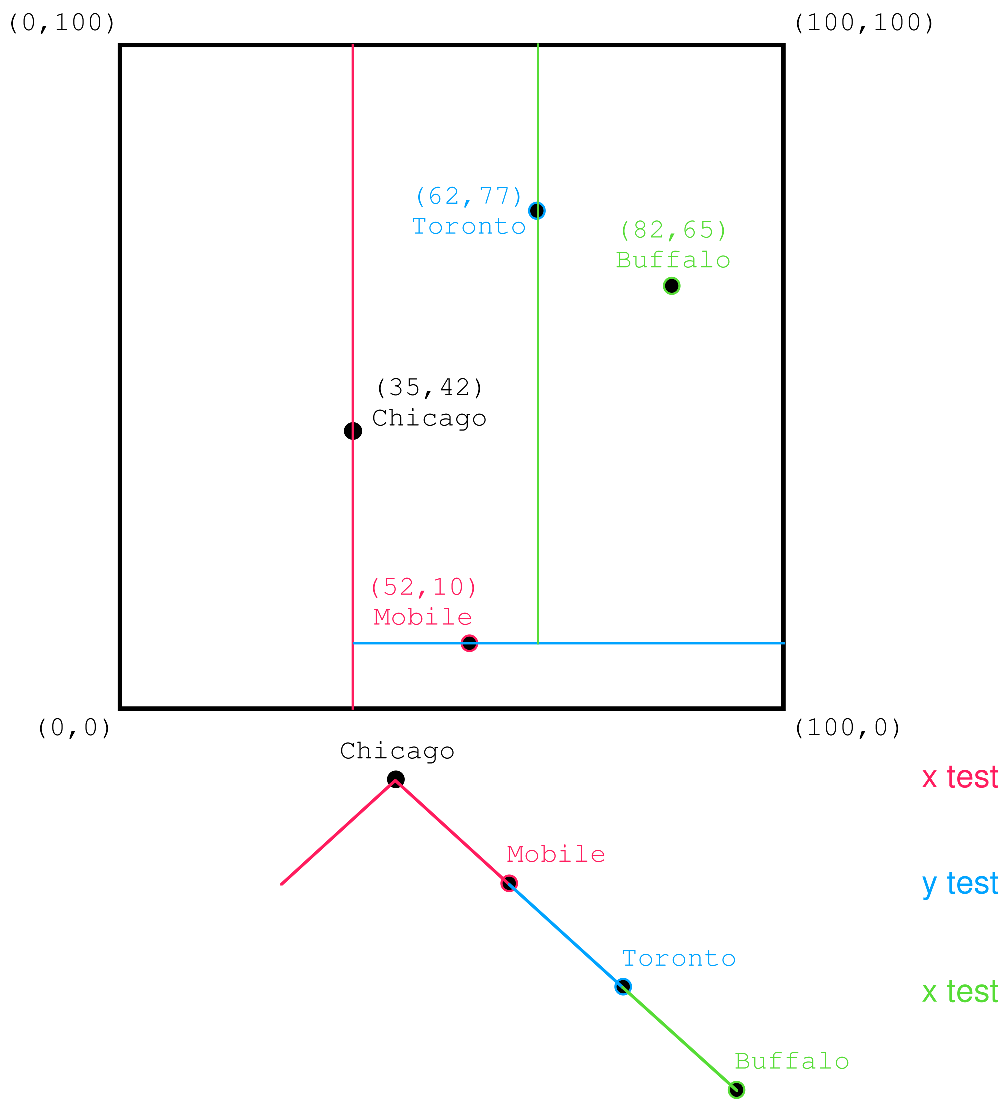
K-D TREE (Bentley)

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



K-D TREE (Bentley)

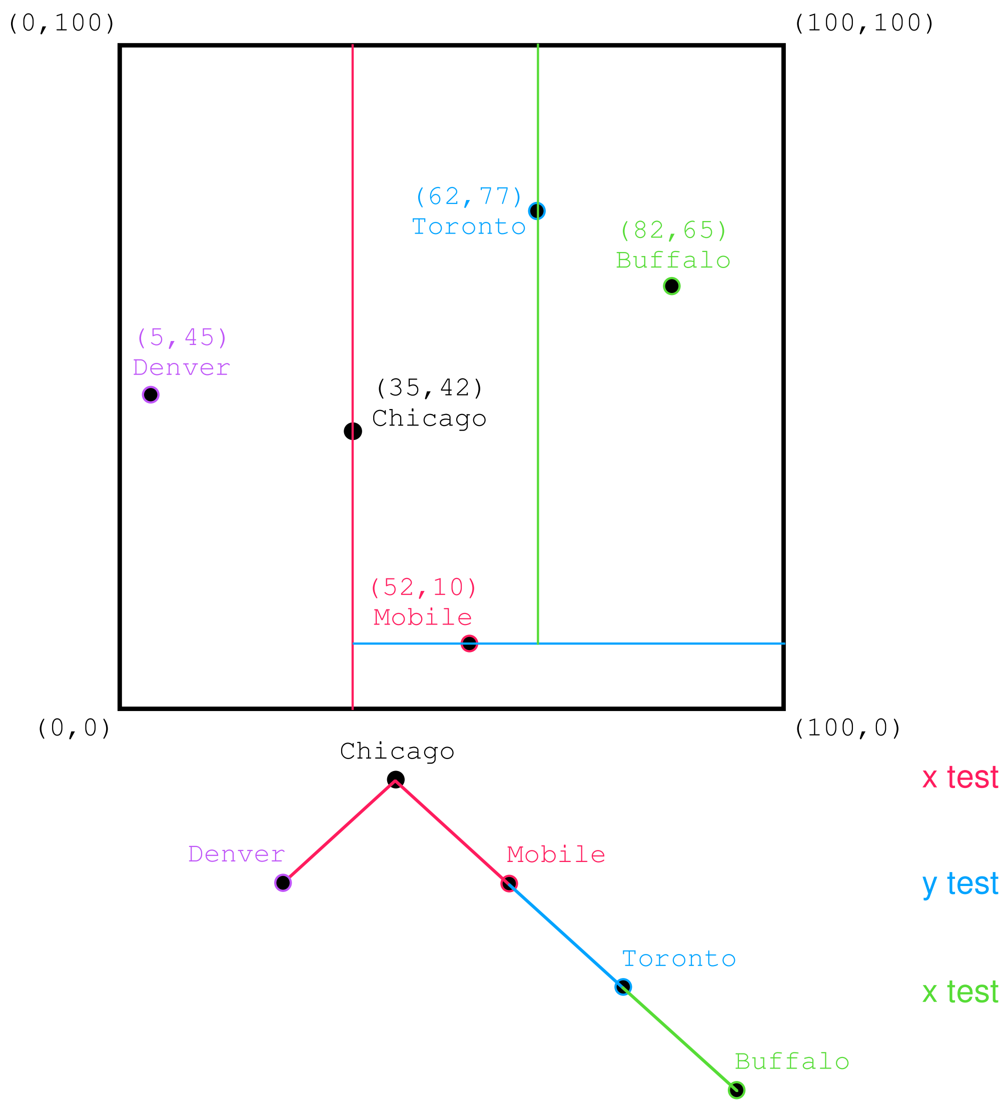
- Test one attribute at a time instead of all simultaneously as in the point quadtree
- Usually cycle through all the attributes
- Shape of the tree depends on the order in which the data is encountered



K-D TREE (Bentley)

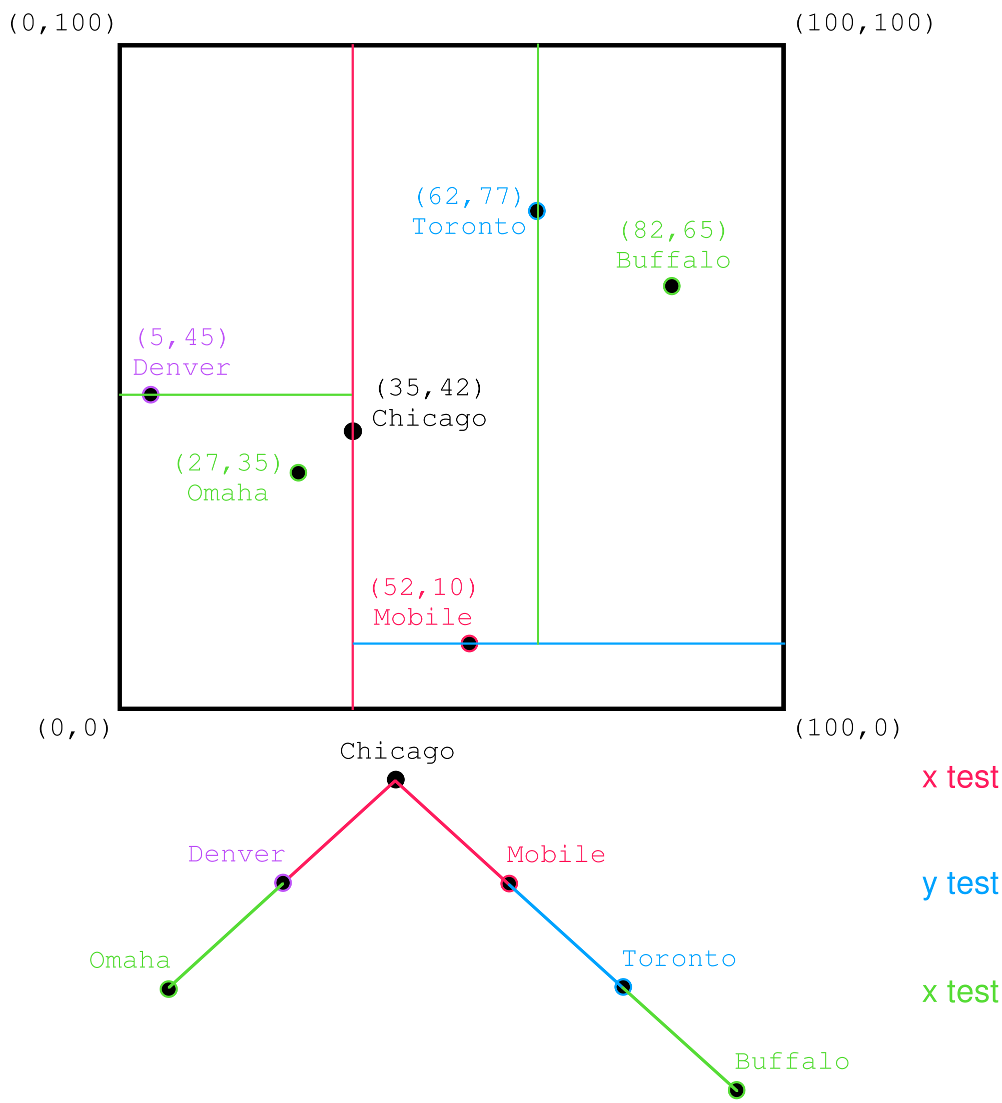
5 4 3 2 1
v g z r b
hp15

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



K-D TREE (Bentley)

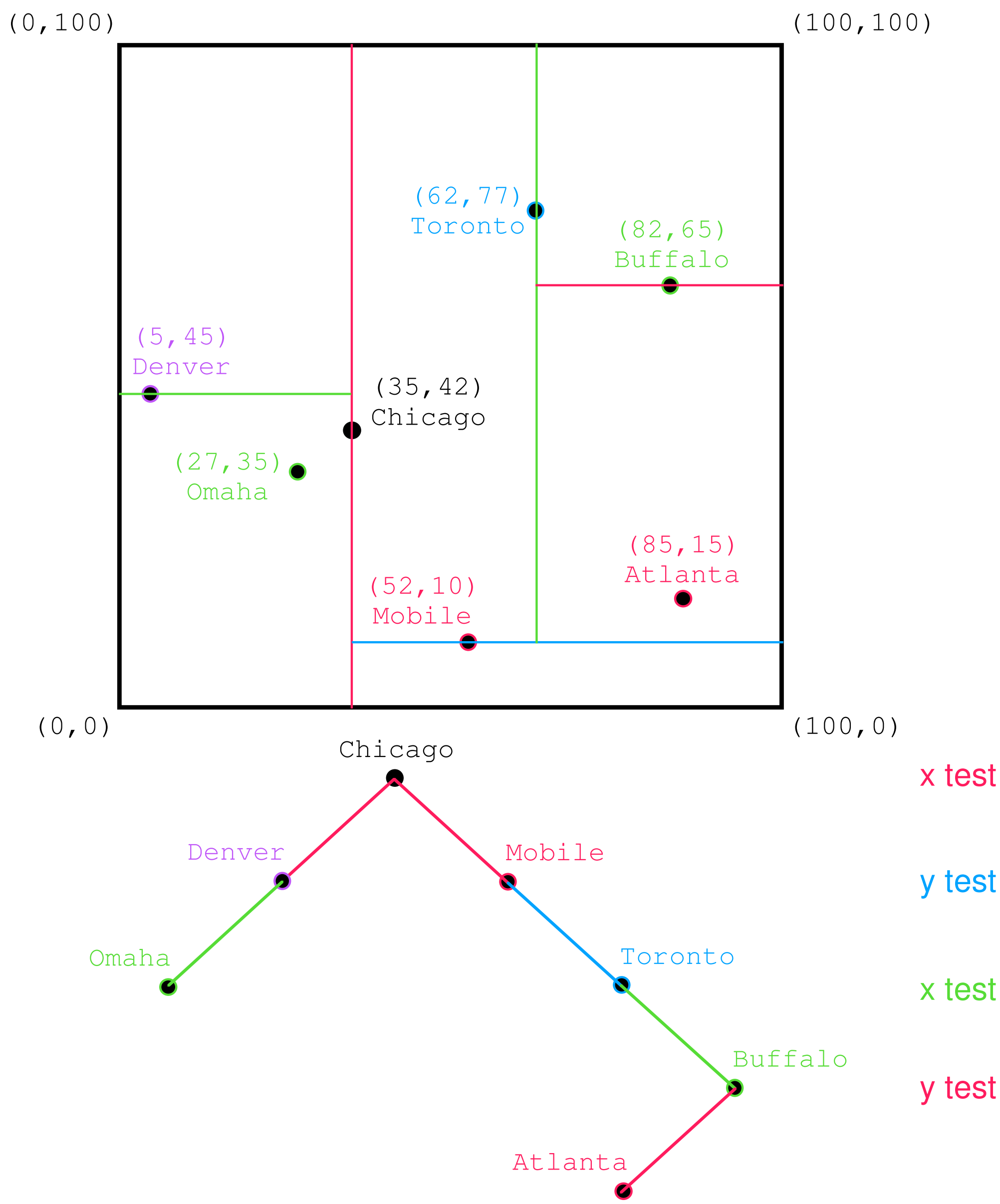
- Test one attribute at a time instead of all simultaneously as in the point quadtree
- Usually cycle through all the attributes
- Shape of the tree depends on the order in which the data is encountered



7 6 5 4 3 2 1
r g v g z r b
hp15

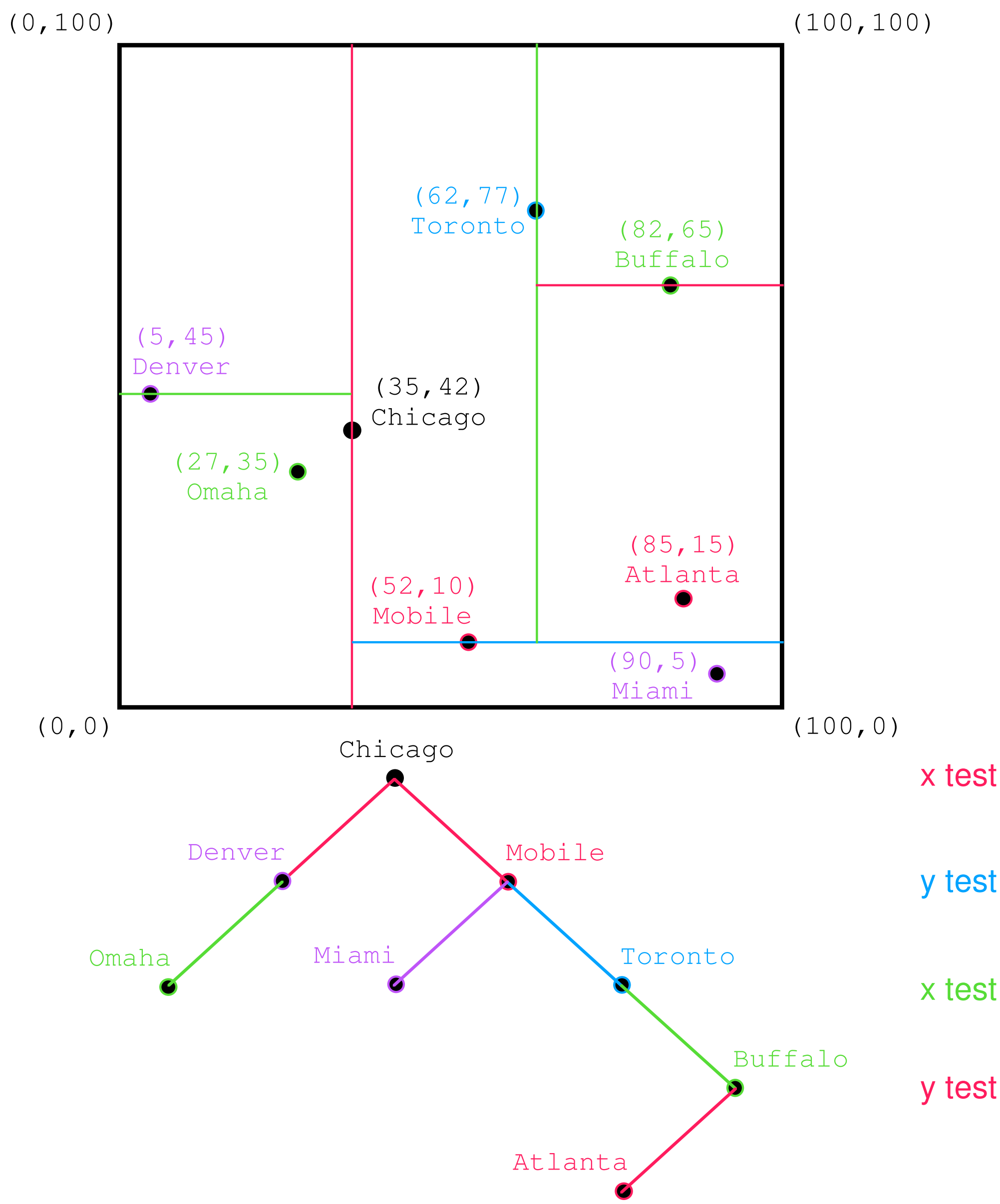
K-D TREE (Bentley)

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



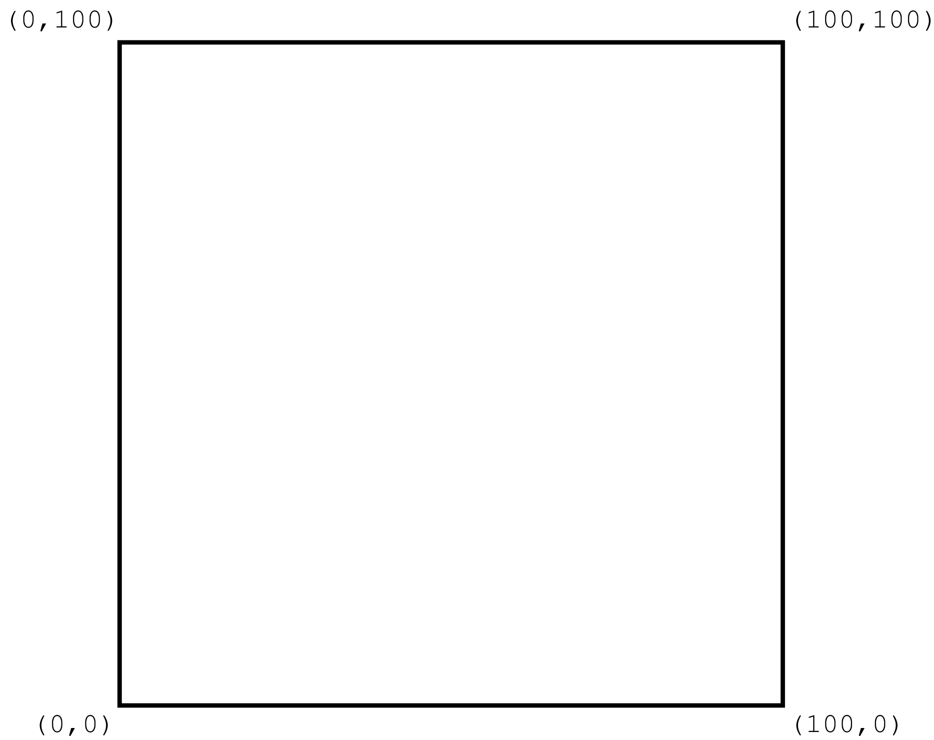
K-D TREE (Bentley)

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



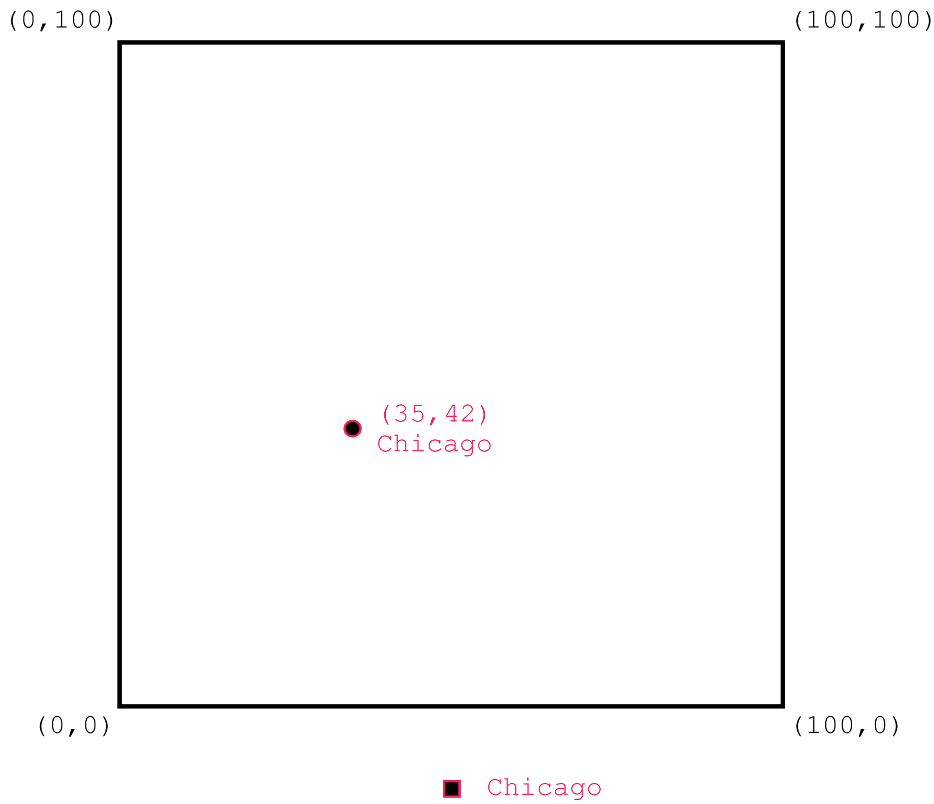
PR K-D TREE (Knowlton)

- A region contains at most one data point
- Analogous to EXCELL with bucket size of 1



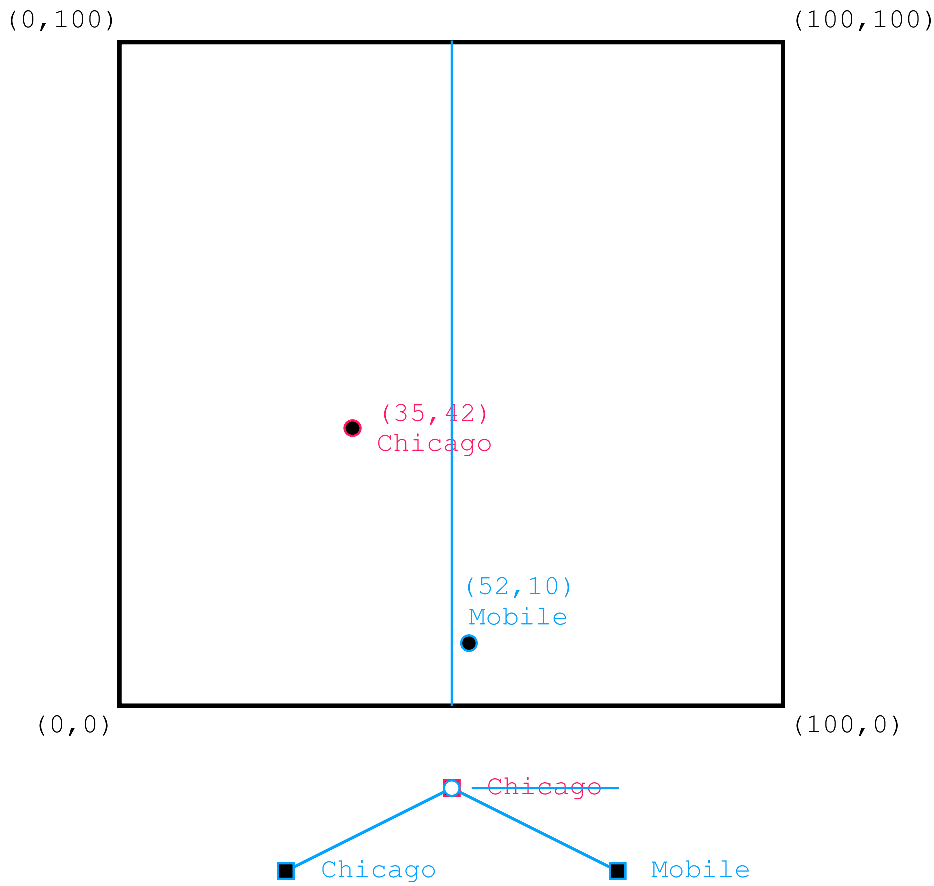
PR K-D TREE (Knowlton)

- A region contains at most one data point
- Analogous to EXCELL with bucket size of 1



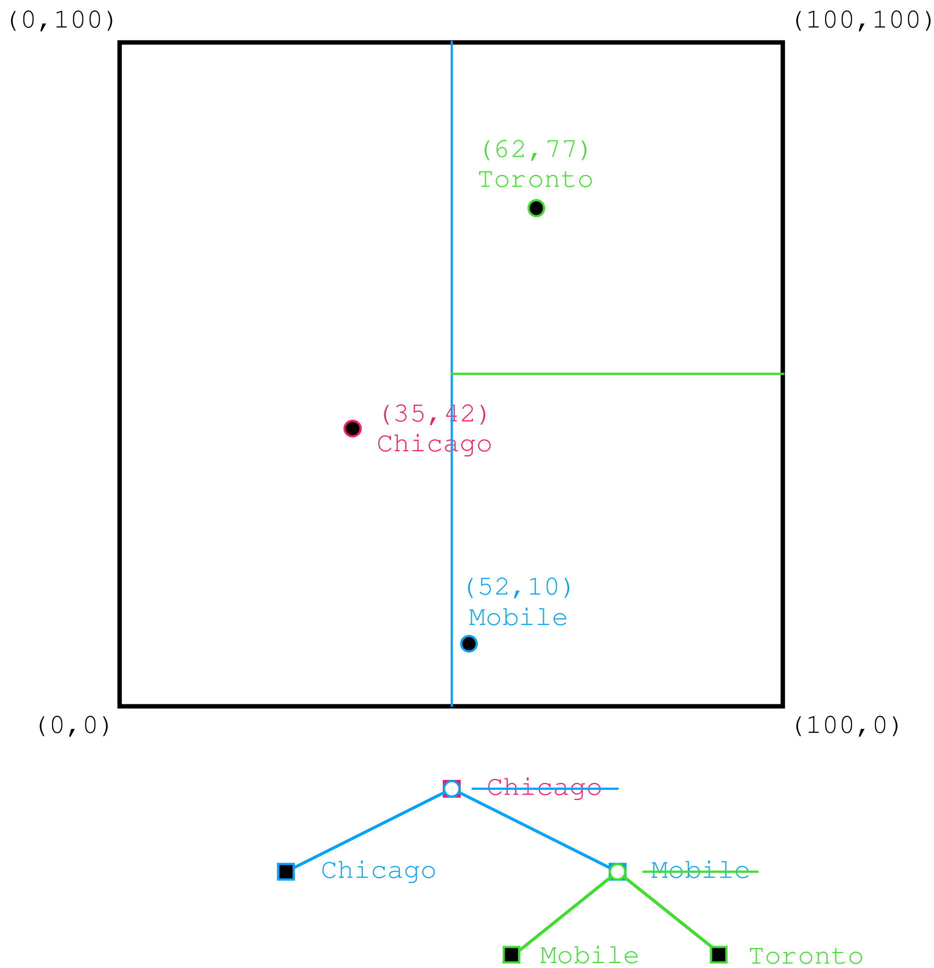
PR K-D TREE (Knowlton)

- A region contains at most one data point
- Analogous to EXCELL with bucket size of 1



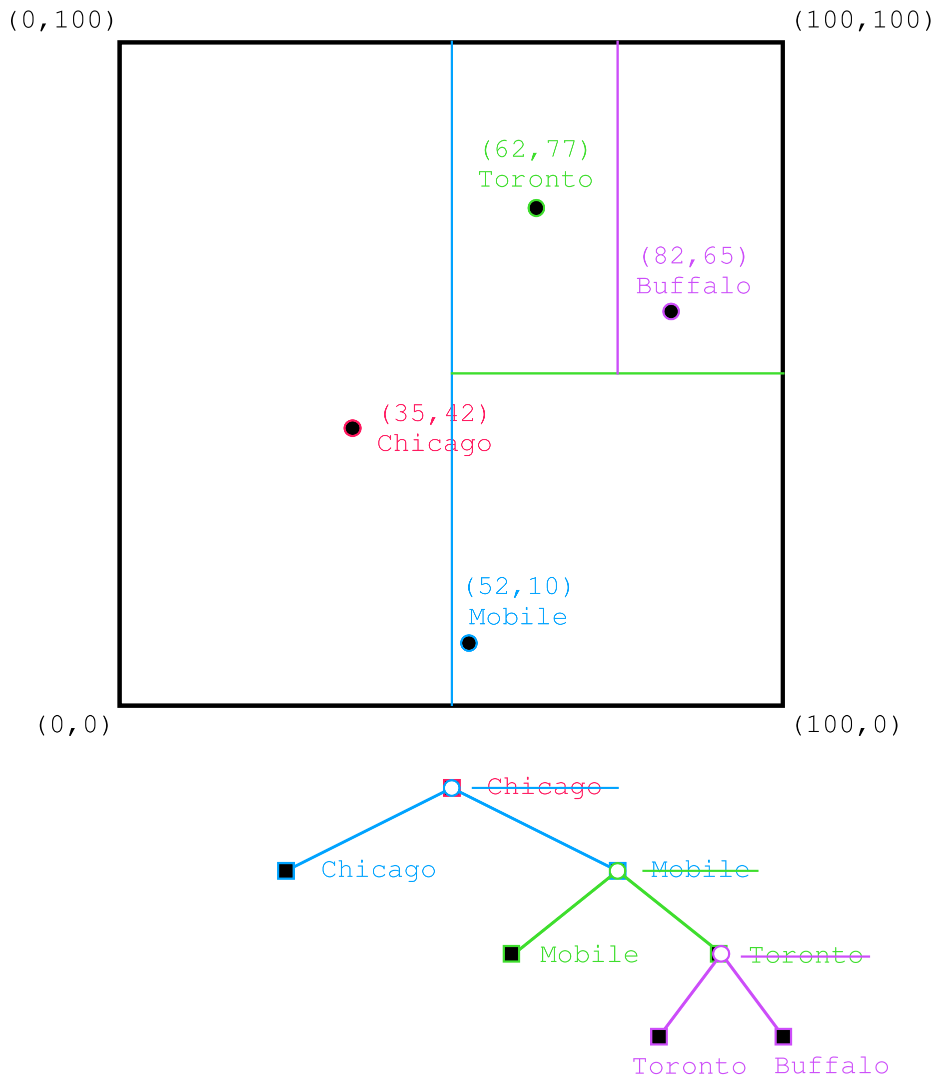
PR K-D TREE (Knowlton)

- A region contains at most one data point
- Analogous to EXCELL with bucket size of 1



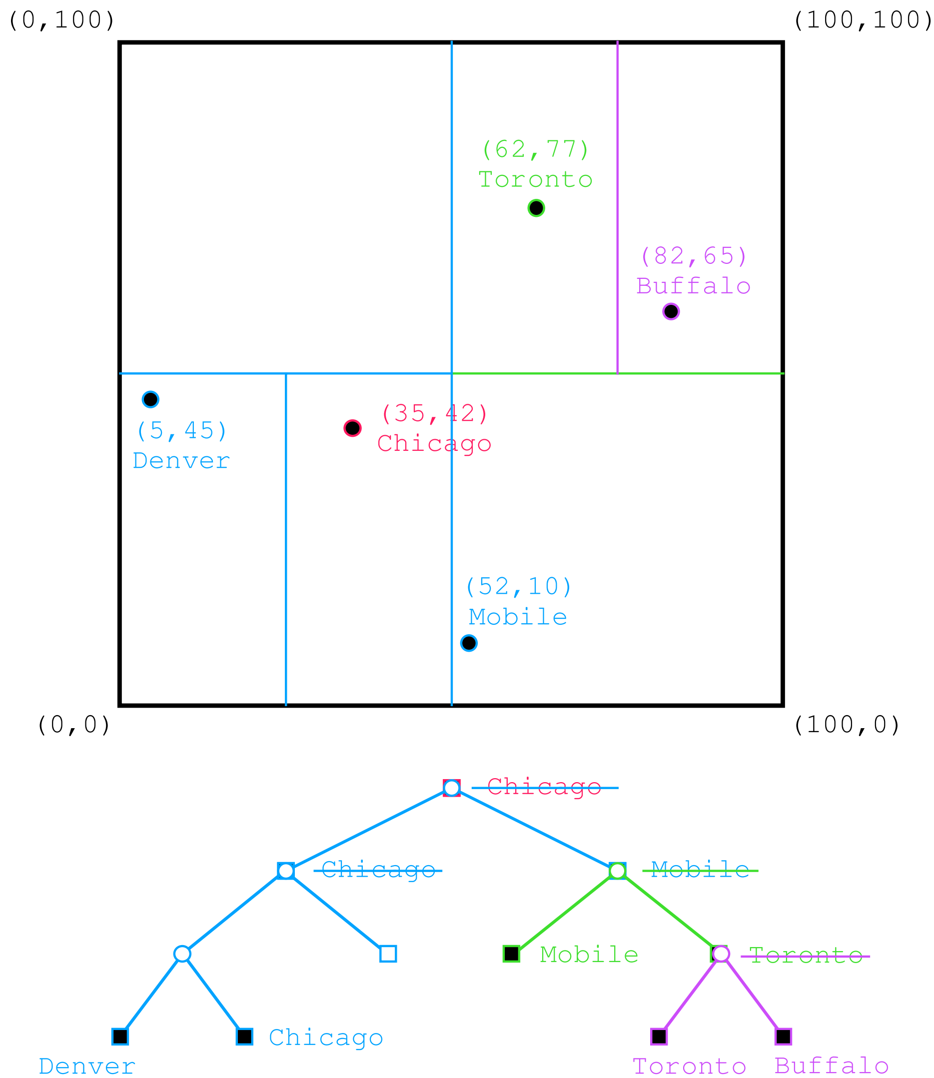
PR K-D TREE (Knowlton)

- A region contains at most one data point
- Analogous to EXCELL with bucket size of 1



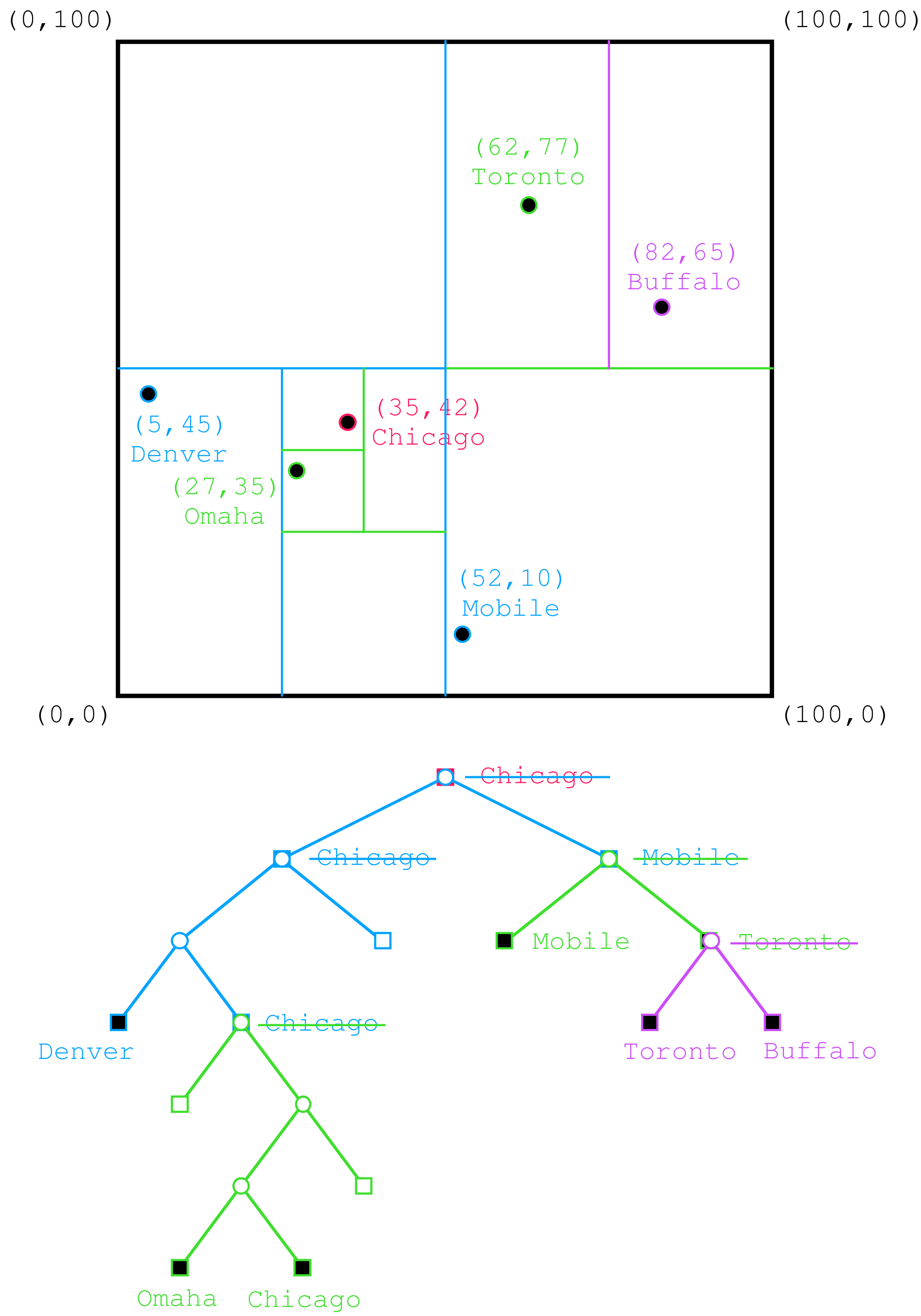
PR K-D TREE (Knowlton)

- A region contains at most one data point
- Analogous to EXCELL with bucket size of 1



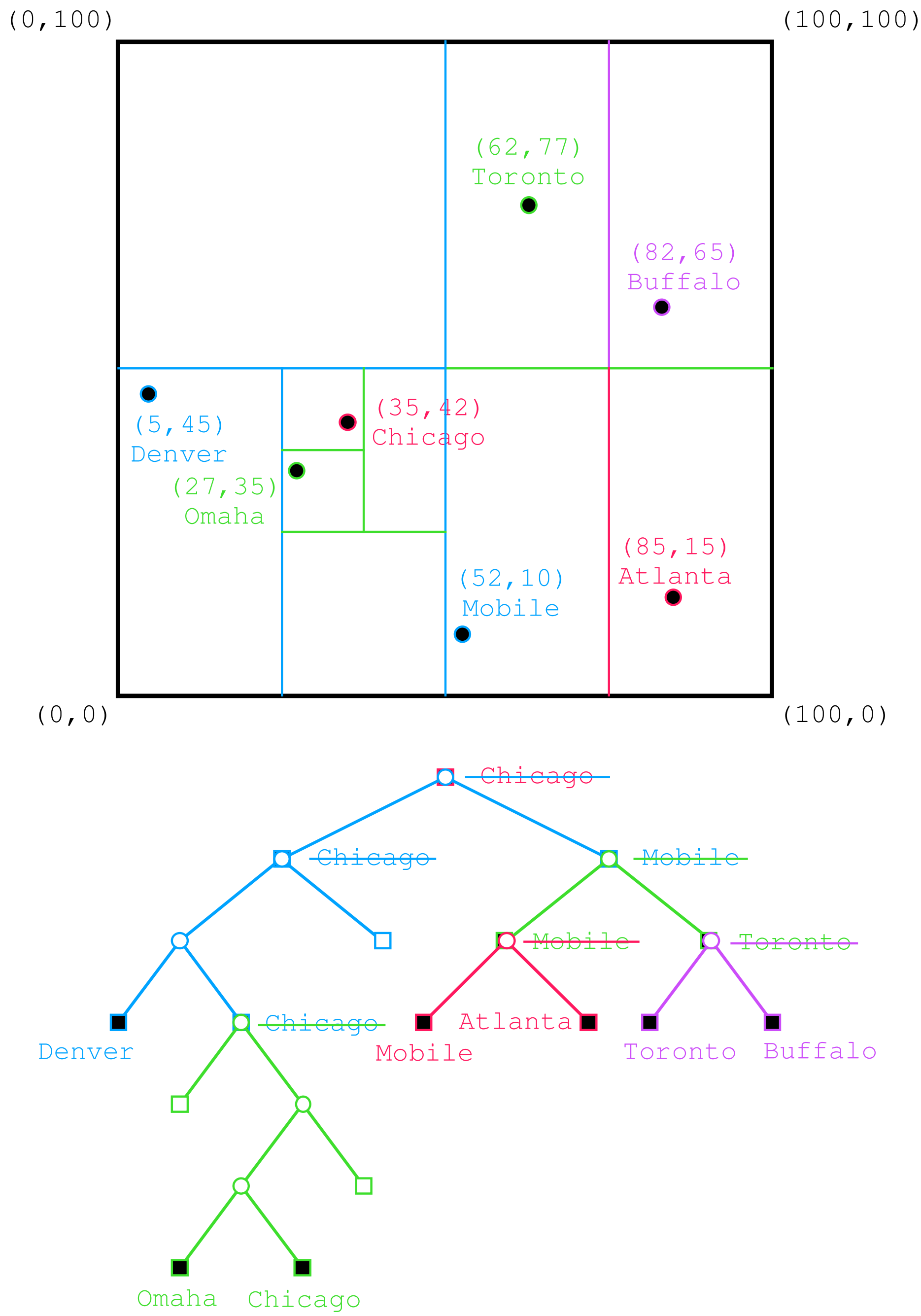
PR K-D TREE (Knowlton)

- A region contains at most one data point
- Analogous to EXCELL with bucket size of 1



PR K-D TREE (Knowlton)

- A region contains at most one data point
- Analogous to EXCELL with bucket size of 1



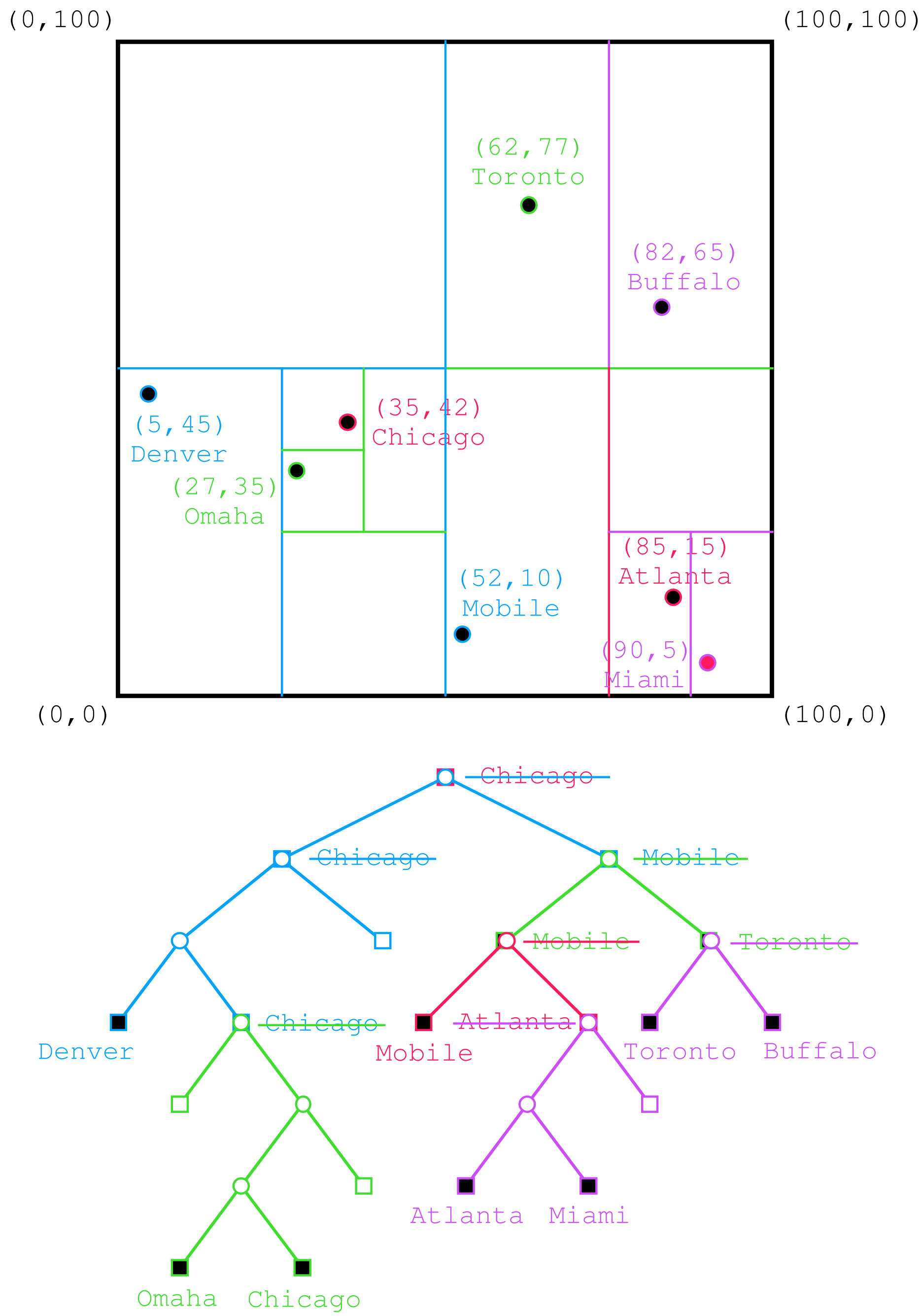
071-072--PR k-d tree

9 8 7 6 5 4 3 2 1
v r g z v g z r b

hp19

PR K-D TREE (Knowlton)

- A region contains at most one data point
- Analogous to EXCELL with bucket size of 1

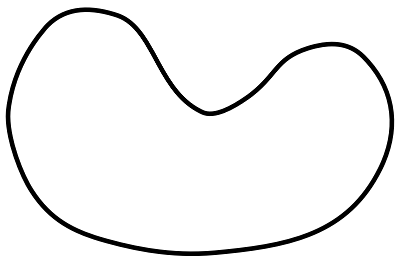


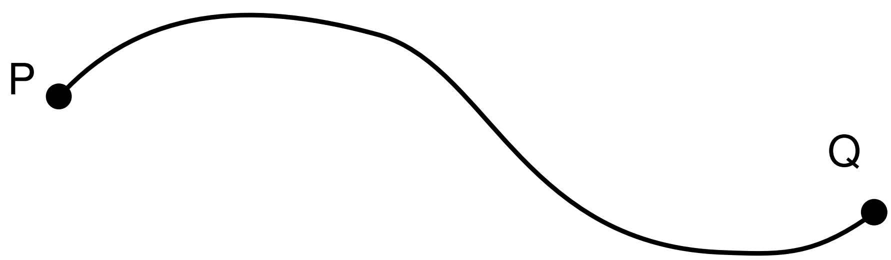
Outline

1. Introduction
2. Points
3. Lines
4. Regions, Volumes, and Surfaces
5. Bounding Box Hierarchies
6. Rectangles
7. Surfaces and Volumes
8. Metric Data
9. Operations
10. Indexing Spatiotextual Data
11. Example system

STRIP TREE (Ballard, Peucker)

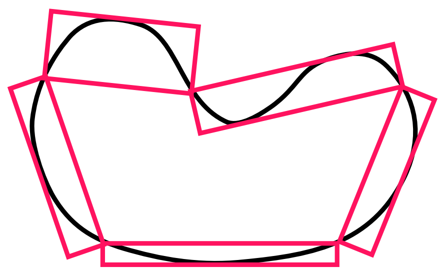
- Top-down hierarchical curve approximation
- Rectangle strips of arbitrary orientation
- Assume curve is continuous
- Ex:

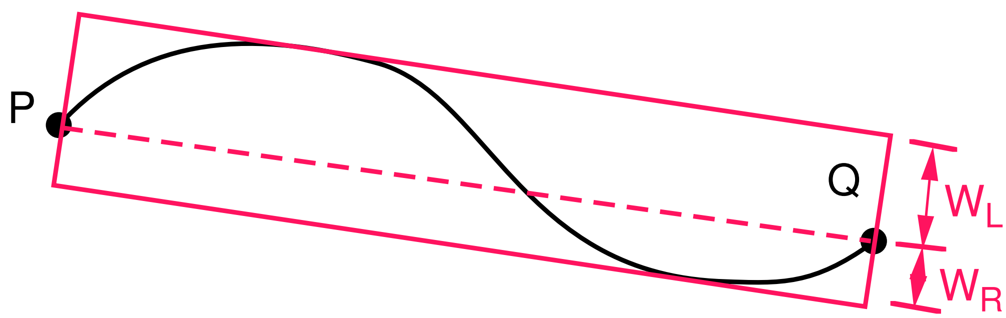




STRIP TREE (Ballard, Peucker)

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



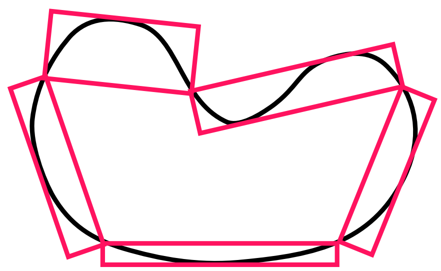


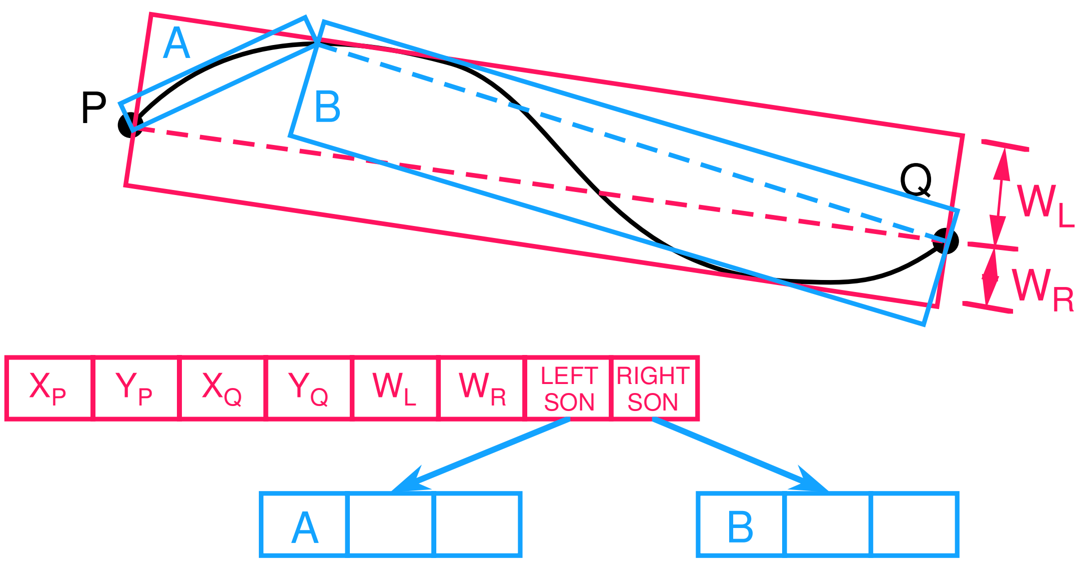
X_P	Y_P	X_Q	Y_Q	W_L	W_R	LEFT SON	RIGHT SON

- *Contact points* = where the curve touches the box
  1. not tangent points
  2. curve need not be differentiable - just continuous

STRIP TREE (Ballard, Peucker)

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

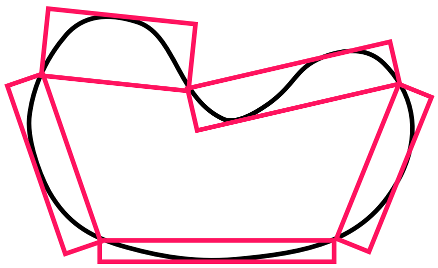


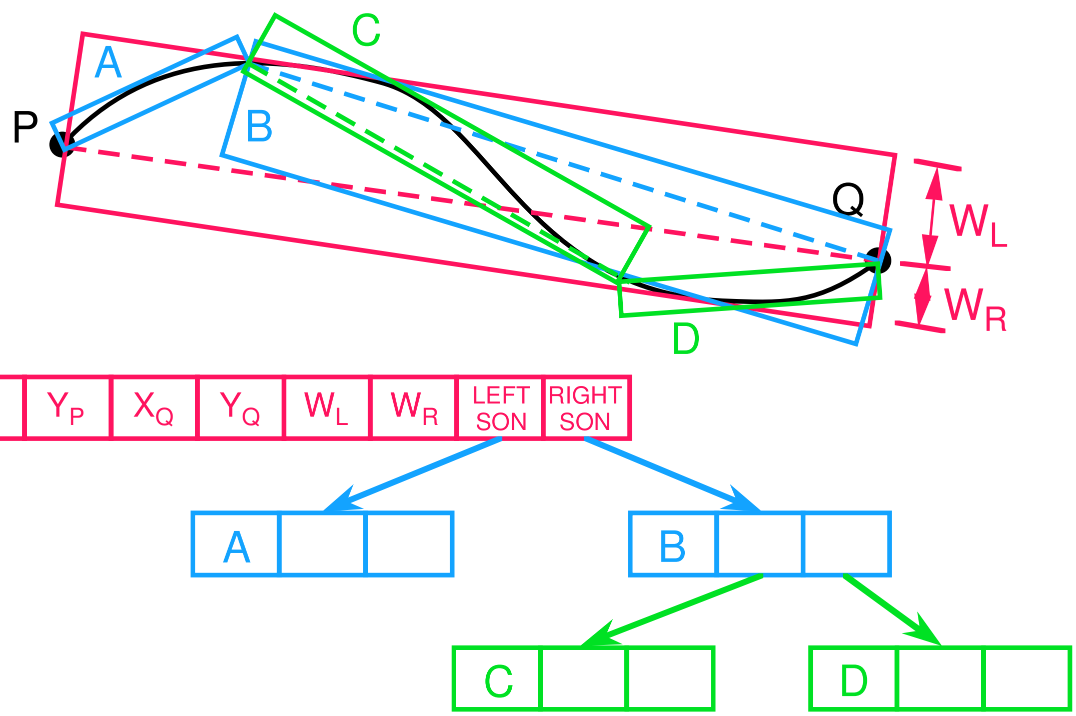


- *Contact points* = where the curve touches the box
  1. not tangent points
  2. curve need not be differentiable - just continuous

STRIP TREE (Ballard, Peucker)

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

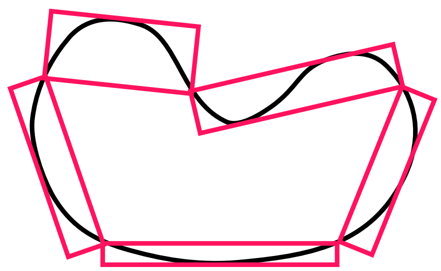


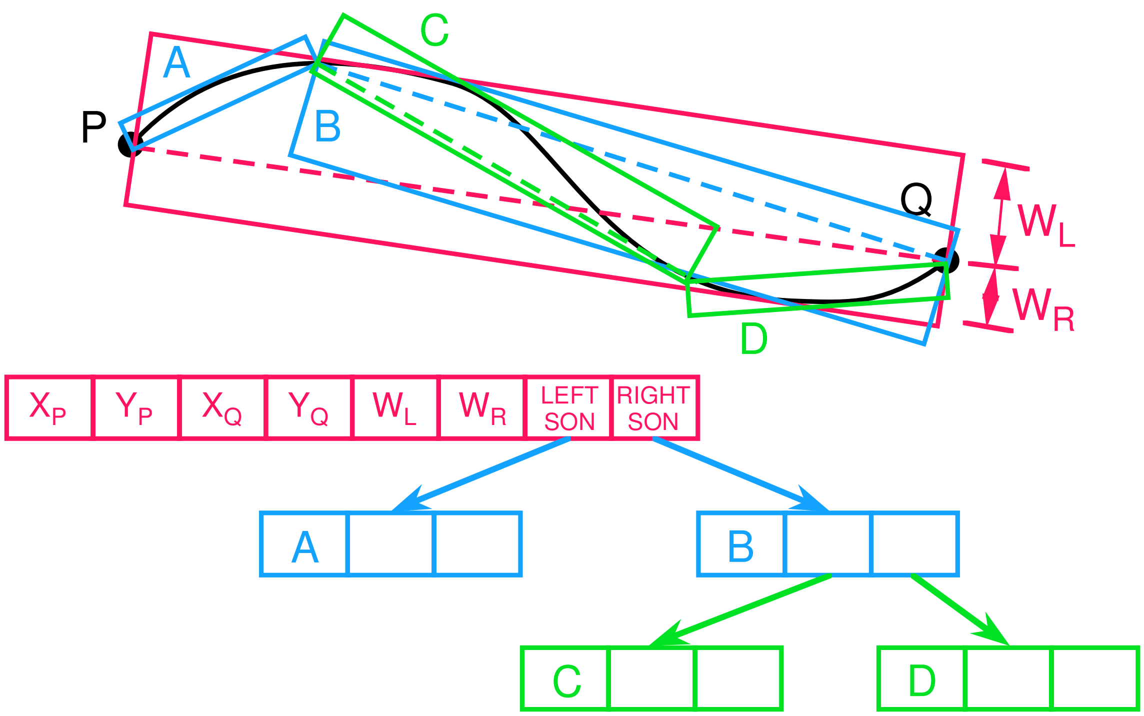


- *Contact points* = where the curve touches the box
  1. not tangent points
  2. curve need not be differentiable - just continuous

STRIP TREE (Ballard, Peucker)

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

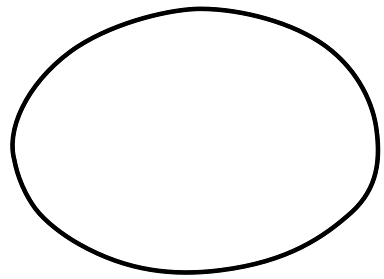




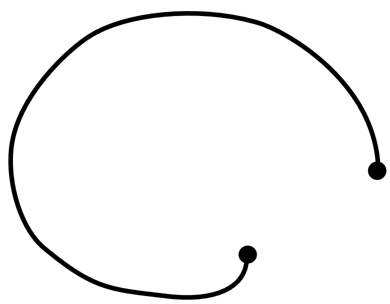
- *Contact points* = where the curve touches the box
  1. not tangent points
  2. curve need not be differentiable - just continuous
- Terminate when all rectangles are of width $\leqslant$ W

SPECIAL CASES

1. Closed curve

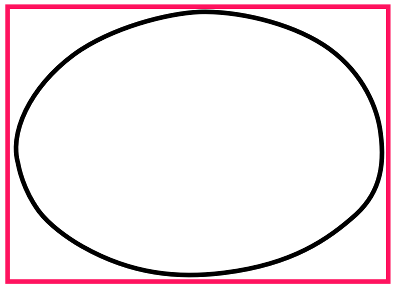


2. Curve extends beyond its endpoints

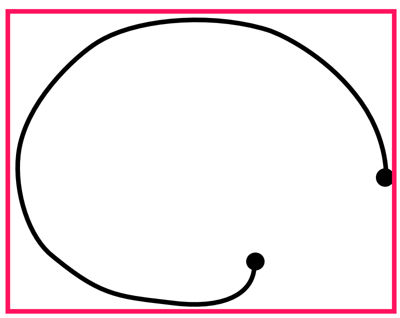


SPECIAL CASES

1. Closed curve



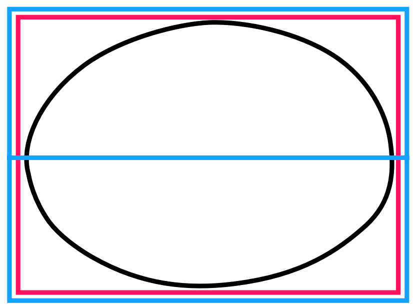
2. Curve extends beyond its endpoints



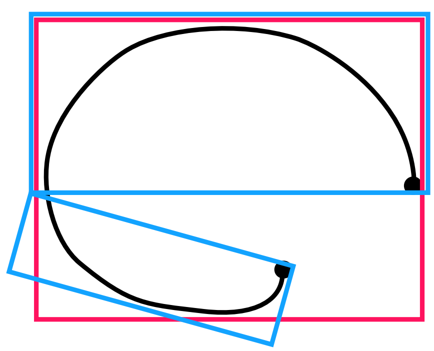
- enclosed by a rectangle

SPECIAL CASES

1. Closed curve



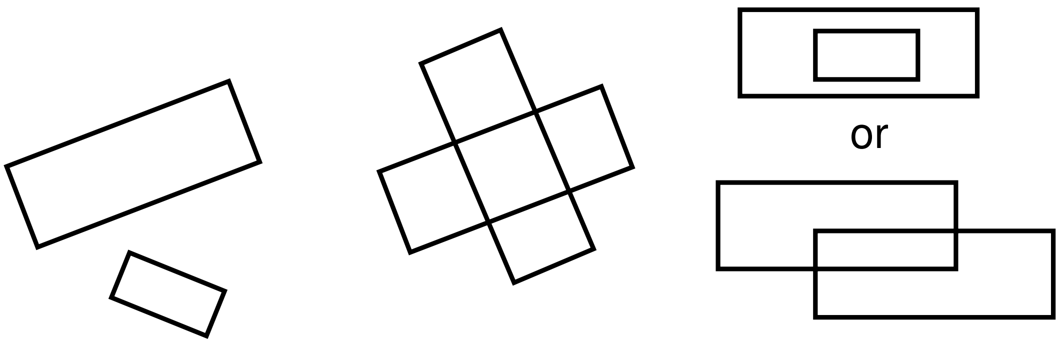
2. Curve extends beyond its endpoints



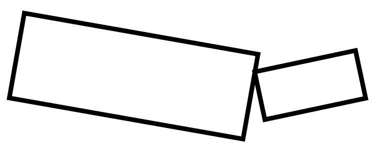
- enclosed by a rectangle
- split into two rectangular strips

APPLICATIONS

1. Curve intersection



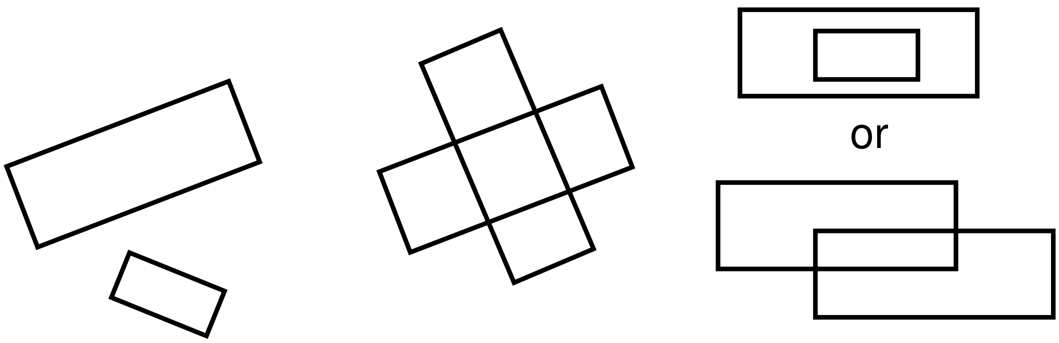
2. Union of two curves



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

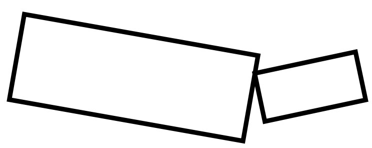
APPLICATIONS

1. Curve intersection



NULL CLEAR POSSIBLE

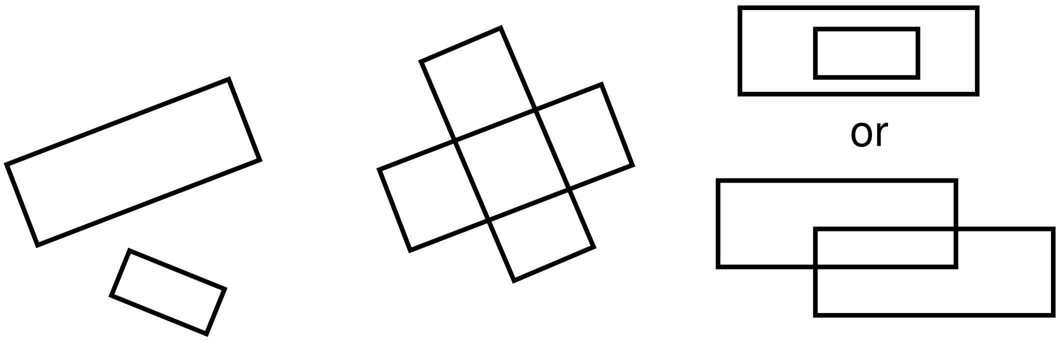
2. Union of two curves



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

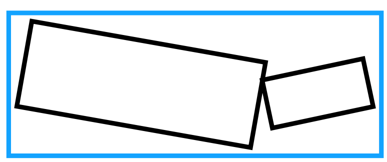
APPLICATIONS

1. Curve intersection



NULL CLEAR POSSIBLE

2. Union of two curves



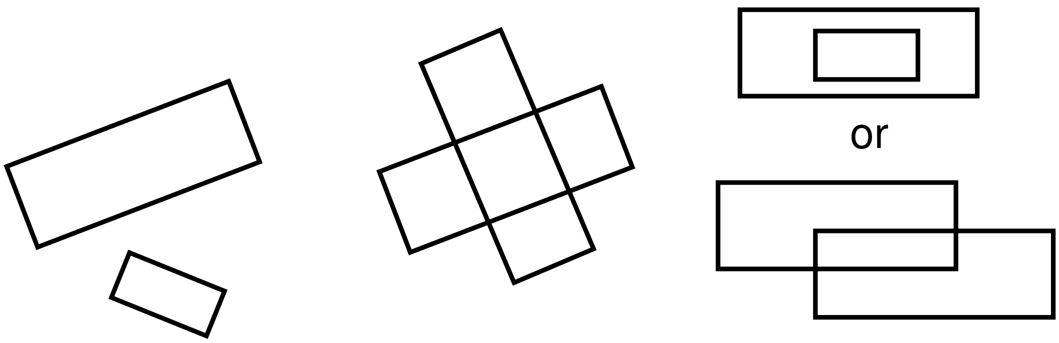
3. Others
   - length
   - area of a closed curve
   - intersection of curves with areas
   - etc.

4 3 2 1
g z r b

cd6

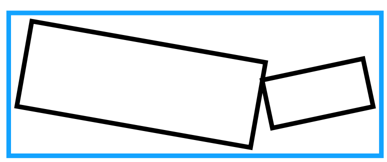
APPLICATIONS

1. Curve intersection



NULL CLEAR POSSIBLE

2. Union of two curves

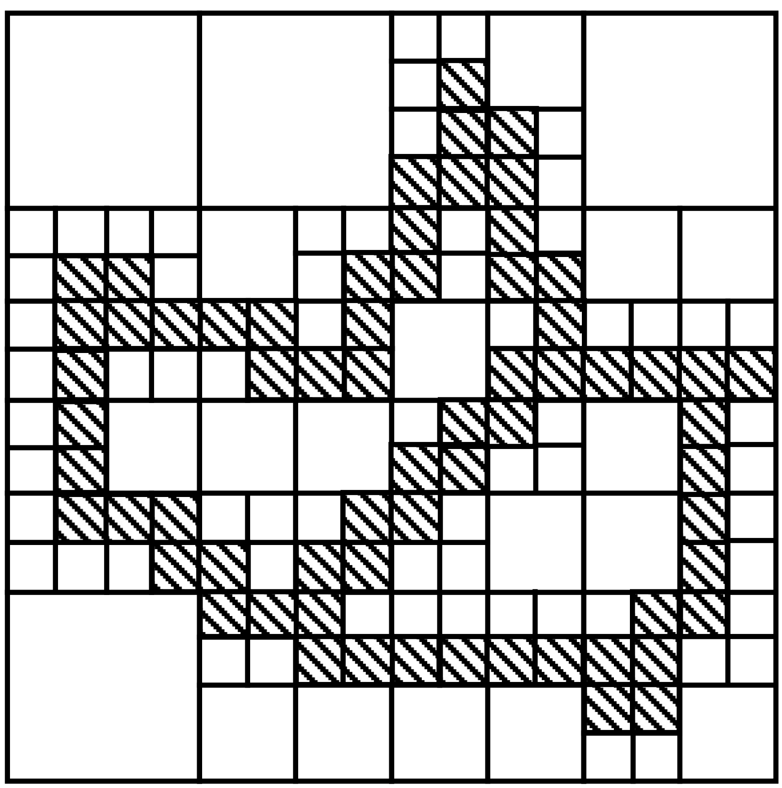


- not possible as the result may fail to be continuous

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

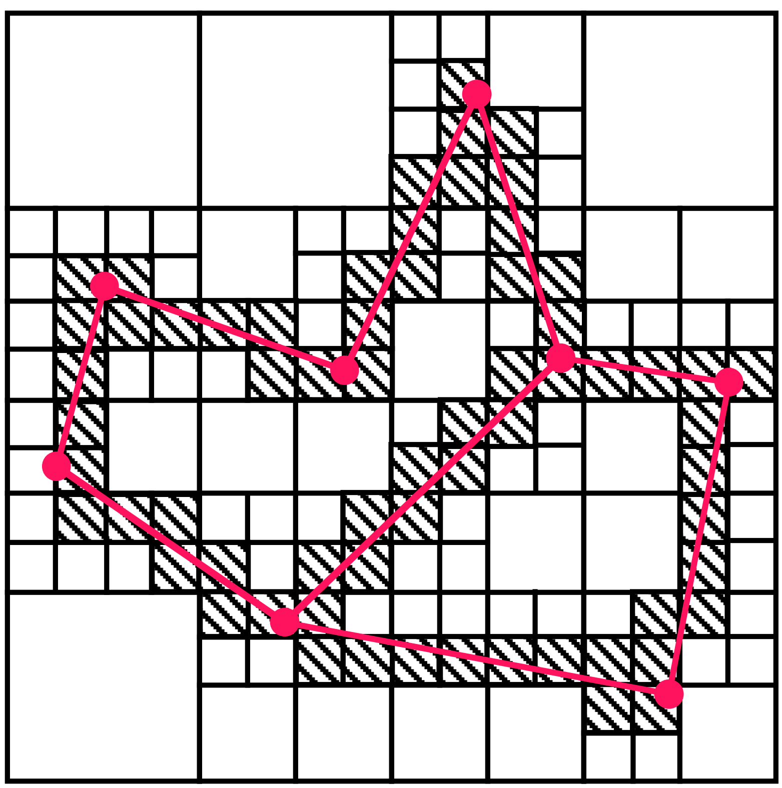
MX QUADTREE FOR REGIONS (Hunter)

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



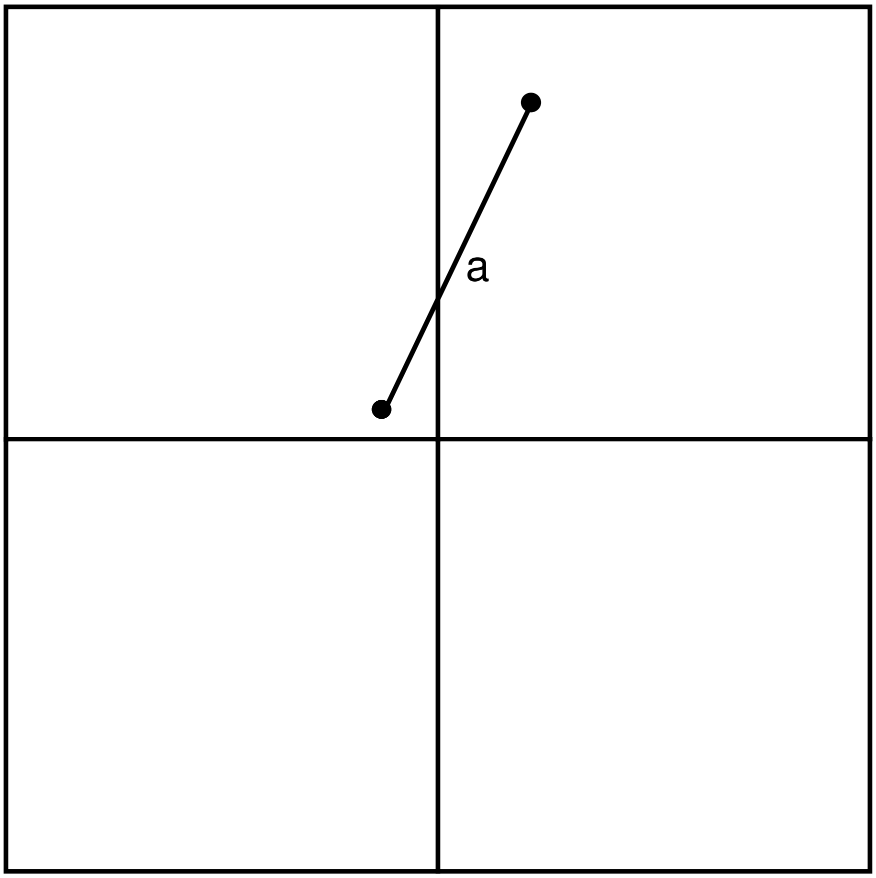
MX QUADTREE FOR REGIONS (Hunter)

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



PM1 QUADTREE

- Vertex-based (one vertex per block)



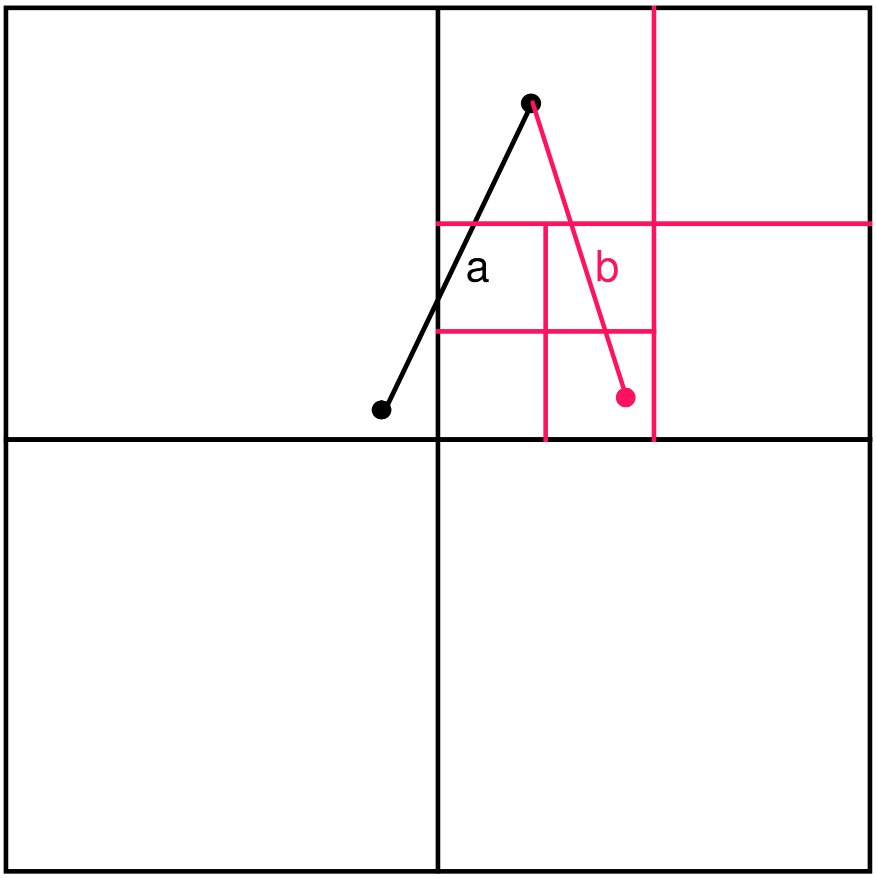
DECOMPOSITION RULE:

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)



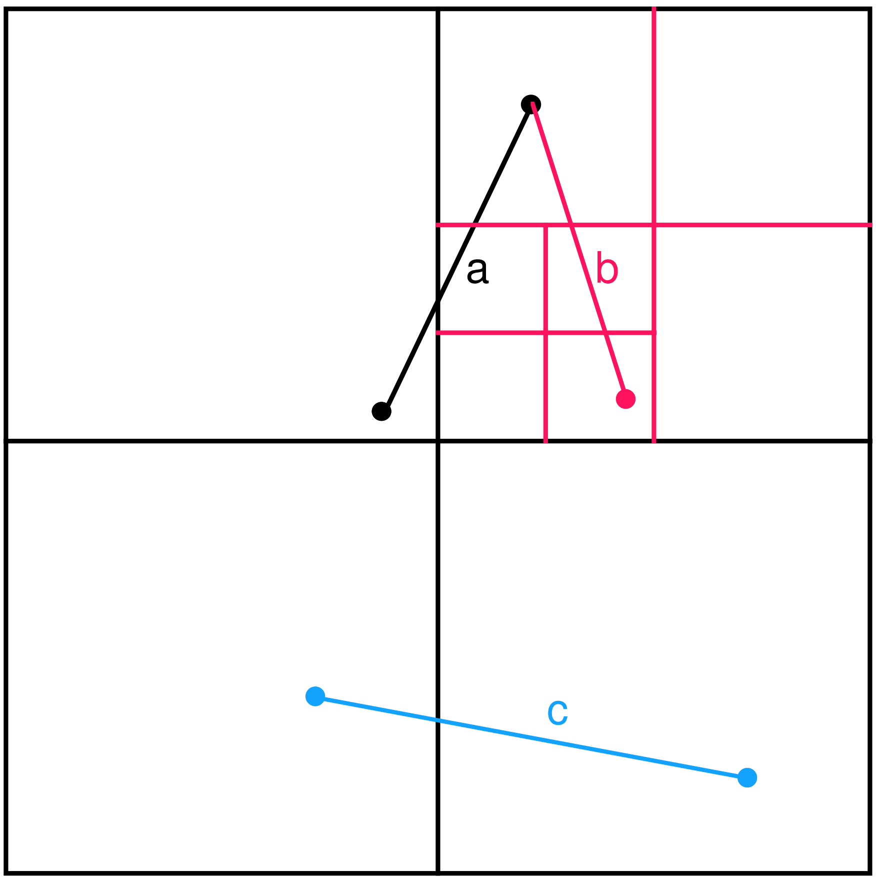
DECOMPOSITION RULE:

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)



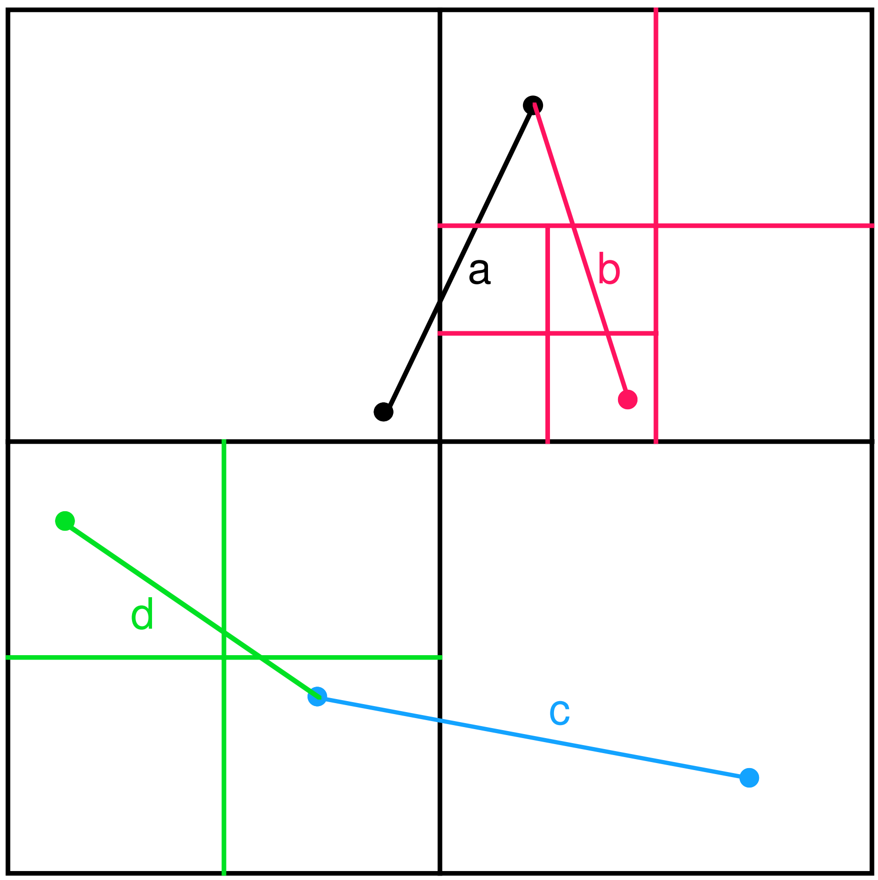
DECOMPOSITION RULE:

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)



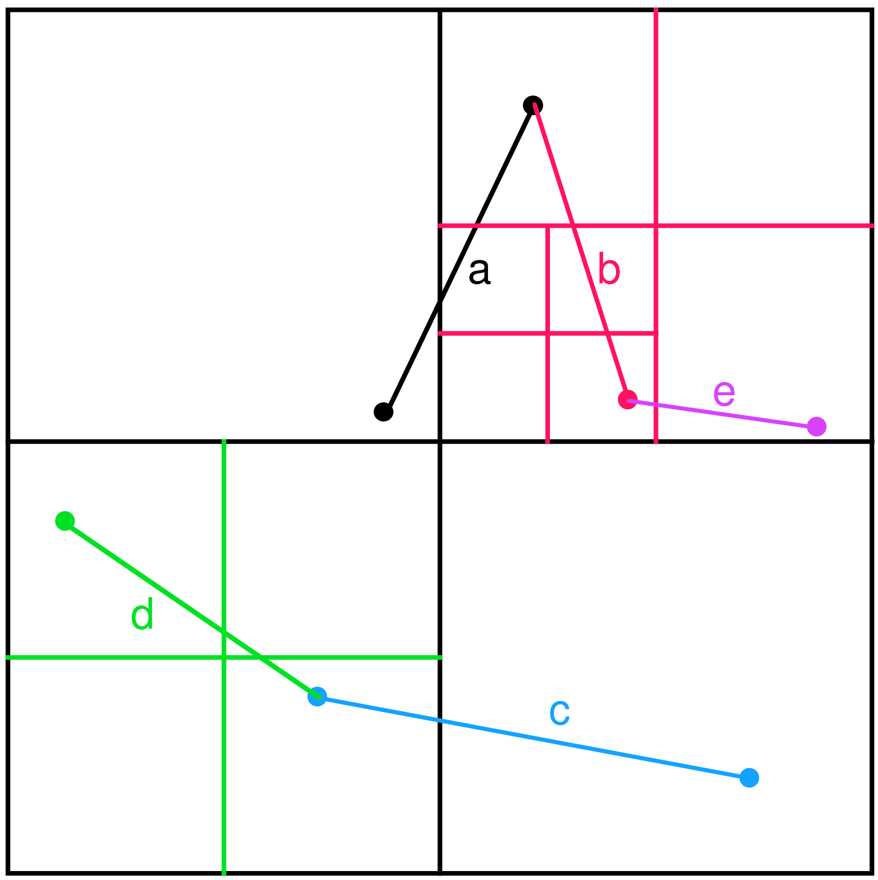
DECOMPOSITION RULE:

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)



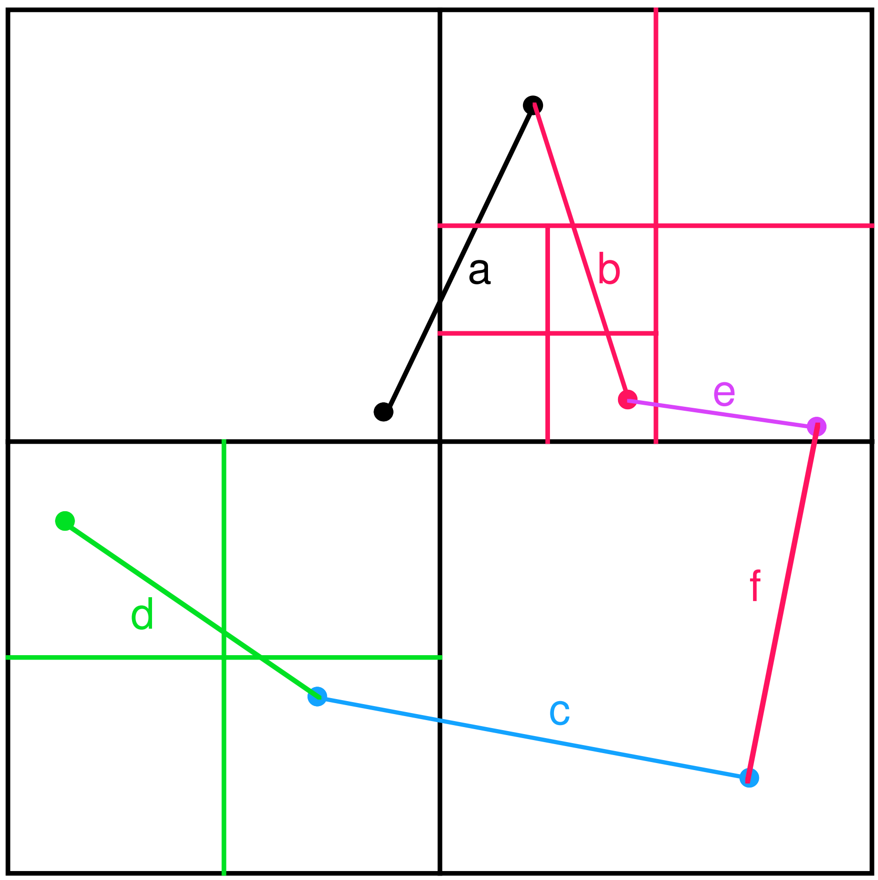
DECOMPOSITION RULE:

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)



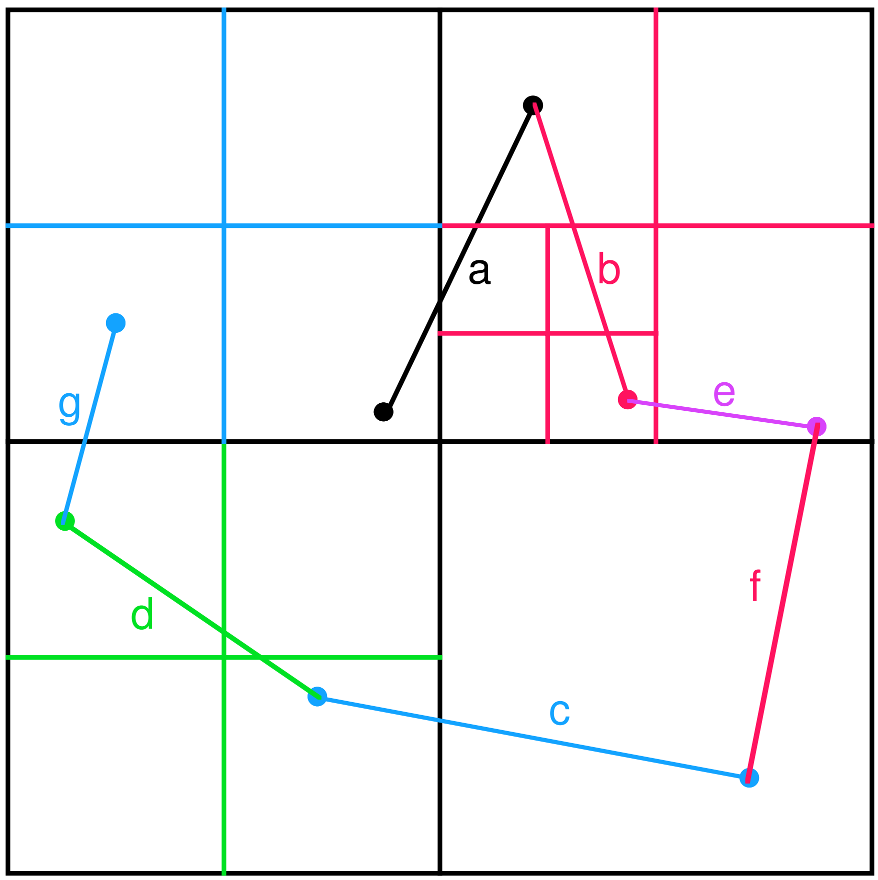
DECOMPOSITION RULE:

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)



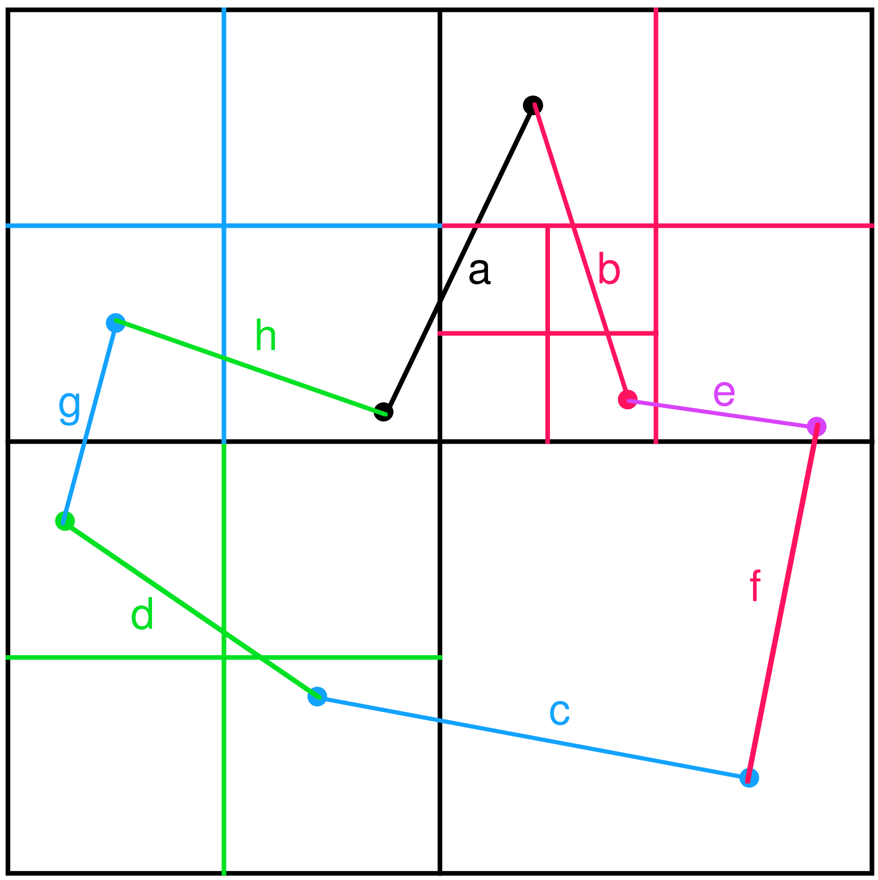
DECOMPOSITION RULE:

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)



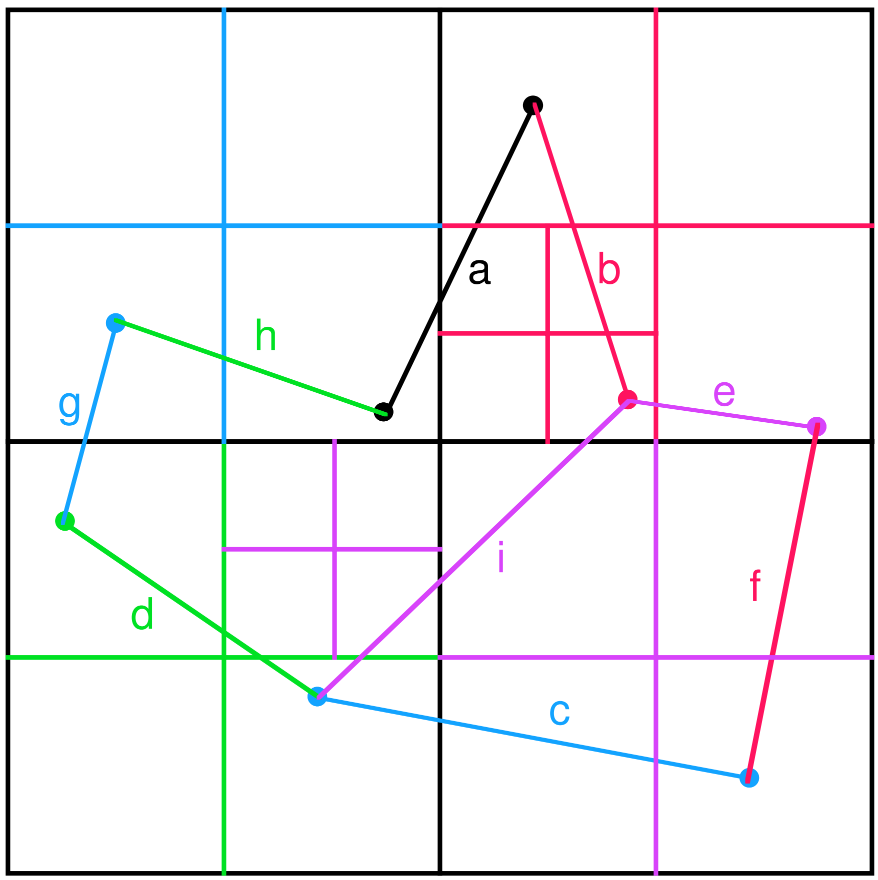
DECOMPOSITION RULE:

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)



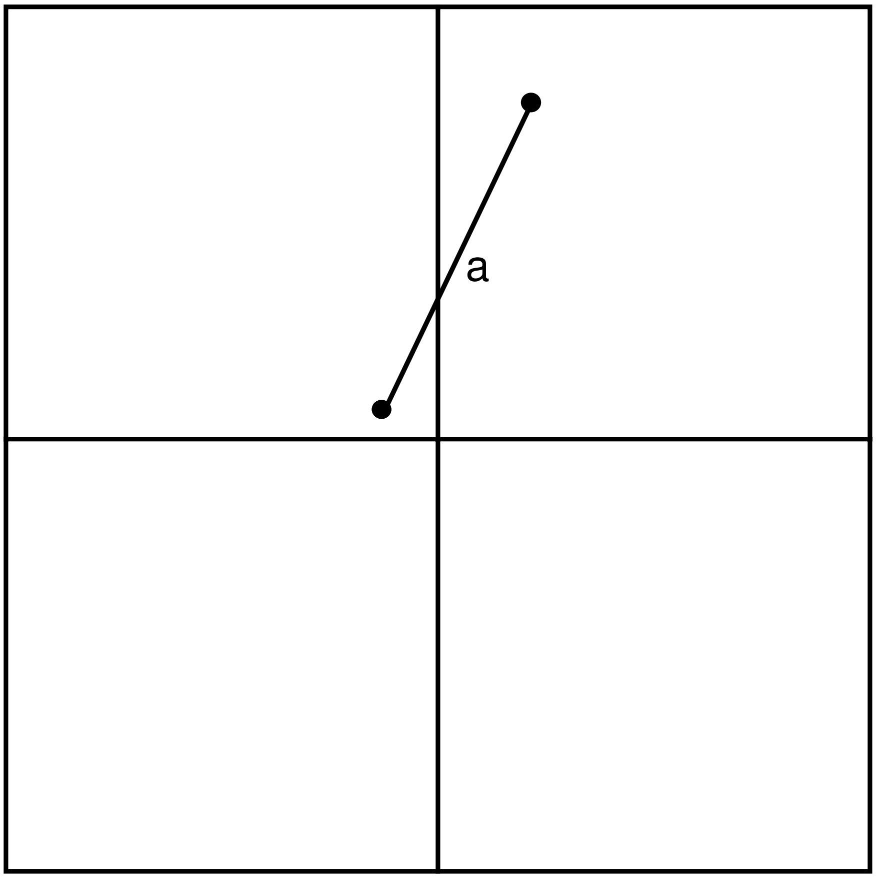
DECOMPOSITION RULE:

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

PM2 QUADTREE

- Vertex-based (one vertex per block)



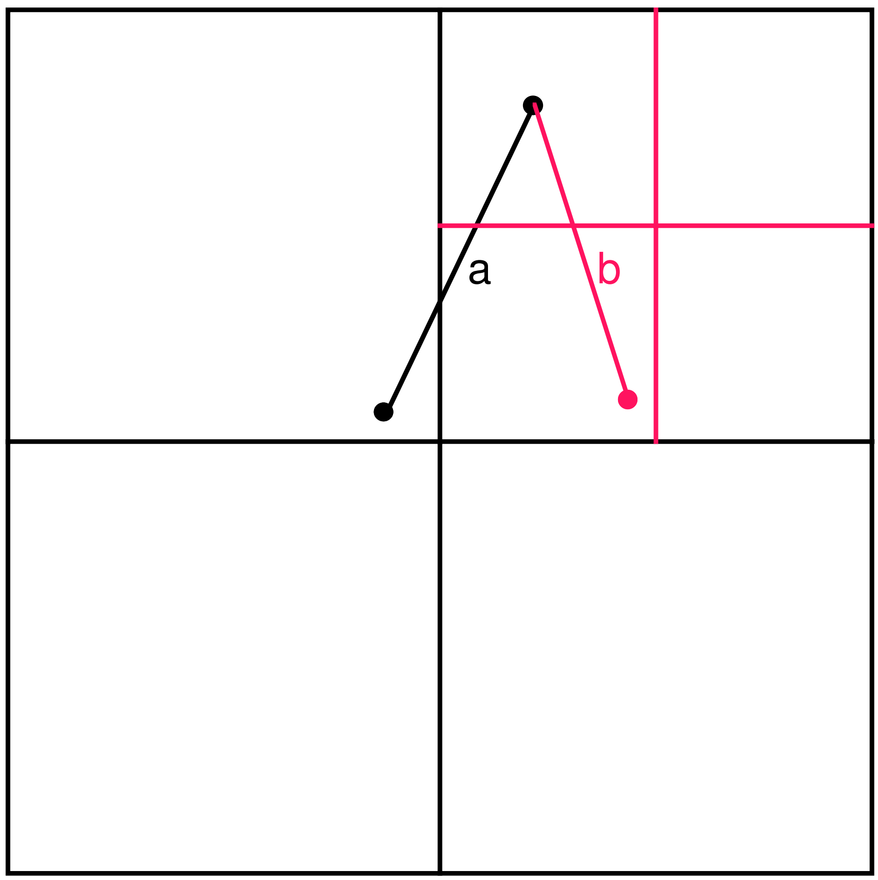
DECOMPOSITION RULE:

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

- Shape independent of order of insertion

PM2 QUADTREE

- Vertex-based (one vertex per block)



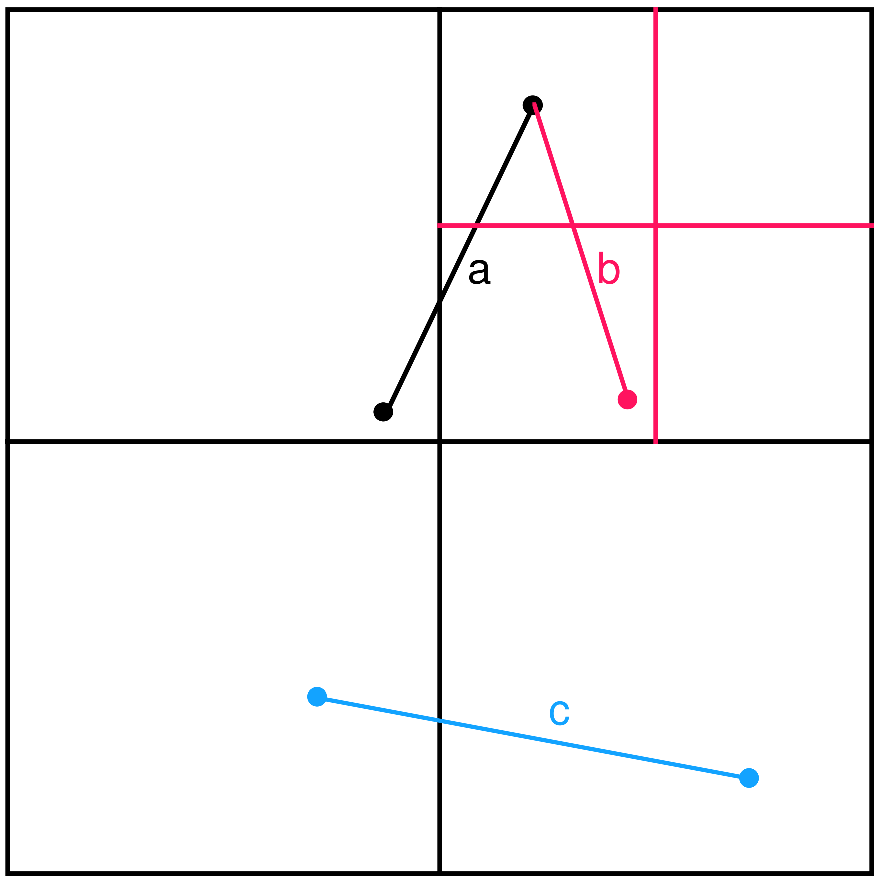
DECOMPOSITION RULE:

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

- Shape independent of order of insertion

PM2 QUADTREE

- Vertex-based (one vertex per block)



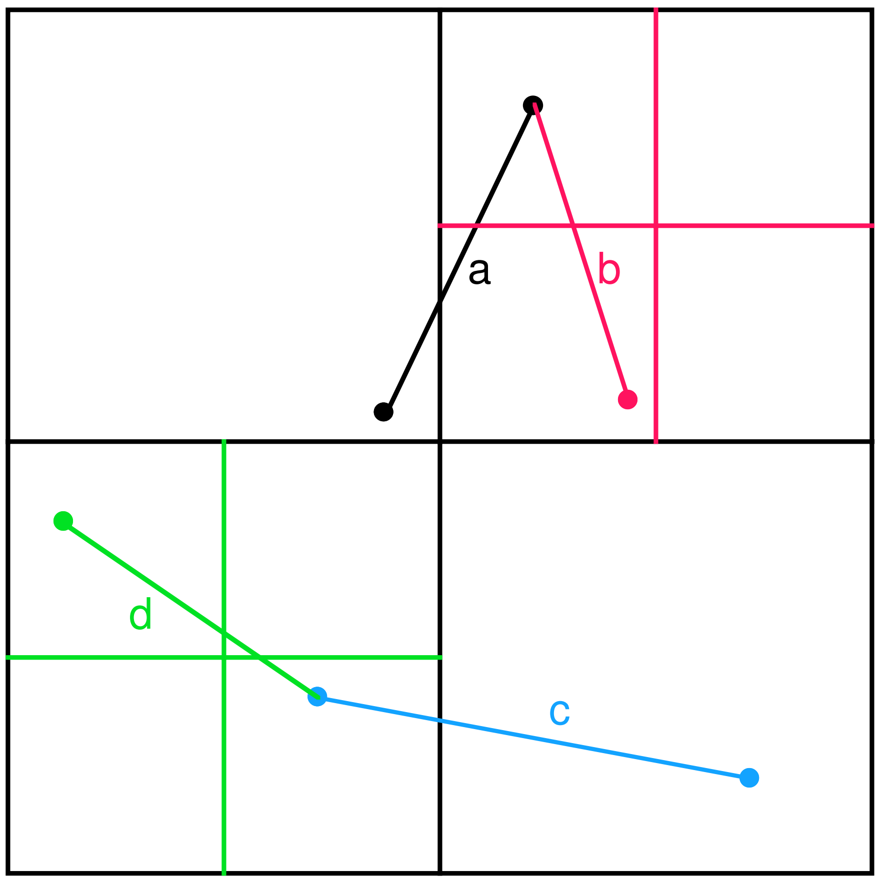
DECOMPOSITION RULE:

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

- Shape independent of order of insertion

PM2 QUADTREE

- Vertex-based (one vertex per block)



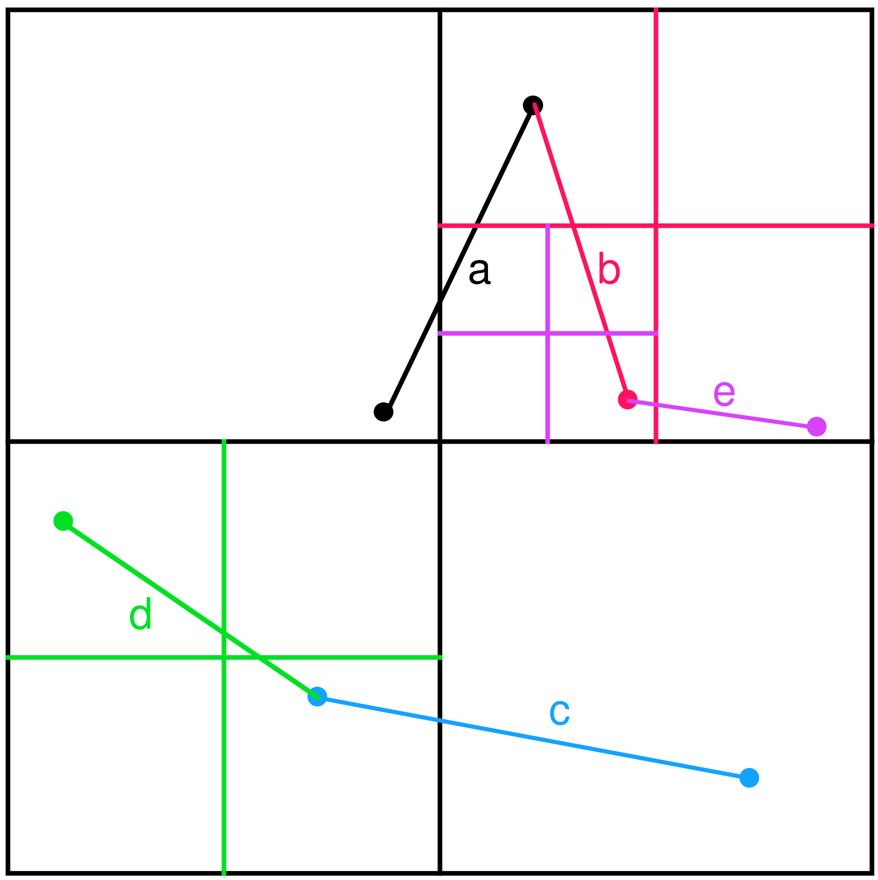
DECOMPOSITION RULE:

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

- Shape independent of order of insertion

PM2 QUADTREE

- Vertex-based (one vertex per block)



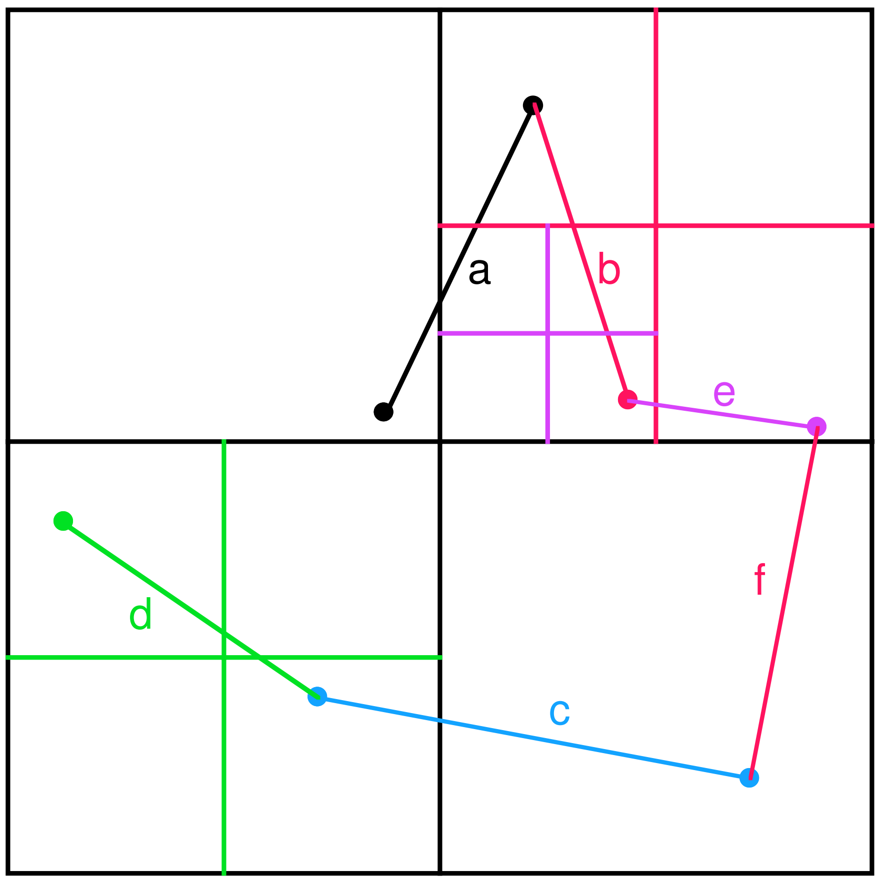
DECOMPOSITION RULE:

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

- Shape independent of order of insertion

PM2 QUADTREE

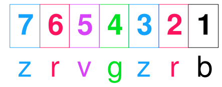
- Vertex-based (one vertex per block)



DECOMPOSITION RULE:

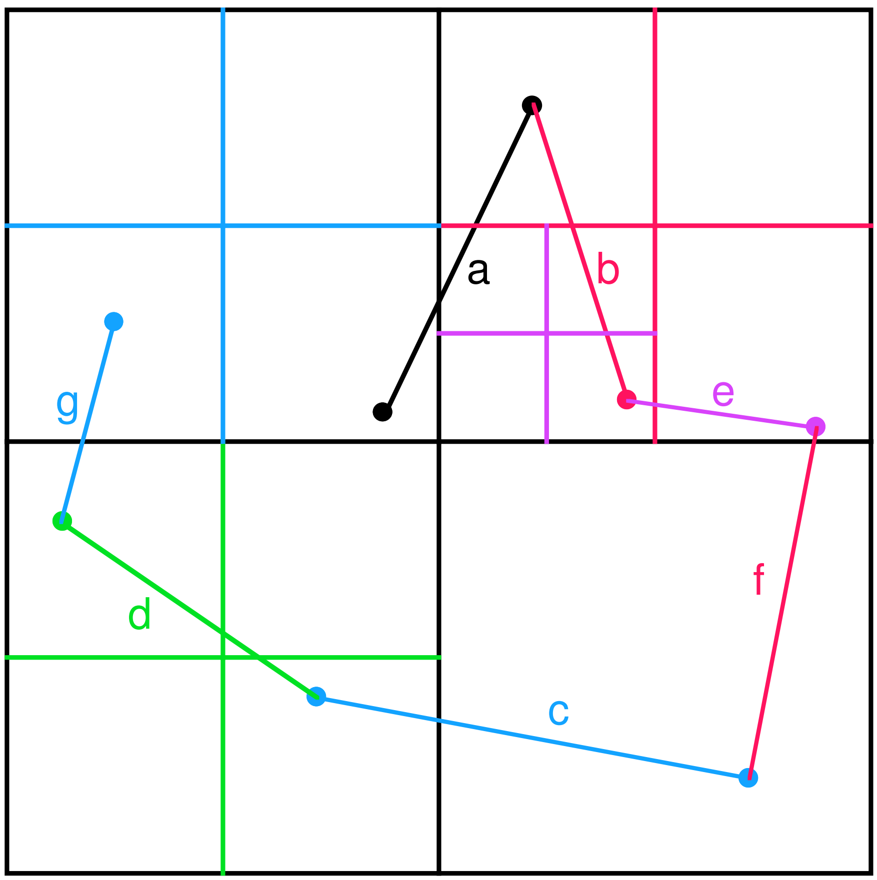
Partitioning occurs when a block contains more than one line segment unless all the segments are incident at the same vertex (the vertex can be in another block!)

- Shape independent of order of insertion



PM2 QUADTREE

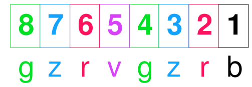
- Vertex-based (one vertex per block)



DECOMPOSITION RULE:

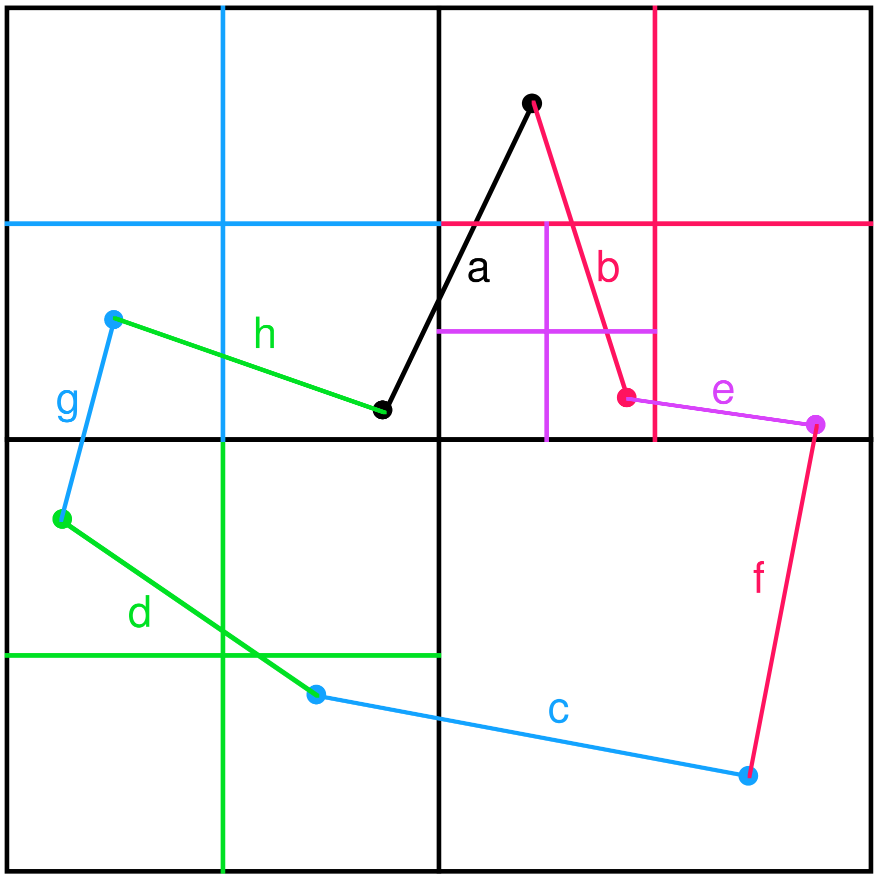
Partitioning occurs when a block contains more than one line segment unless all the segments are incident at the same vertex (the vertex can be in another block!)

- Shape independent of order of insertion



PM2 QUADTREE

- Vertex-based (one vertex per block)



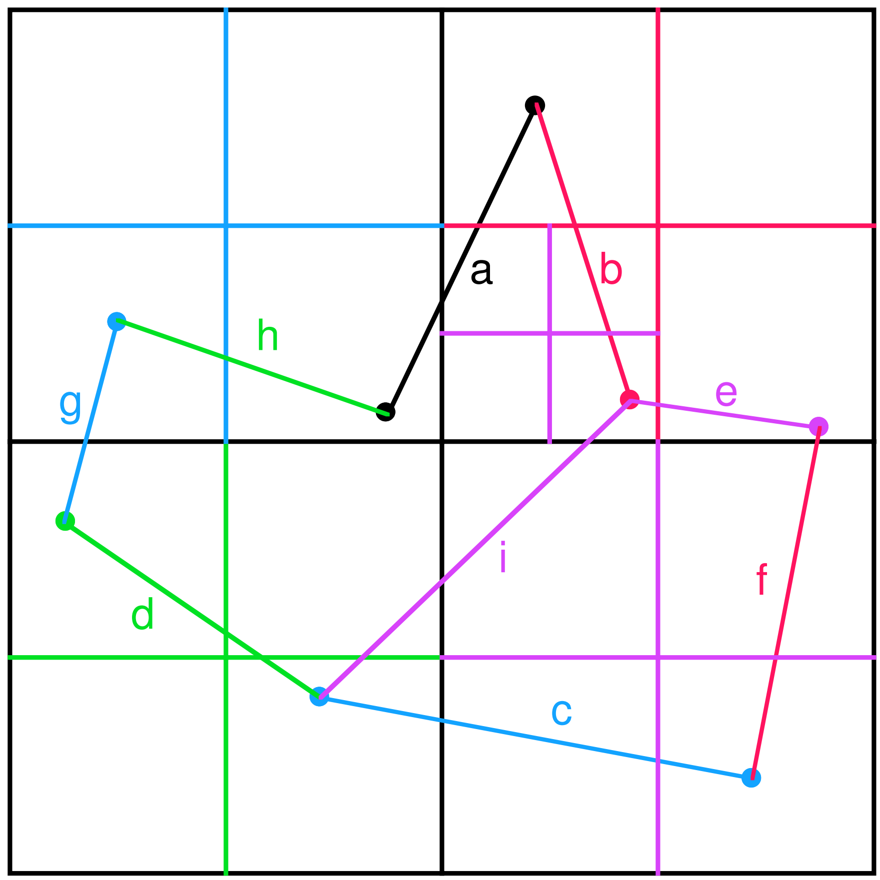
DECOMPOSITION RULE:

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

- Shape independent of order of insertion

PM2 QUADTREE

- Vertex-based (one vertex per block)



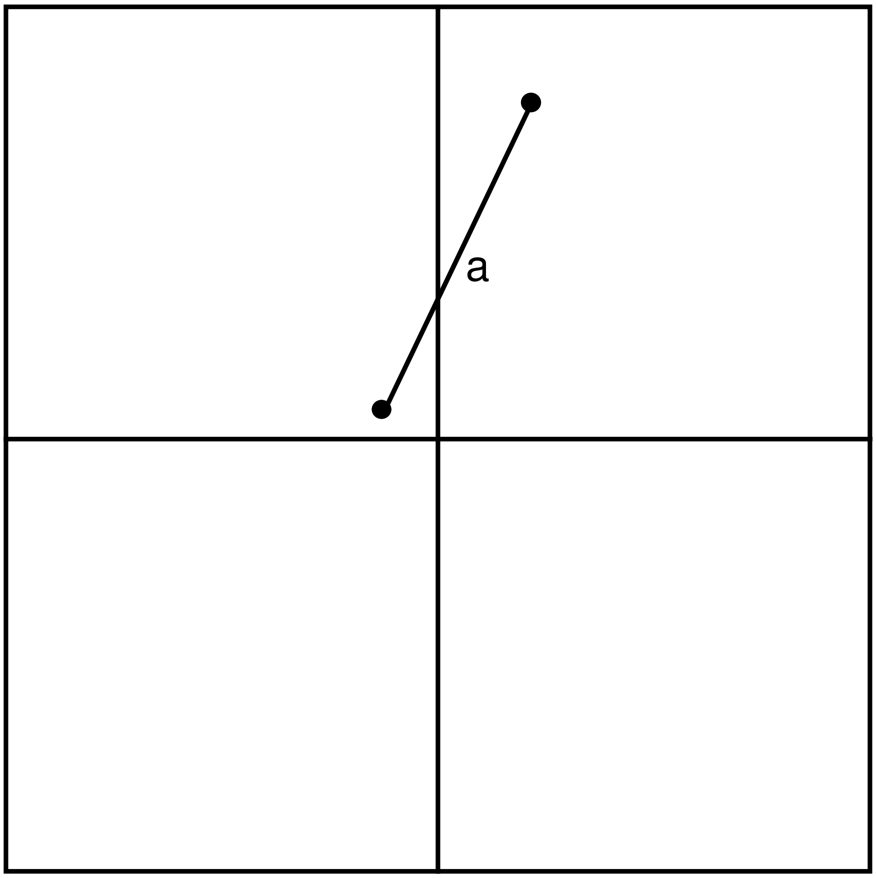
DECOMPOSITION RULE:

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

- Shape independent of order of insertion

PM3 QUADTREE

- Vertex-based (one vertex per block)



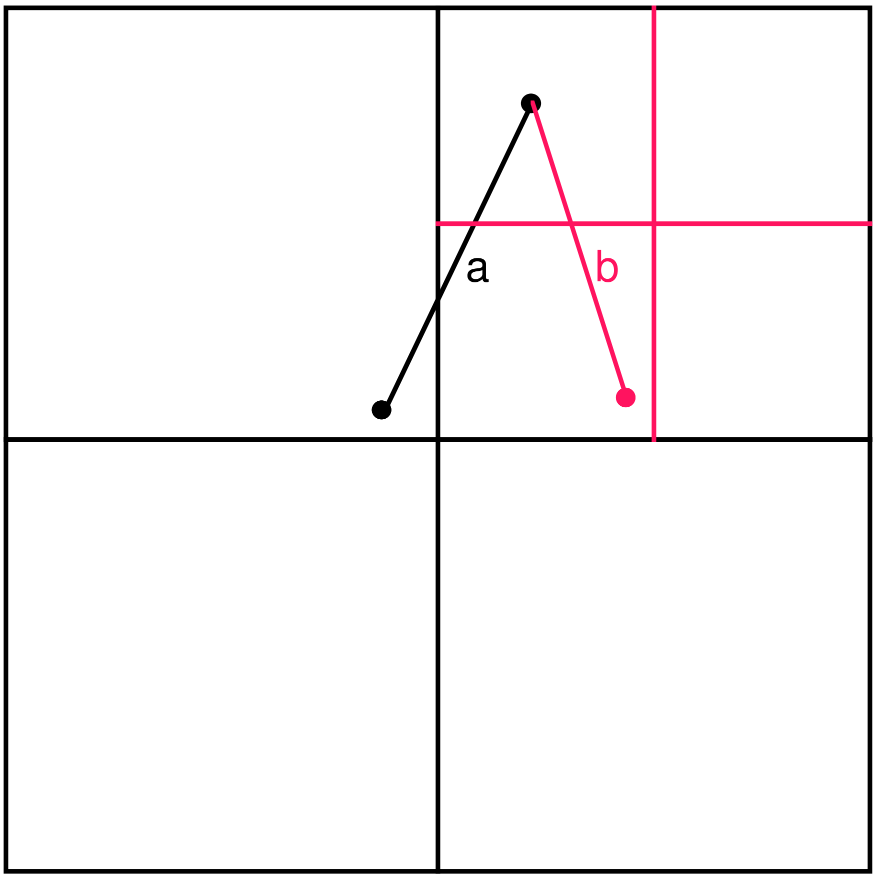
DECOMPOSITION RULE:

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

- Shape independent of order of insertion

PM3 QUADTREE

- Vertex-based (one vertex per block)



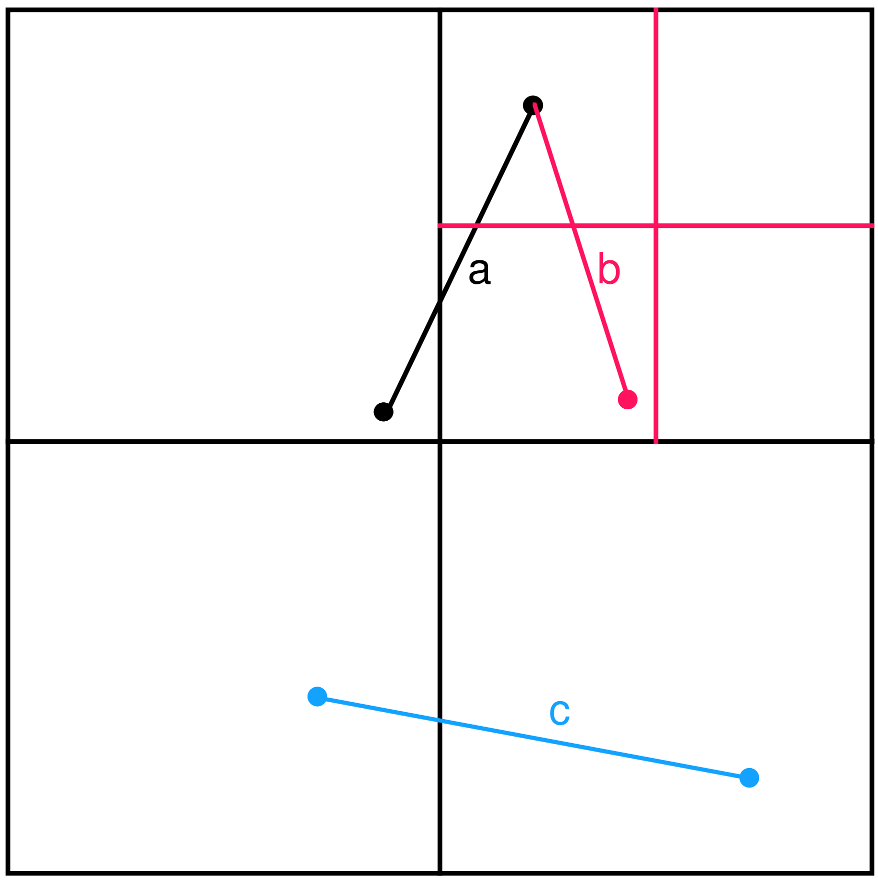
DECOMPOSITION RULE:

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

- Shape independent of order of insertion

PM3 QUADTREE

- Vertex-based (one vertex per block)



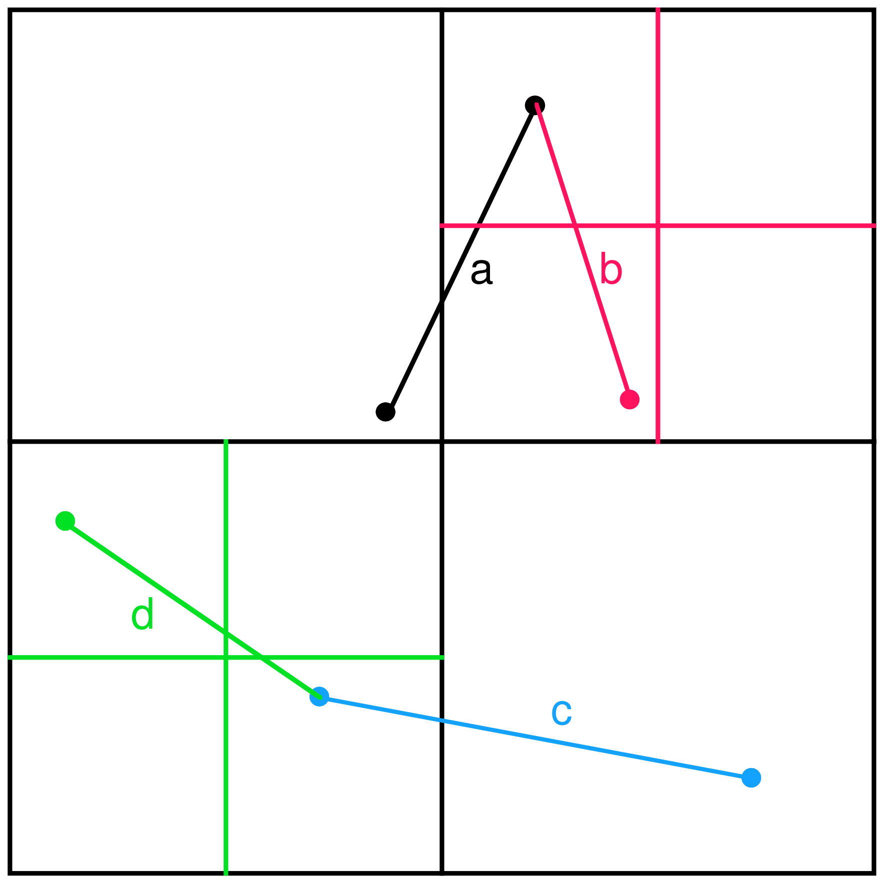
DECOMPOSITION RULE:

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

- Shape independent of order of insertion

PM3 QUADTREE

- Vertex-based (one vertex per block)



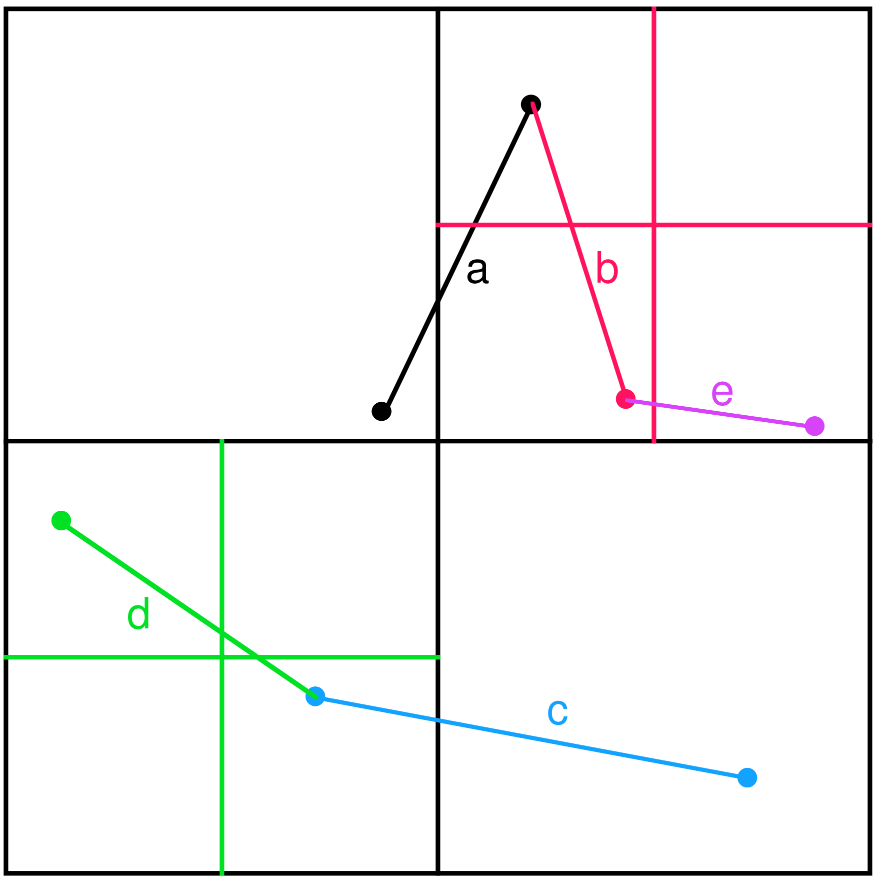
DECOMPOSITION RULE:

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

- Shape independent of order of insertion

PM3 QUADTREE

- Vertex-based (one vertex per block)



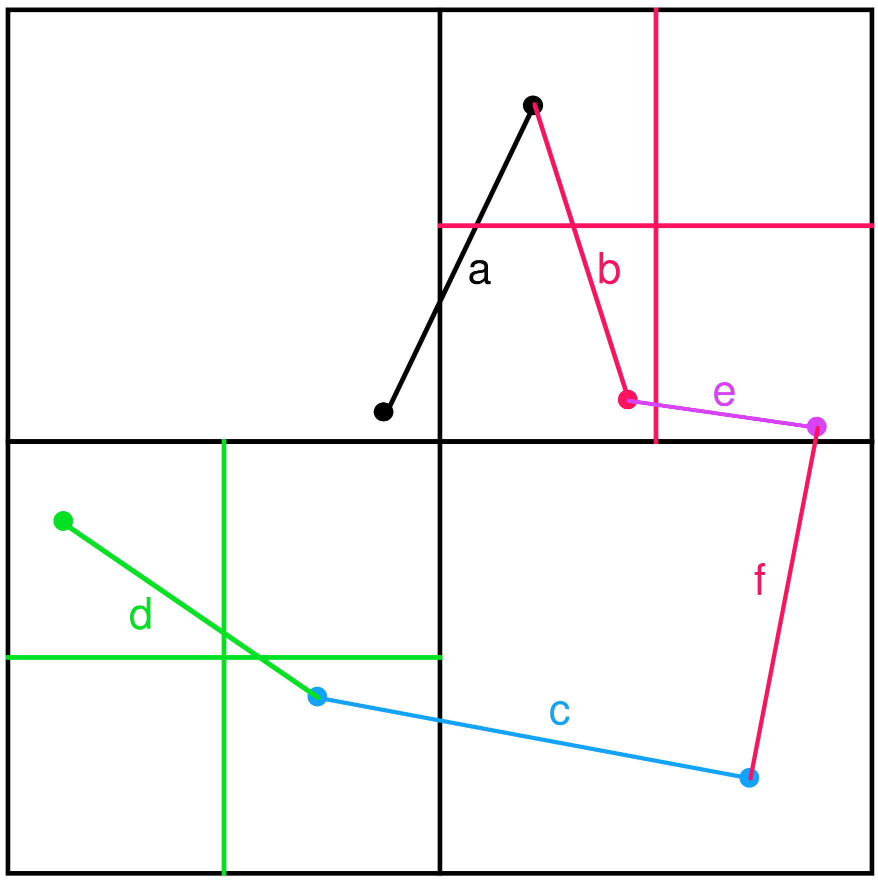
DECOMPOSITION RULE:

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

- Shape independent of order of insertion

PM3 QUADTREE

- Vertex-based (one vertex per block)



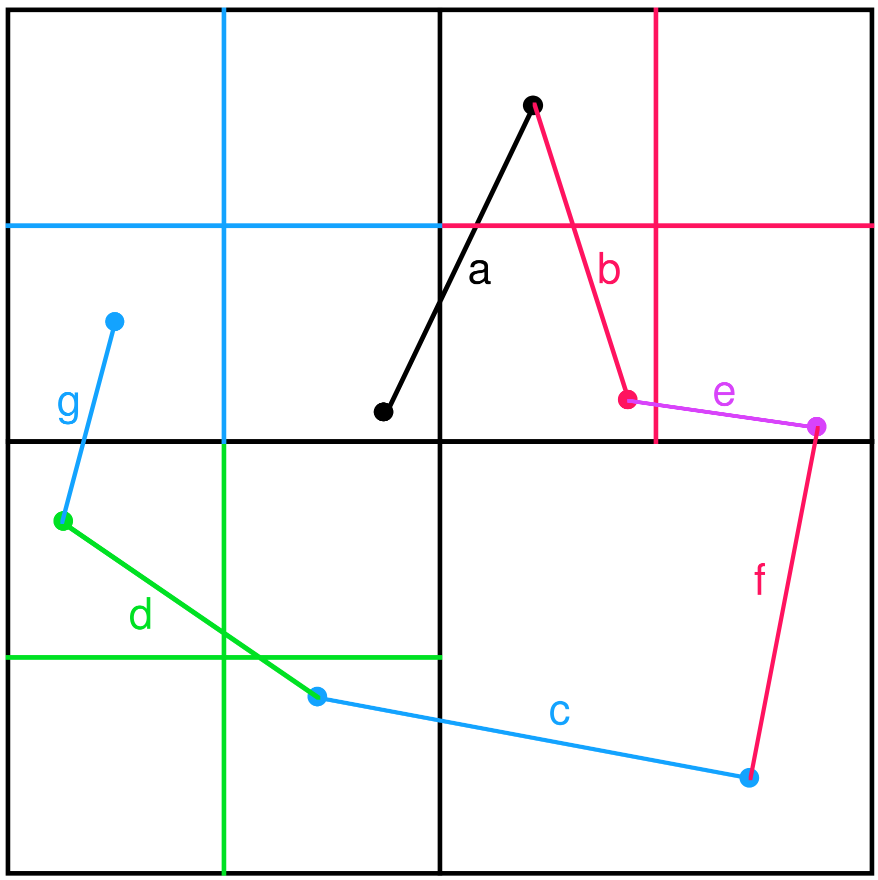
DECOMPOSITION RULE:

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

- Shape independent of order of insertion

PM3 QUADTREE

- Vertex-based (one vertex per block)



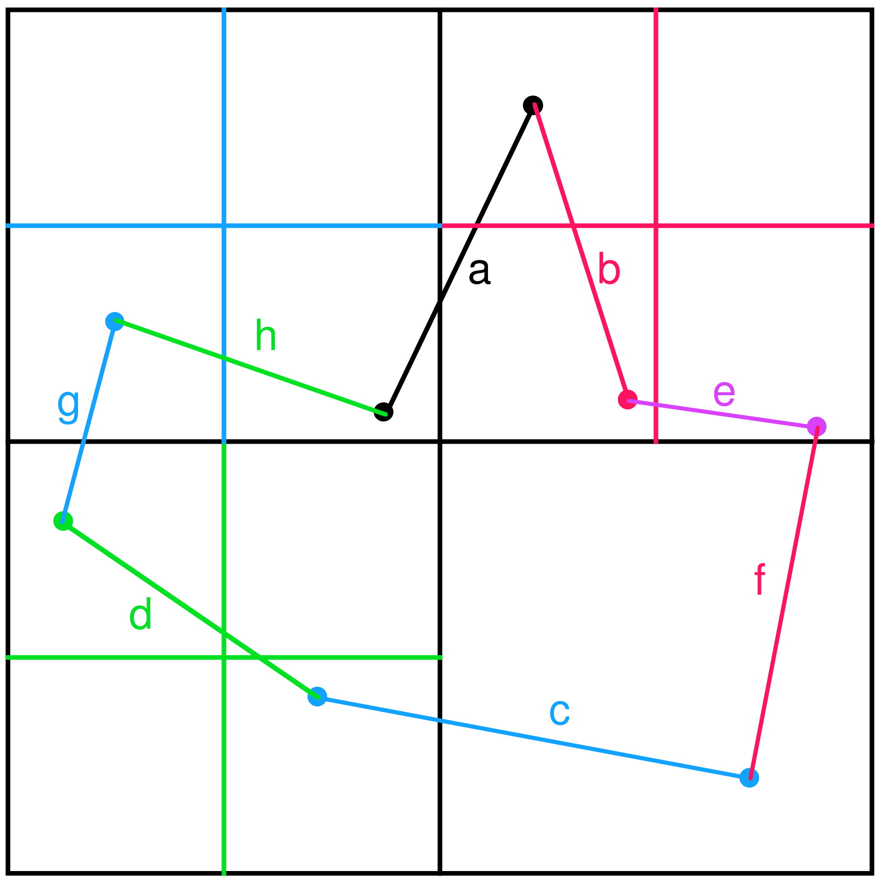
DECOMPOSITION RULE:

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

- Shape independent of order of insertion

PM3 QUADTREE

- Vertex-based (one vertex per block)



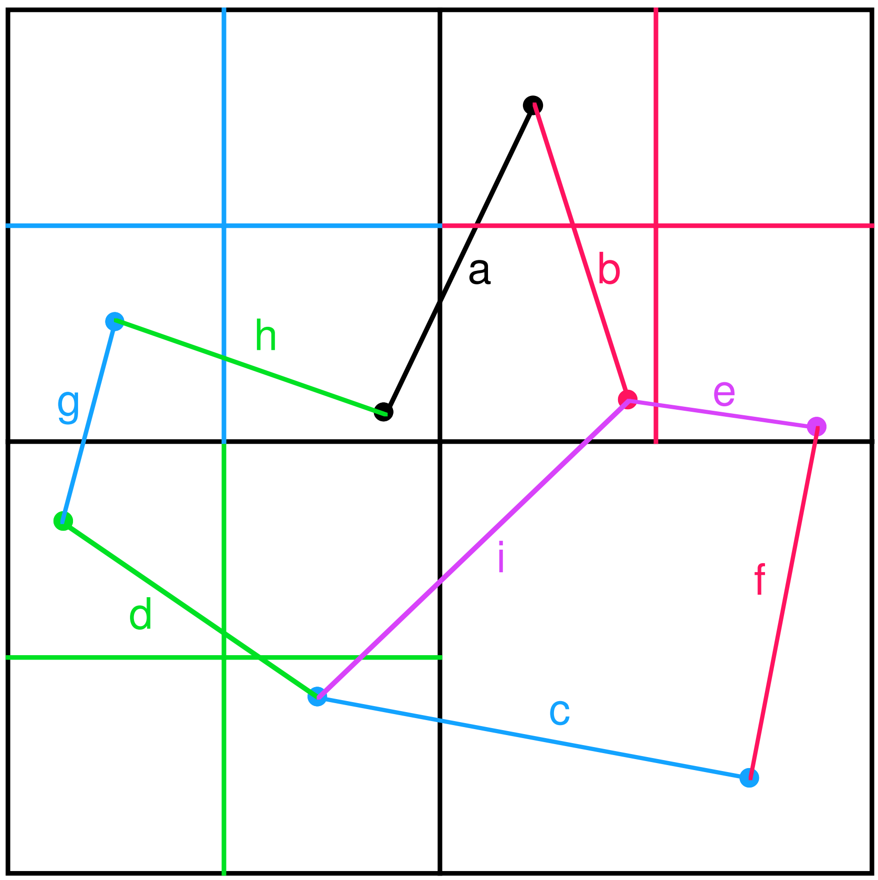
DECOMPOSITION RULE:

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

- Shape independent of order of insertion

PM3 QUADTREE

- Vertex-based (one vertex per block)



DECOMPOSITION RULE:

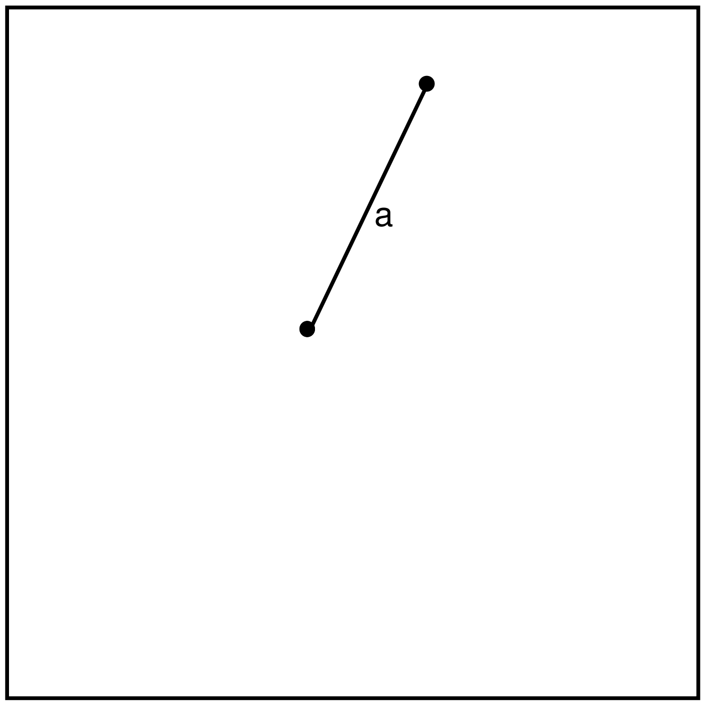
Partitioning occurs when a block contains more than one vertex (i.e., a PR quadtree with edges)

- Shape independent of order of insertion

PMR QUADTREE

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

Ex: $N = 2$



DECOMPOSITION RULE:

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

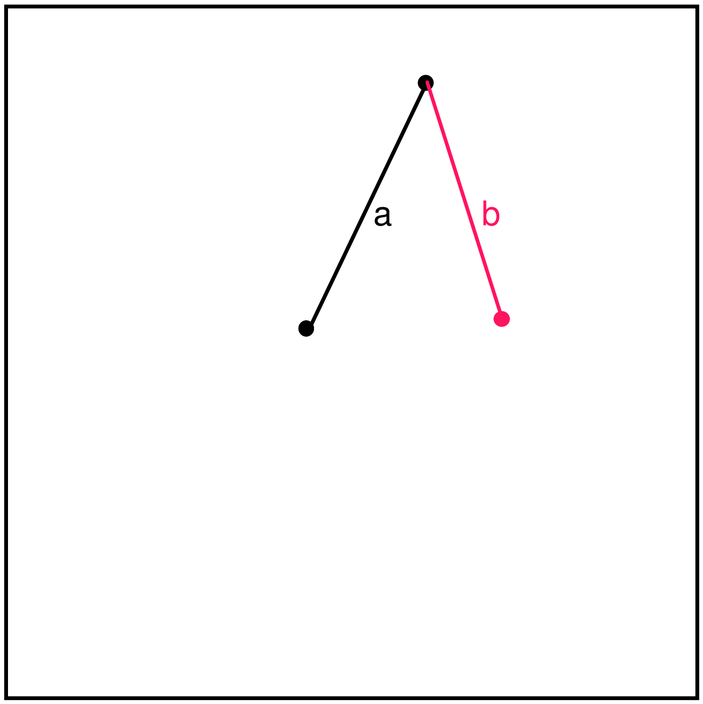
Merge a block with its siblings if the total number of line segments intersecting them is less than N

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

PMR QUADTREE

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

Ex: $N = 2$



DECOMPOSITION RULE:

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

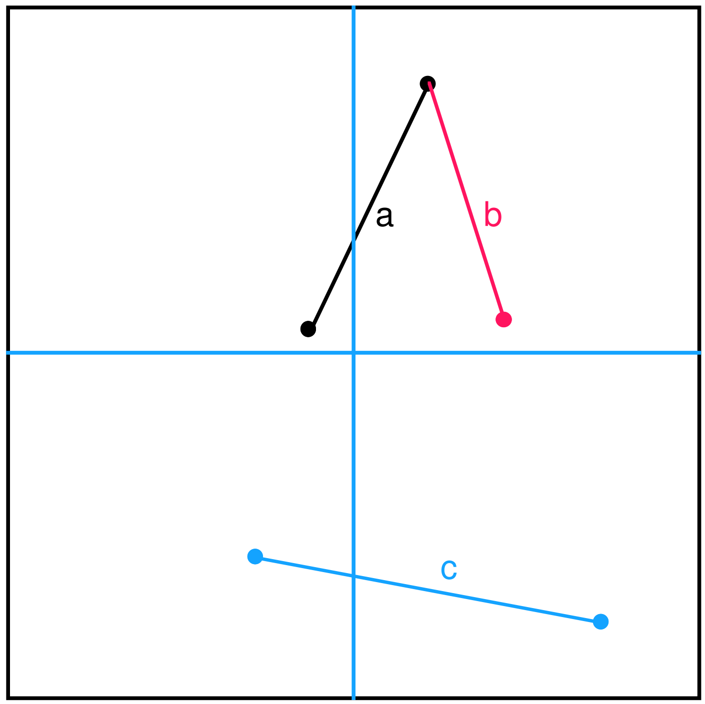
Merge a block with its siblings if the total number of line segments intersecting them is less than *N*

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

PMR QUADTREE

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

Ex: $N = 2$



DECOMPOSITION RULE:

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

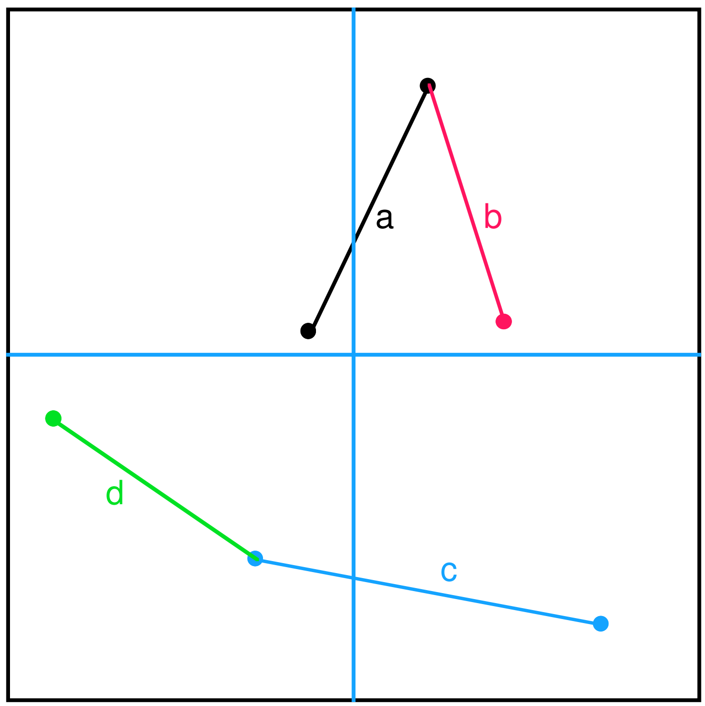
Merge a block with its siblings if the total number of line segments intersecting them is less than *N*

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

PMR QUADTREE

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

Ex: *N* = 2



DECOMPOSITION RULE:

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

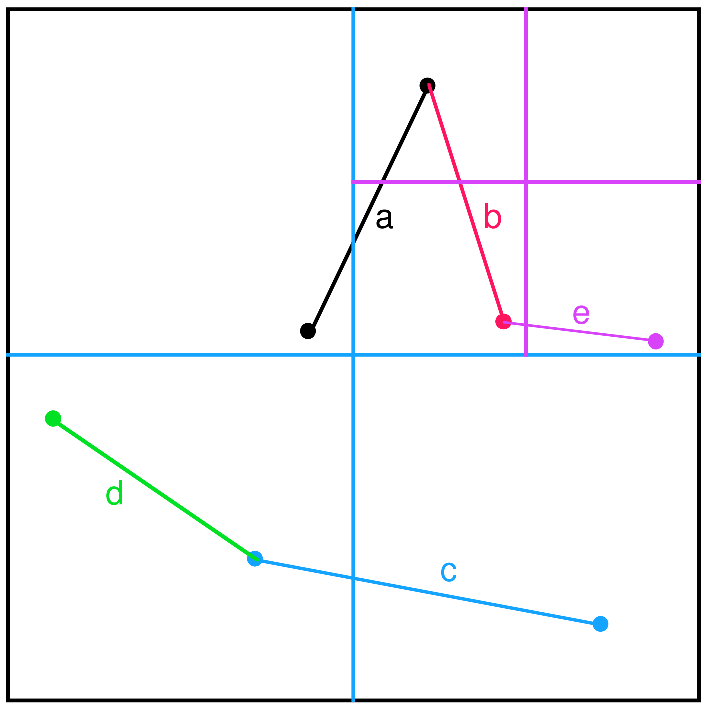
Merge a block with its siblings if the total number of line segments intersecting them is less than *N*

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

PMR QUADTREE

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

Ex: *N* = 2

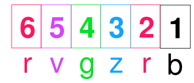


DECOMPOSITION RULE:

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

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

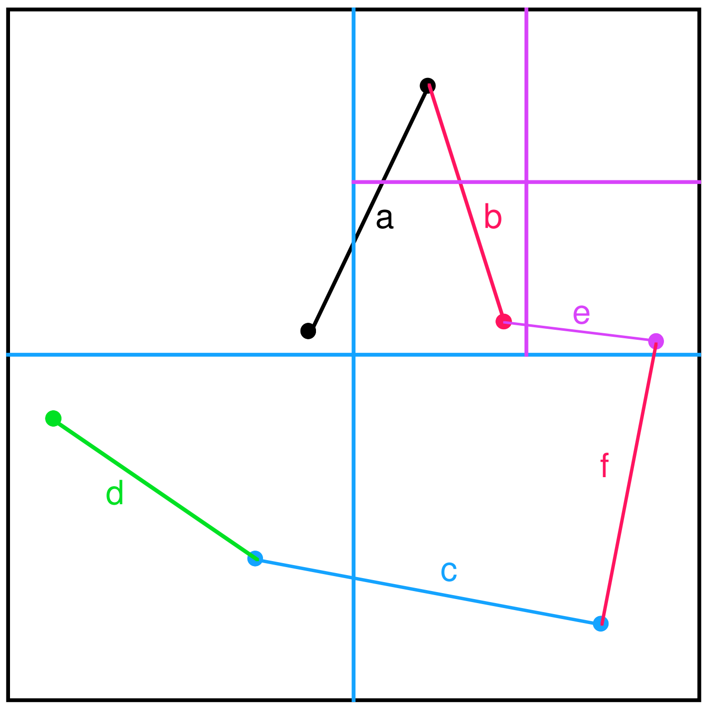
- Merges can be performed more than once
- Does not guarantee that each block will contain at most *N* line segments
- Splitting threshold is not the same as bucket capacity
- Shape depends on order of insertion



PMR QUADTREE

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

Ex: $N = 2$



DECOMPOSITION RULE:

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

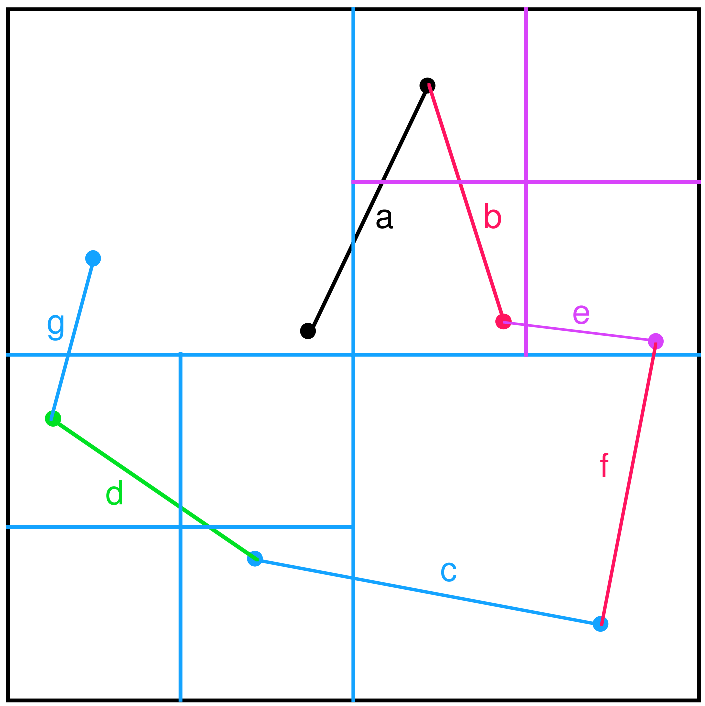
Merge a block with its siblings if the total number of line segments intersecting them is less than *N*

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

PMR QUADTREE

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

Ex: $N = 2$



DECOMPOSITION RULE:

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

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

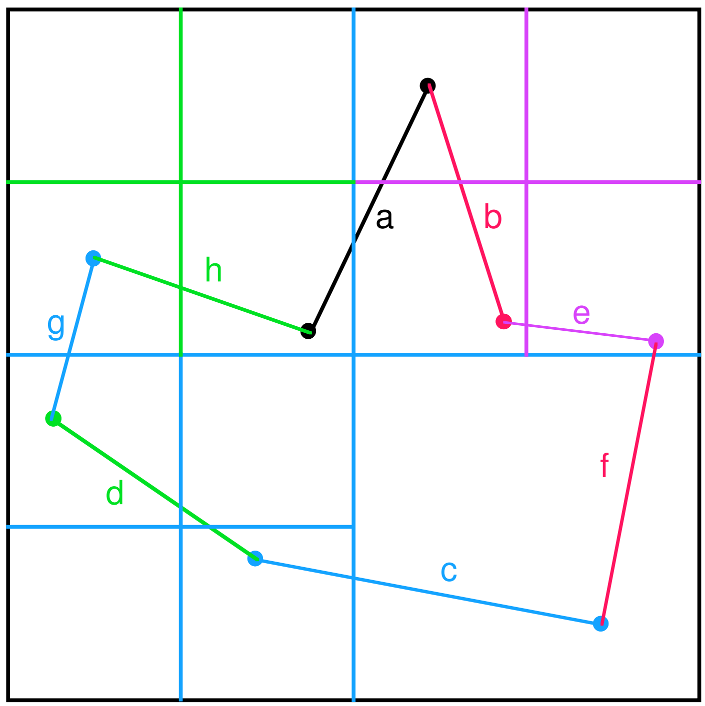
- Merges can be performed more than once
- Does not guarantee that each block will contain at most N line segments
- Splitting threshold is not the same as bucket capacity
- Shape depends on order of insertion

8	7	6	5	4	3	2	1
g	z	r	v	g	z	r	b

PMR QUADTREE

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

Ex: $N = 2$

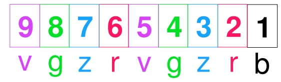


DECOMPOSITION RULE:

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

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

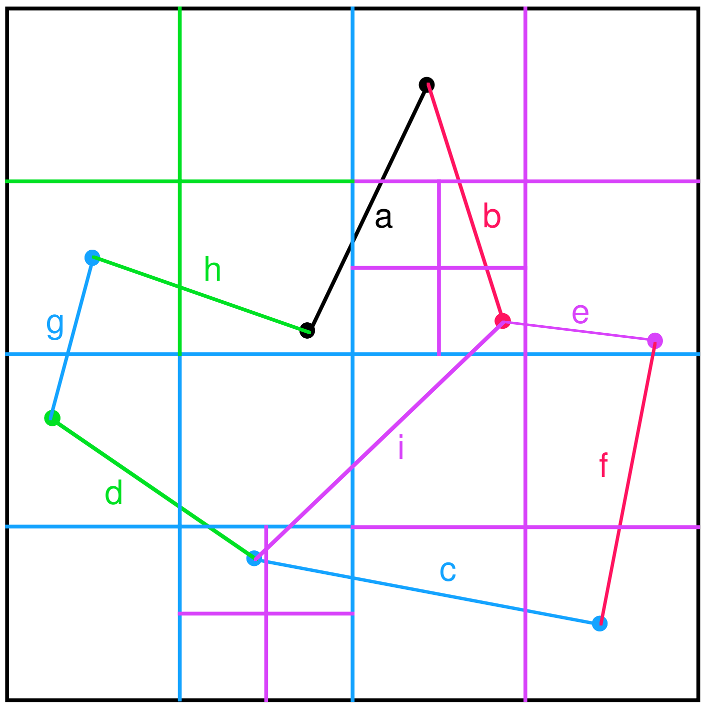
- Merges can be performed more than once
- Does not guarantee that each block will contain at most *N* line segments
- Splitting threshold is not the same as bucket capacity
- Shape depends on order of insertion



PMR QUADTREE

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

Ex: *N* = 2



DECOMPOSITION RULE:

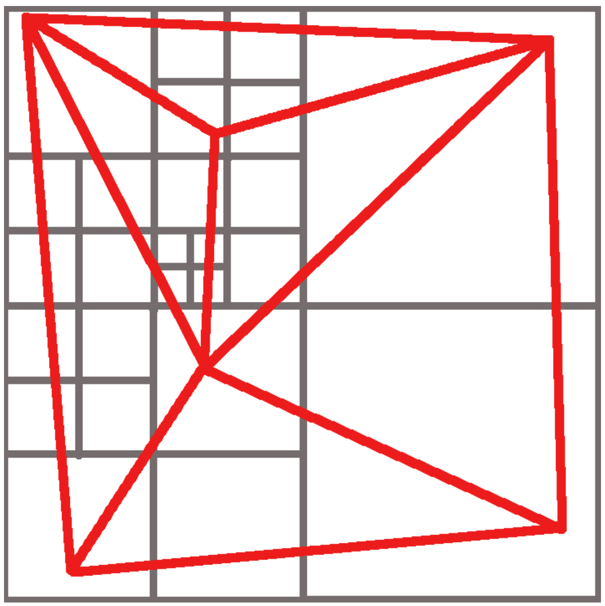
Split a block *once* if upon insertion the number of segments intersecting a block exceeds *N*

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

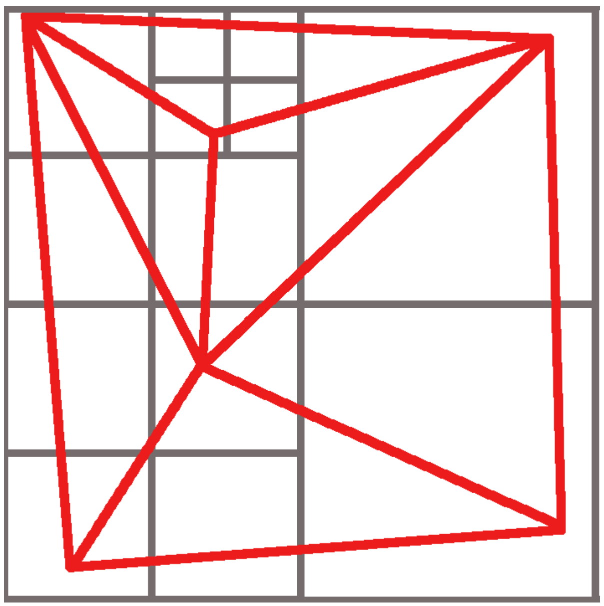
- Merges can be performed more than once
- Does not guarantee that each block will contain at most *N* line segments
- Splitting threshold is not the same as bucket capacity
- Shape depends on order of insertion

Triangulations

- PM_2 quadtree is quite useful vis-a-vis PM_1 quadtree
- Given a triangle table, only need to store at most a single vertex with each cell and can reconstruct mesh with the aid of clipping
- Example triangular mesh



PM_1 quadtree



PM_2 quadtree

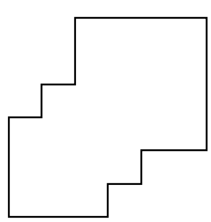
- Can also formulate a PM-triangle quadtree variant

Outline

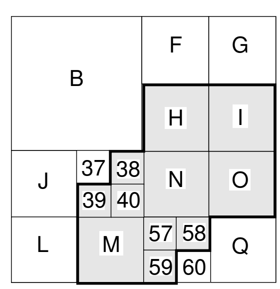
1. Introduction
2. Points
3. Lines
4. Regions, Volumes, and Surfaces
5. Bounding Box Hierarchies
6. Rectangles
7. Surfaces and Volumes
8. Metric Data
9. Operations
10. Indexing Spatiotextual Data
11. Example system

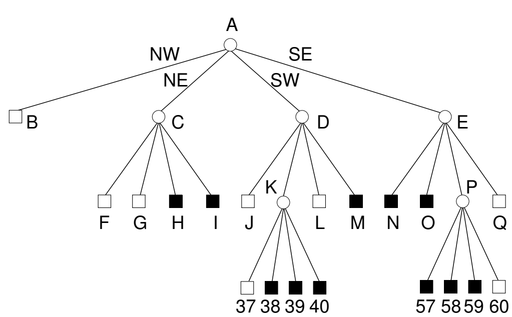
REGION QUADTREE

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



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



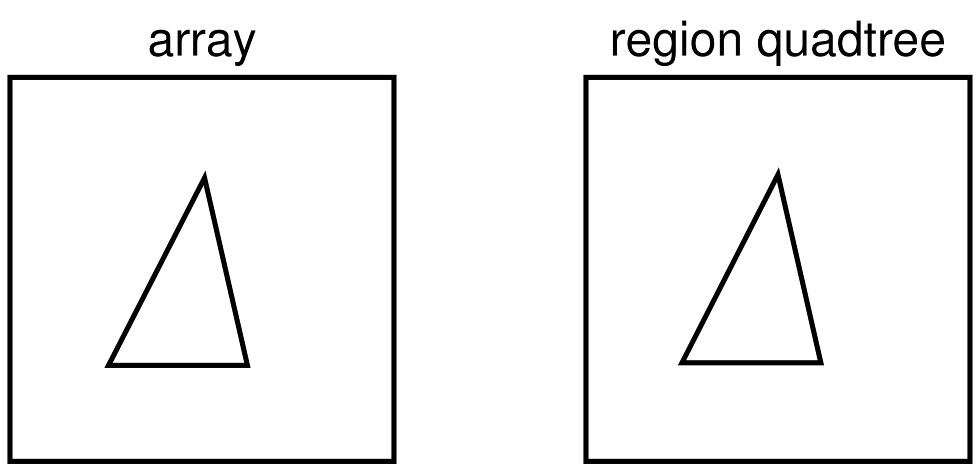


SPACE REQUIREMENTS

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

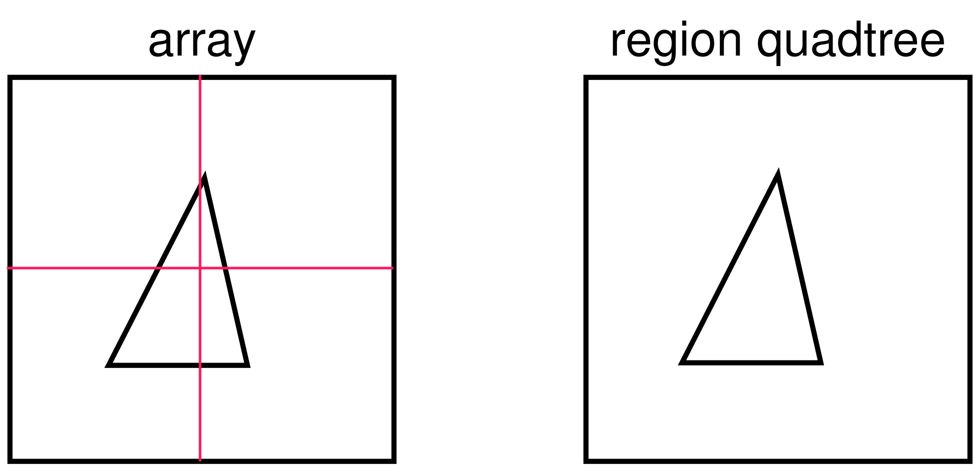
DIMENSION REDUCTION

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



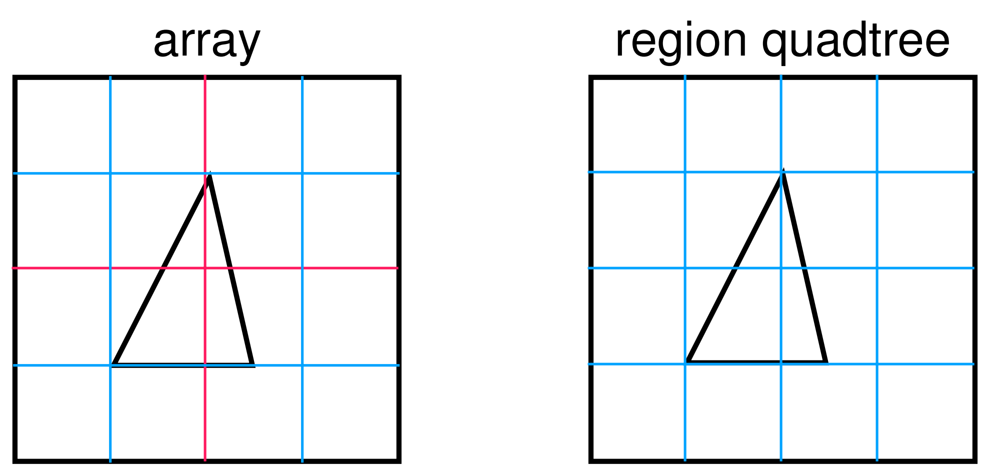
DIMENSION REDUCTION

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

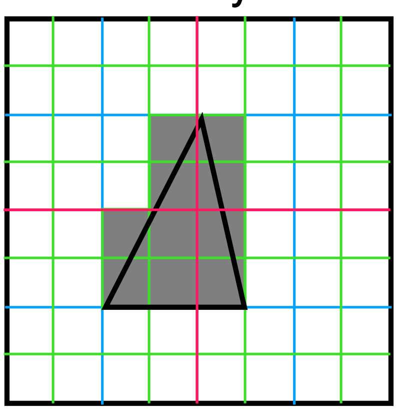
1. Number of blocks necessary to store a simple polygon as a region quadtree is proportional to its perimeter (Hunter)
   - implies that many quadtree algorithms execute in *O*(perimeter) time as they are tree traversals
   - the region quadtree is a dimension reducing device as perimeter (ignoring fractal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



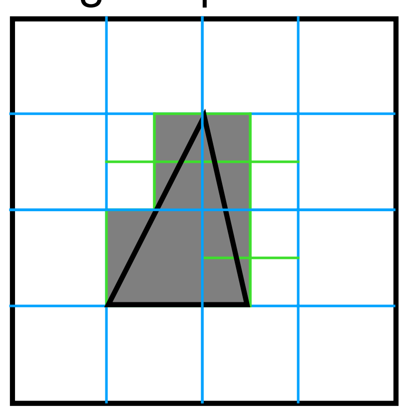
DIMENSION REDUCTION

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.

array

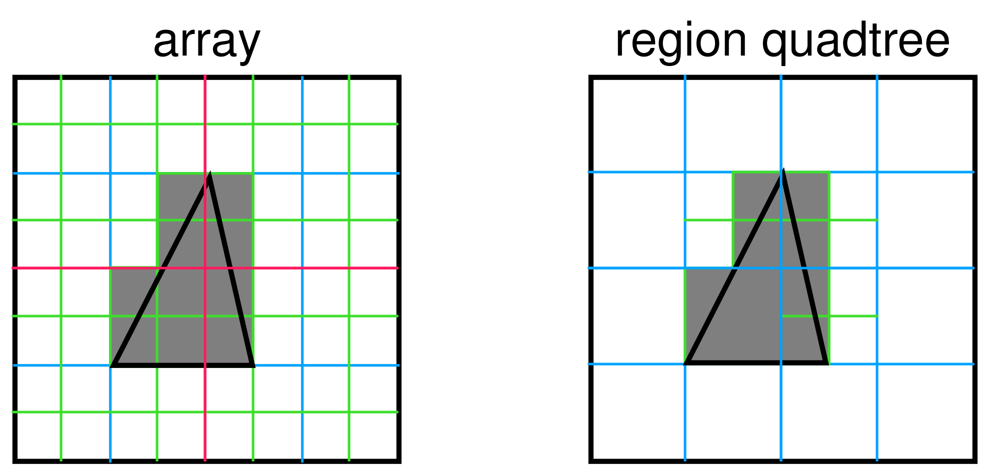


region quadtree



DIMENSION REDUCTION

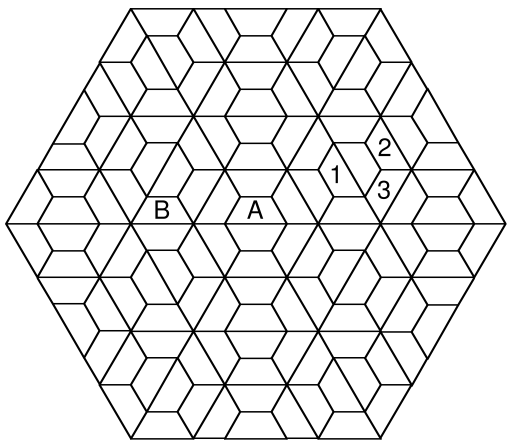
1. Number of blocks necessary to store a simple polygon as a region quadtree is proportional to its perimeter (Hunter)
   - implies that many quadtree algorithms execute in *O*(perimeter) time as they are tree traversals
   - the region quadtree is a dimension reducing device as perimeter (ignoring fractal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

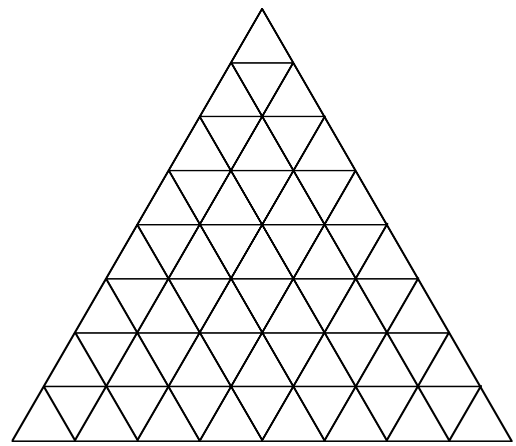


   - easy to see dependence on perimeter as decomposition only takes place on the boundary as the resolution increases

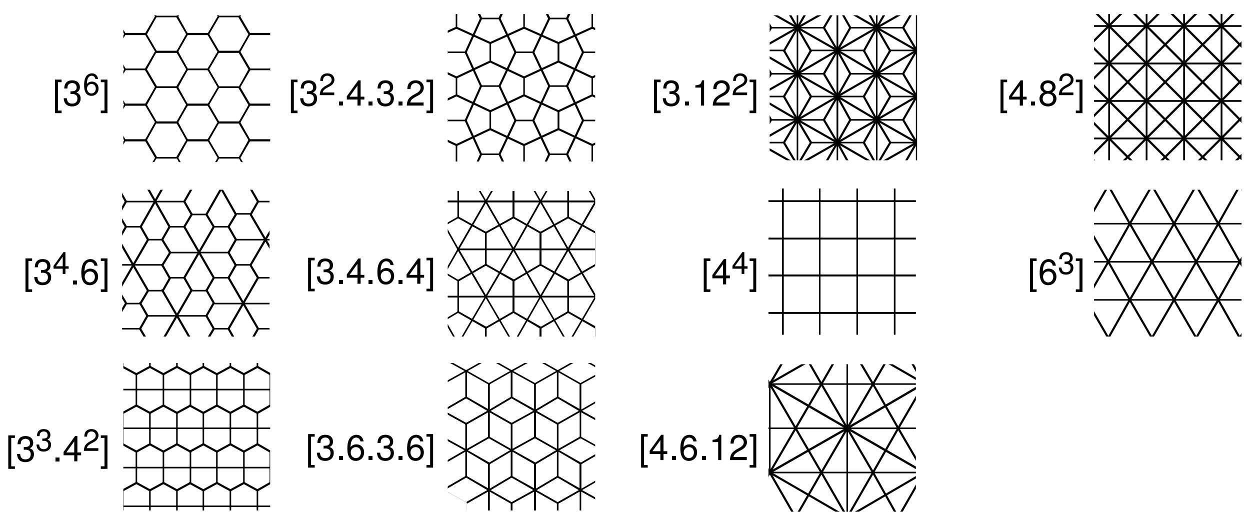
ALTERNATIVE DECOMPOSITION METHODS

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





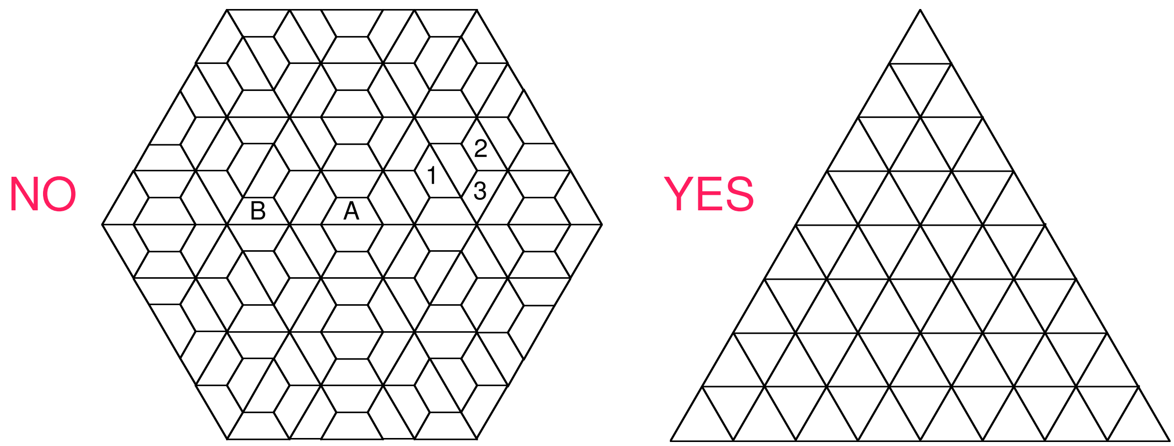
  2. regular — each tile is a regular polygon
- There are 81 types if classify by their symmetry groups
- Only 11 types if classify by their adjacency structure



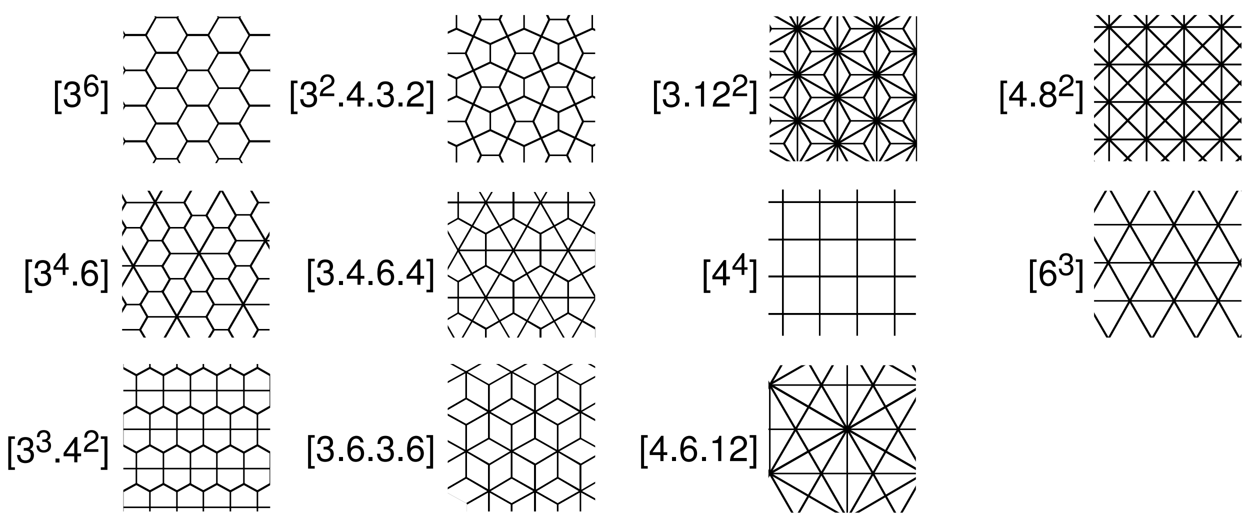
- $[3.12^2]$ means 3 edges at the first vertex of the polygonal tile followed by 12 edges at the next two vertices

ALTERNATIVE DECOMPOSITION METHODS

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



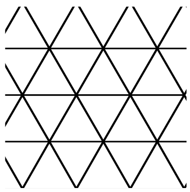
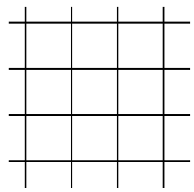
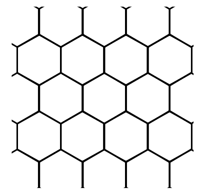
  2. regular — each tile is a regular polygon
- There are 81 types if classify by their symmetry groups
- Only 11 types if classify by their adjacency structure



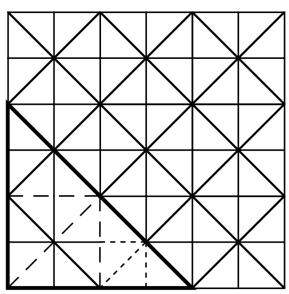
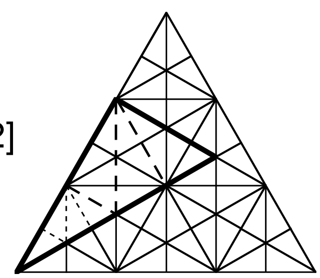
- [3.12^2] means 3 edges at the first vertex of the polygonal tile followed by 12 edges at the next two vertices

PROPERTIES OF TILINGS — SIMILARITY

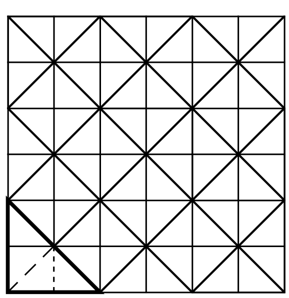
- Similarity — a tile at level *k* has the same shape as a tile at level 0 (basic tile shape)

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

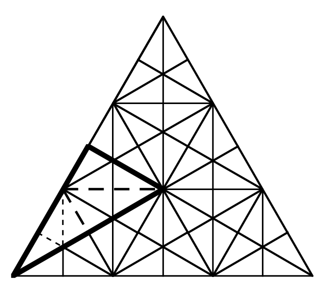
- Limited ≡ NOT similar (i.e., cannot be decomposed infinitely into smaller tiles of the same shape)
- Unlimited: each edge of each tile lies on an infinitely straight line composed entirely of edges
- Only 4 unlimited tilings [4^4], [6^3], [4.8^2], and [4.6.12]

[4.8^2]  [4.6.12] 

- Two additional hierarchies:



rotation of 135° between levels

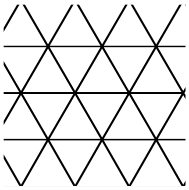
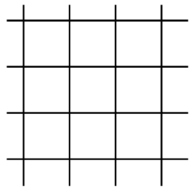
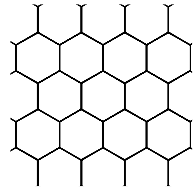


reflection between levels

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

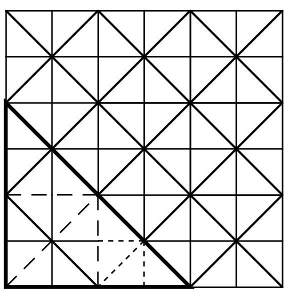
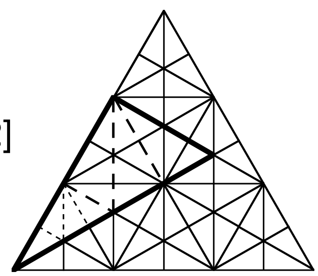
PROPERTIES OF TILINGS — SIMILARITY

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

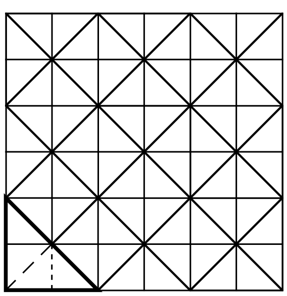
[6^3]  [4^4]  [3^6] 

YES YES NO

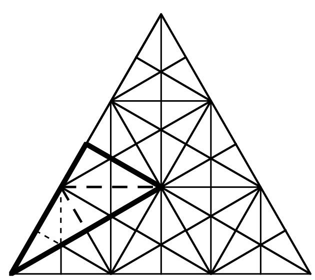
- Limited ≡ NOT similar (i.e., cannot be decomposed infinitely into smaller tiles of the same shape)
- Unlimited: each edge of each tile lies on an infinitely straight line composed entirely of edges
- Only 4 unlimited tilings [4^4], [6^3], [4.8^2], and [4.6.12]

[4.8^2]  [4.6.12] 

- Two additional hierarchies:



rotation of 135° between levels



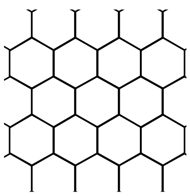
reflection between levels

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

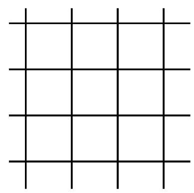
PROPERTIES OF TILINGS — ADJACENCY

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

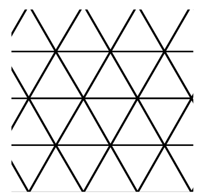
[3^6]



[4^4]

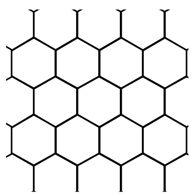


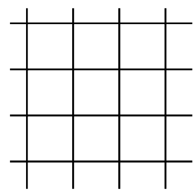
[6^3]

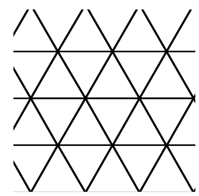


PROPERTIES OF TILINGS — ADJACENCY

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

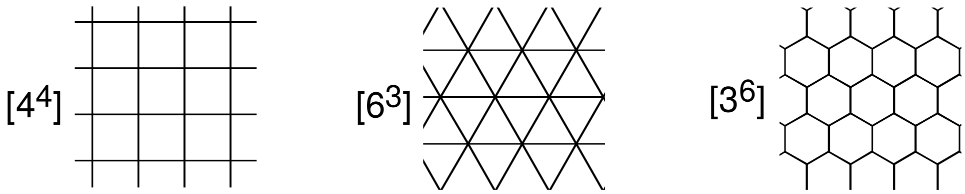
$[3^6]$

A=1

$[4^4]$

A=2

$[6^3]$

A=3

PROPERTIES OF TILINGS — UNIFORM ORIENTATION

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



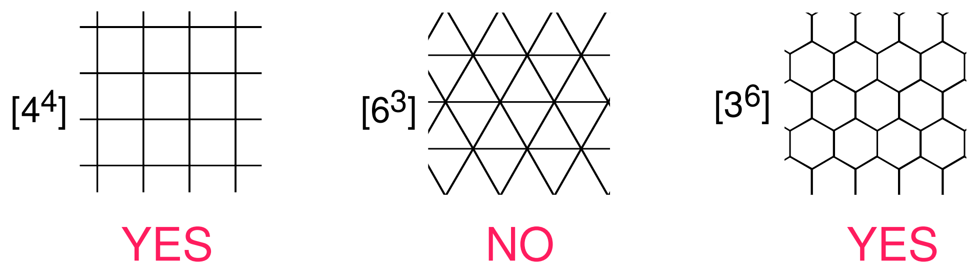
Conclusion:

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

  Use $[4^4]$!

PROPERTIES OF TILINGS — UNIFORM ORIENTATION

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



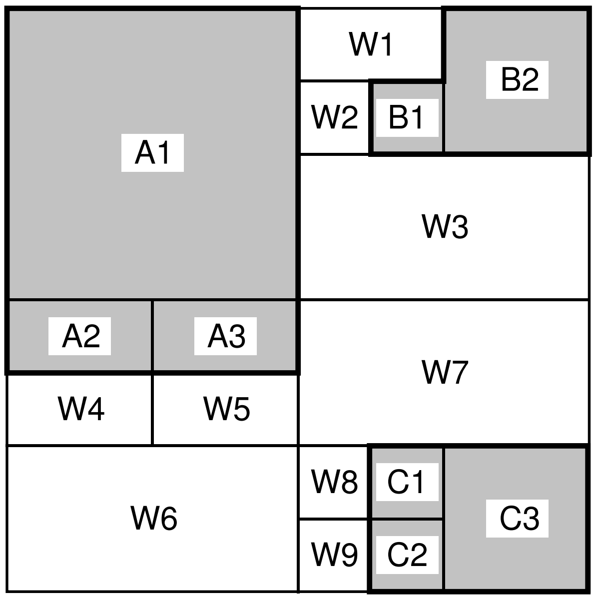
Conclusion:

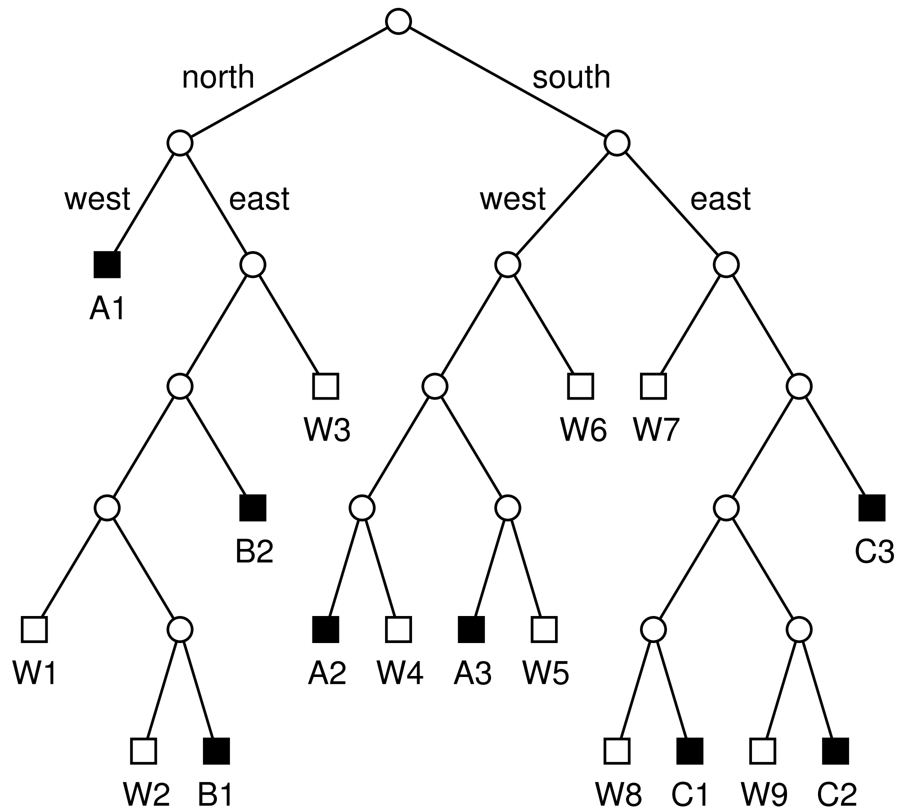
- [4^4] has a lower adjacency number than [6^3]
- [4^4] has a uniform orientation while [6^3] does not
- [4^4] is unlimited while [3^6] is limited

  Use [4^4]!

Bintree

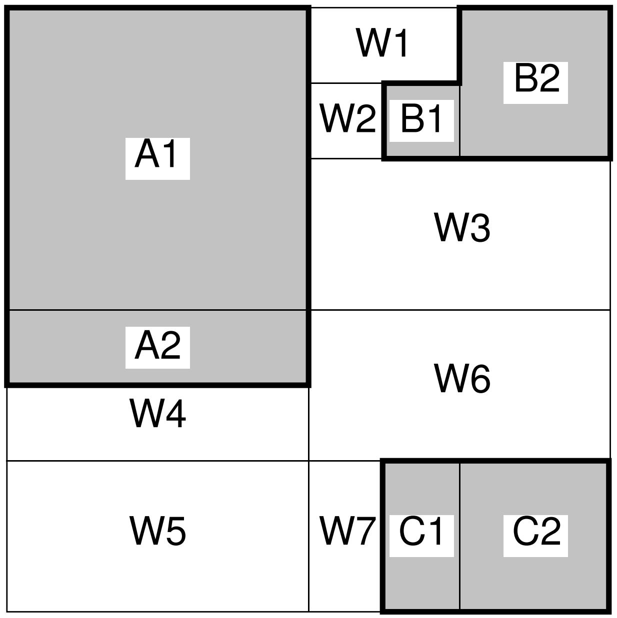
- Regular decomposition k-d tree
- Cycle through attributes

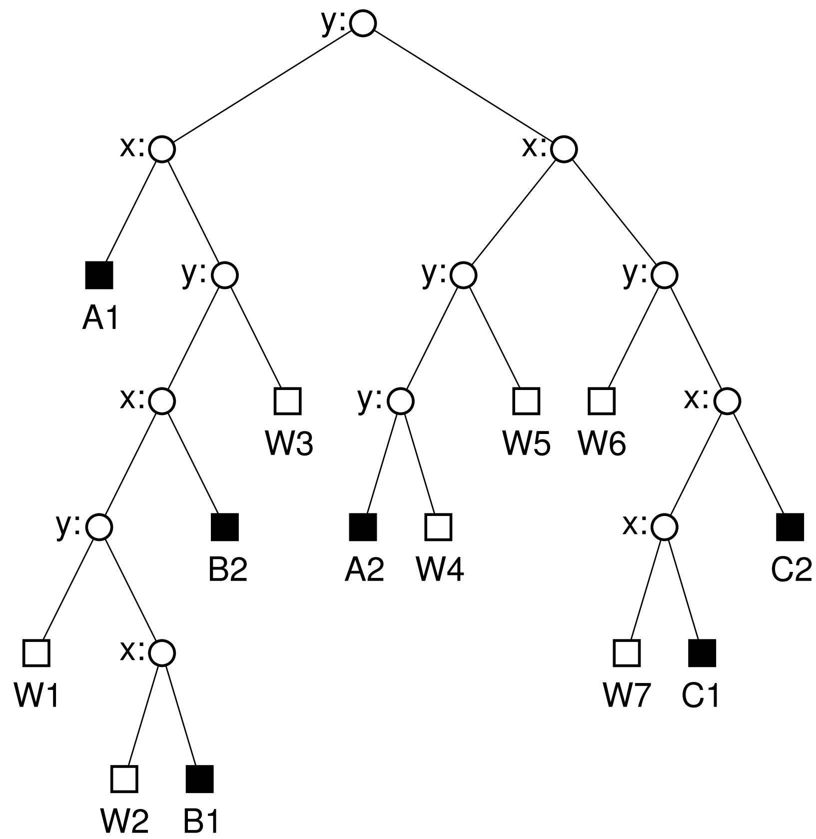




Generalized Bintree

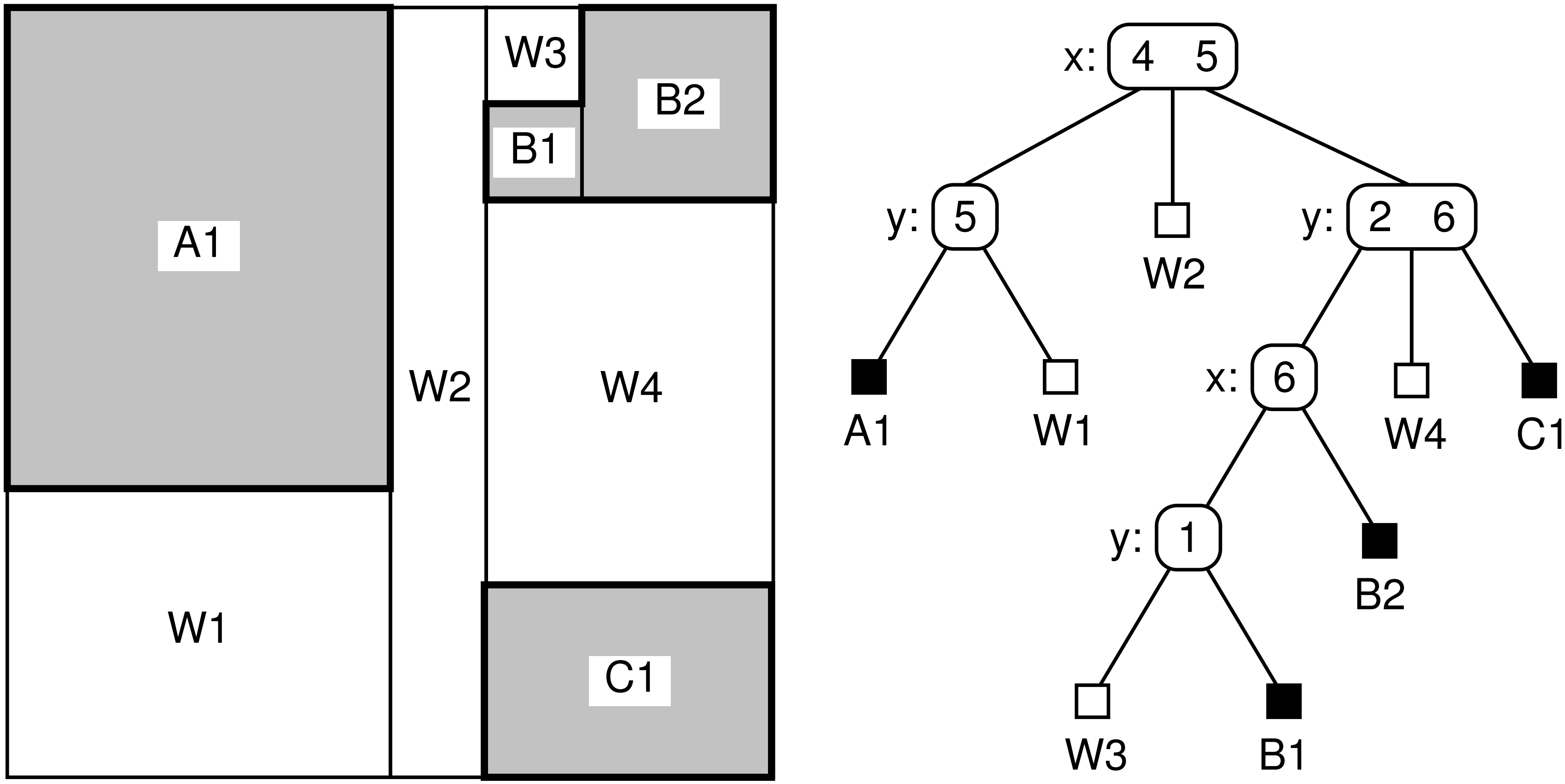
- Regular decomposition k-d tree but no need to cycle through attributes
- Need to record identity of partition axis at each nonleaf node





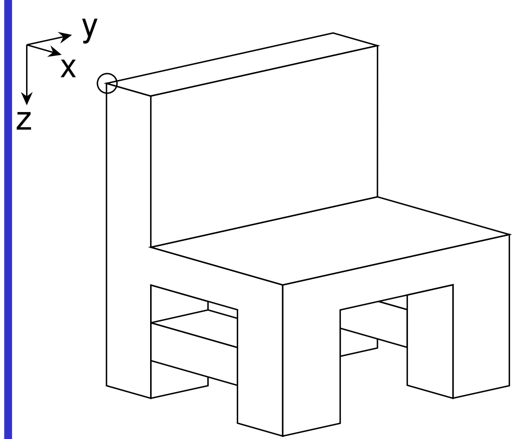
X-Y Tree, Treemap, and Puzzletree

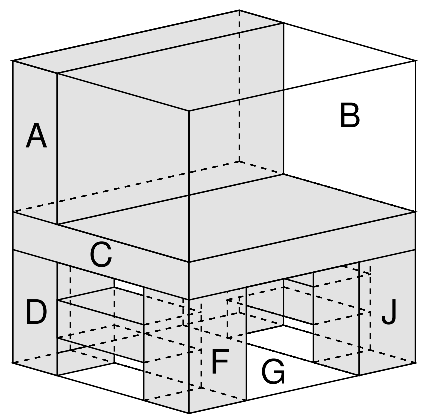
- Split into two or more parts at each partition step
- Implies no two successive partitions along the same attribute as they are combined
- Implies cycle through attributes in two dimensions

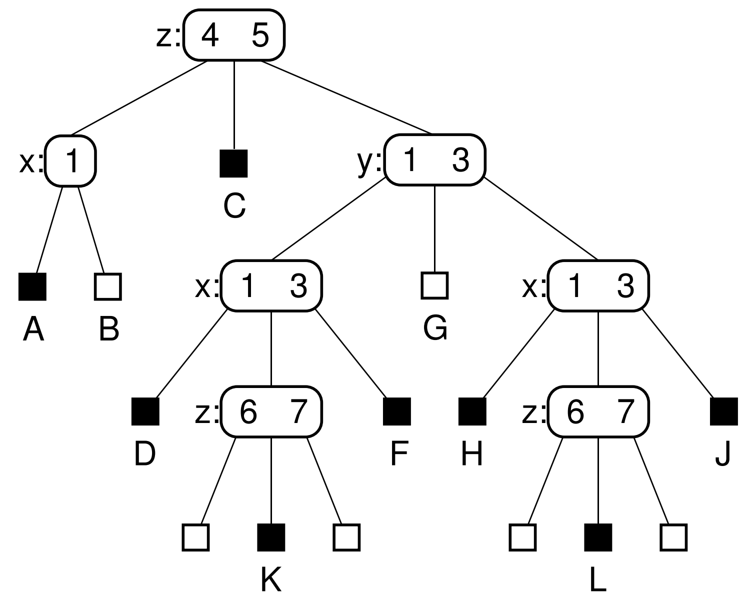


Three-Dimensional X-Y Tree, Treemap, and Puzzletree

- No longer require cycling through dimensions as this results in losing some perceptually appealing block combinations

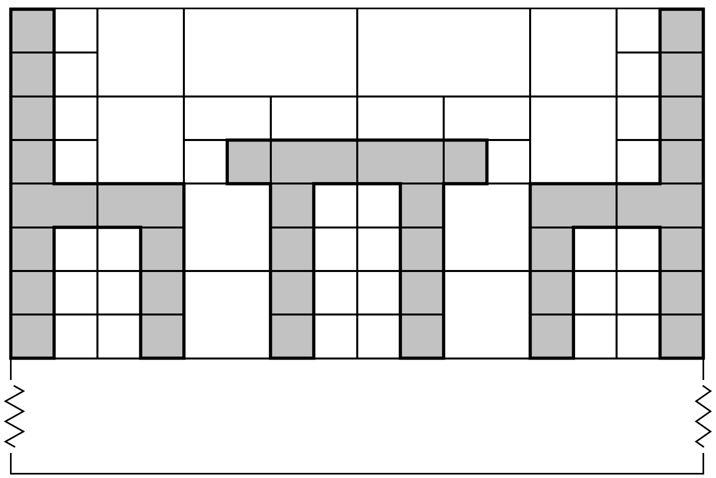




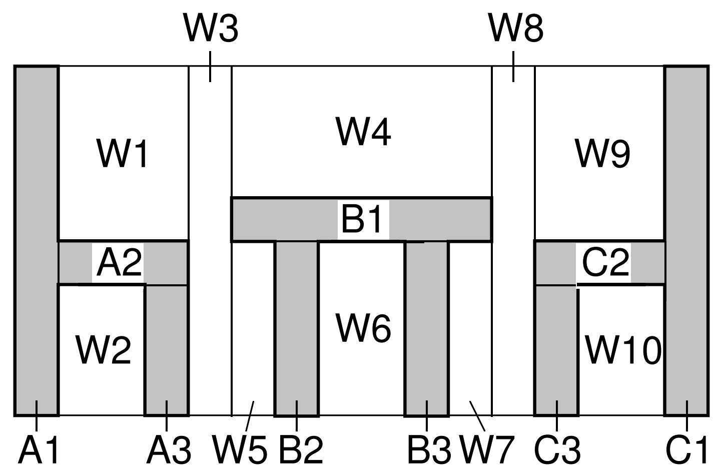


Bintree compared with X-Y Tree, Treemap, Puzzletree

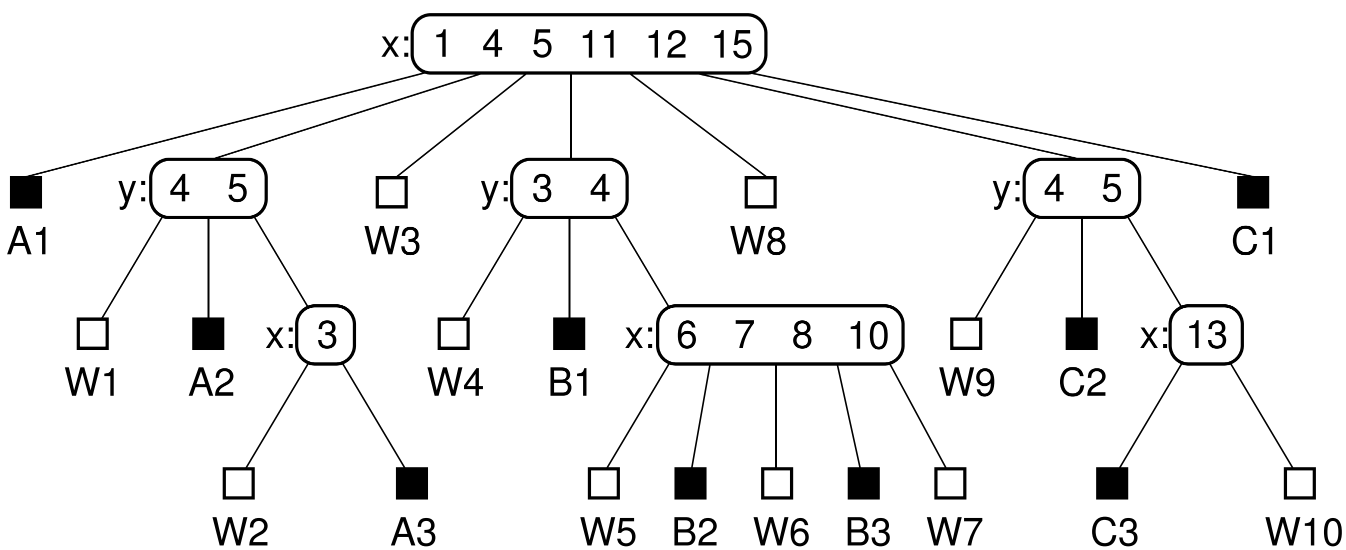
- Much more decomposition in bintree



Bintree

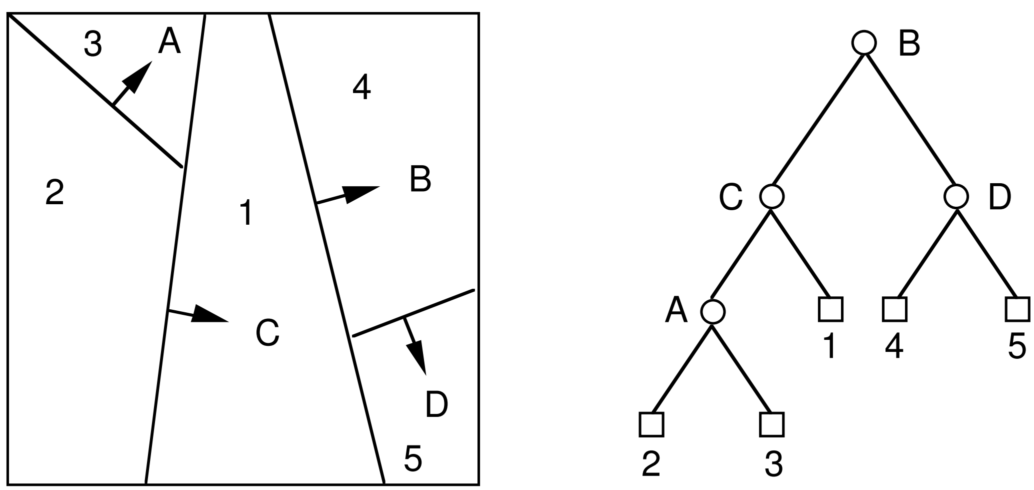


X-Y Tree



BSP TREES (Fuchs, Kedem, Naylor)

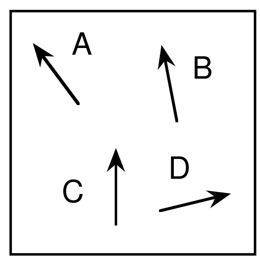
- Like a bintree except that the decomposition lines are at arbitrary orientations (i.e., they need not be parallel or orthogonal)
- For data of arbitrary dimensions
- In 2D (3D), partition along the edges (faces) of a polygon (polyhedron)
- Ex: arrows indicate direction of positive area



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

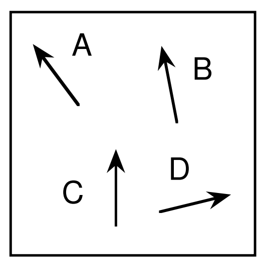
DRAWBACKS OF BSP TREES

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

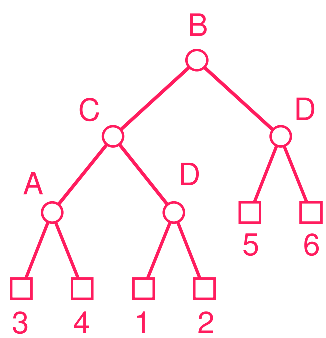


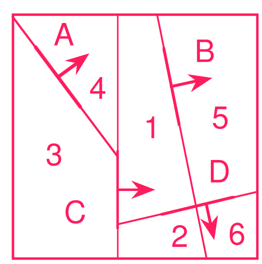
DRAWBACKS OF BSP TREES

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



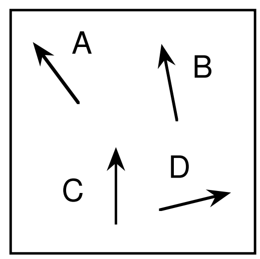
1. partition induced by choosing B as the root



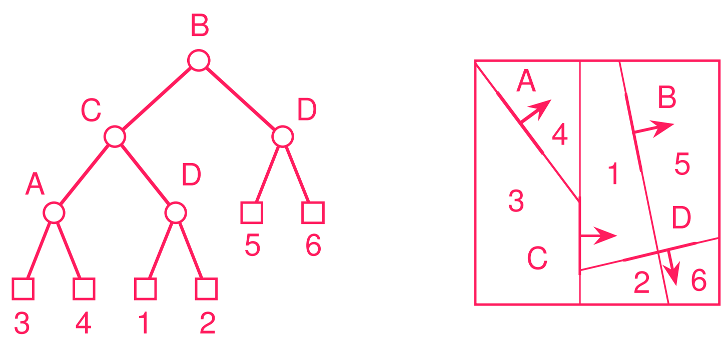


DRAWBACKS OF BSP TREES

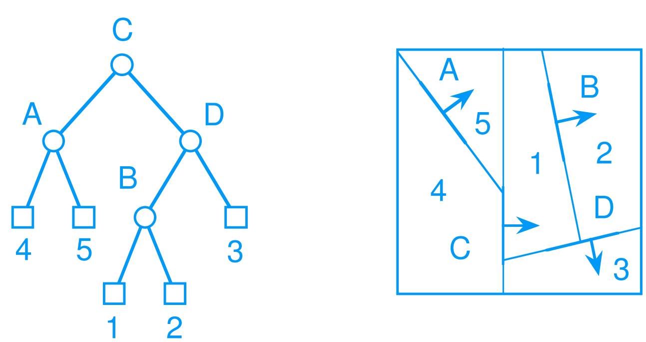
- A polygon may be included in both the left and right subtrees of node
- Same issues of duplicate reporting as in representations based on a disjoint decomposition of the underlying space
- Shape of the BSP tree depends on the order in which the polygons are processed and on the polygons chosen to serve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



1. partition induced by choosing B as the root



2. partition induced by choosing C as the root

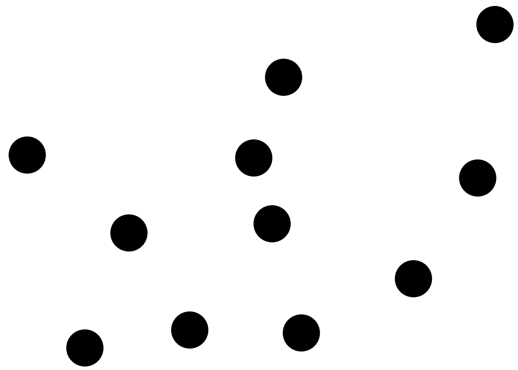


Outline

1. Introduction
2. Points
3. Lines
4. Regions, Volumes, and Surfaces
5. Bounding Box Hierarchies
6. Rectangles
7. Surfaces and Volumes
8. Metric Data
9. Operations
10. Indexing Spatiotextual Data
11. Example system

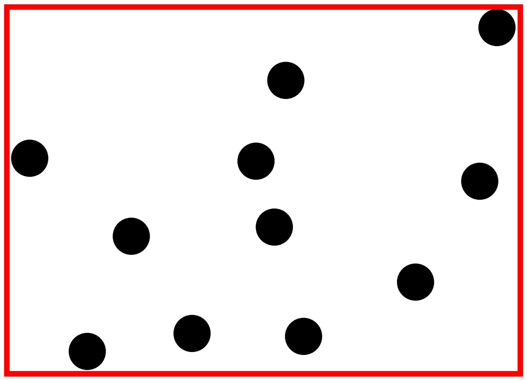
Bounding Box Hierarchies

1. Axis-aligned bounding boxes (AABB)
2. Oriented bounding boxes (OBB)
   - Arbitrary orientation for bounding hyperrectangles
3. Minimum bounding hyperspheres (sphere tree, SS-tree)
4. Combination of hyperspheres and hyperrectangles (SR-tree)
5. 3-dimensional pie slices (BOXTREE)
6. Truncated tetrahedra (prism tree)



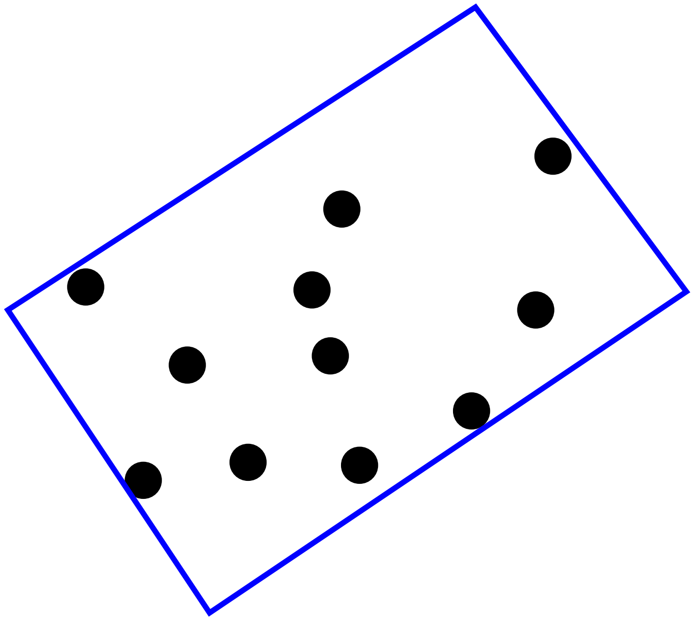
Bounding Box Hierarchies

1. Axis-aligned bounding boxes (AABB)
2. Oriented bounding boxes (OBB)
   - Arbitrary orientation for bounding hyperrectangles
3. Minimum bounding hyperspheres (sphere tree, SS-tree)
4. Combination of hyperspheres and hyperrectangles (SR-tree)
5. 3-dimensional pie slices (BOXTREE)
6. Truncated tetrahedra (prism tree)



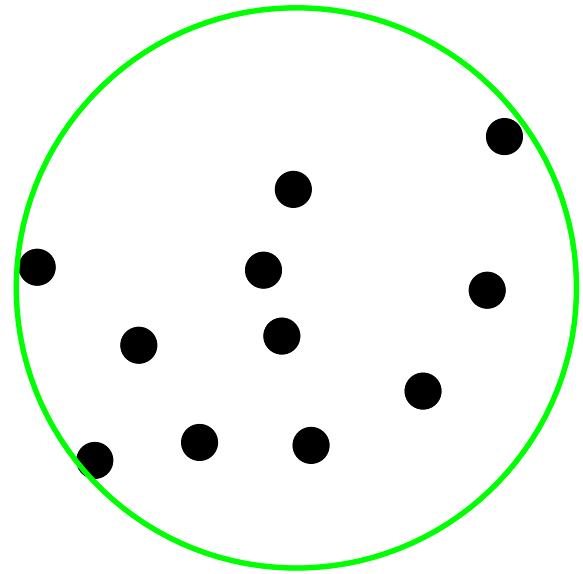
Bounding Box Hierarchies

1. Axis-aligned bounding boxes (AABB)
2. Oriented bounding boxes (OBB)
   - Arbitrary orientation for bounding hyperrectangles
3. Minimum bounding hyperspheres (sphere tree, SS-tree)
4. Combination of hyperspheres and hyperrectangles (SR-tree)
5. 3-dimensional pie slices (BOXTREE)
6. Truncated tetrahedra (prism tree)



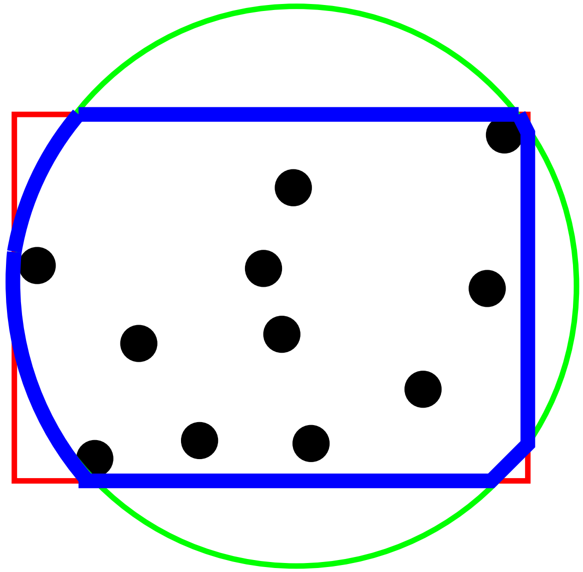
Bounding Box Hierarchies

1. Axis-aligned bounding boxes (AABB)
2. Oriented bounding boxes (OBB)
   - Arbitrary orientation for bounding hyperrectangles
3. Minimum bounding hyperspheres (sphere tree, SS-tree)
4. Combination of hyperspheres and hyperrectangles (SR-tree)
5. 3-dimensional pie slices (BOXTREE)
6. Truncated tetrahedra (prism tree)



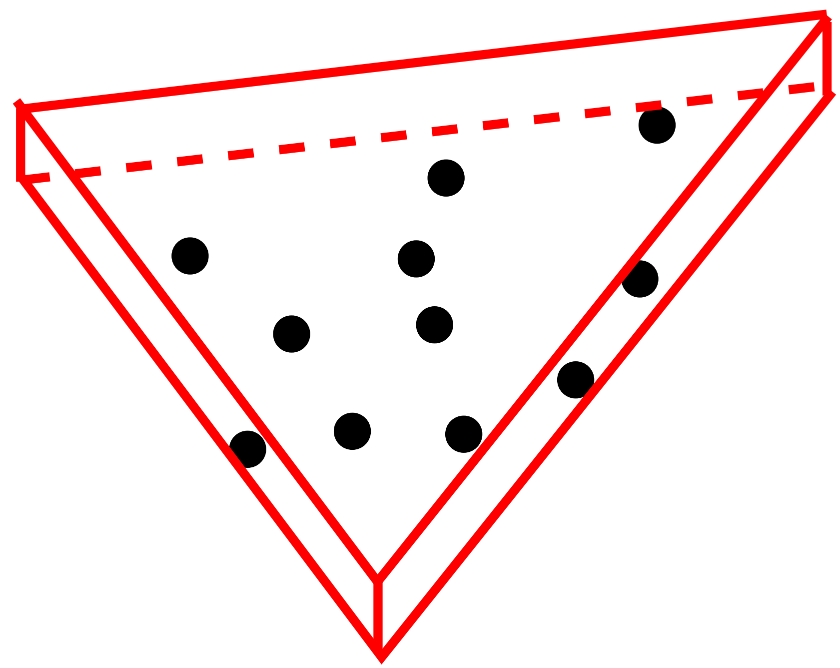
Bounding Box Hierarchies

1. Axis-aligned bounding boxes (AABB)
2. Oriented bounding boxes (OBB)
   - Arbitrary orientation for bounding hyperrectangles
3. Minimum bounding hyperspheres (sphere tree, SS-tree)
4. Combination of hyperspheres and hyperrectangles (SR-tree)
5. 3-dimensional pie slices (BOXTREE)
6. Truncated tetrahedra (prism tree)



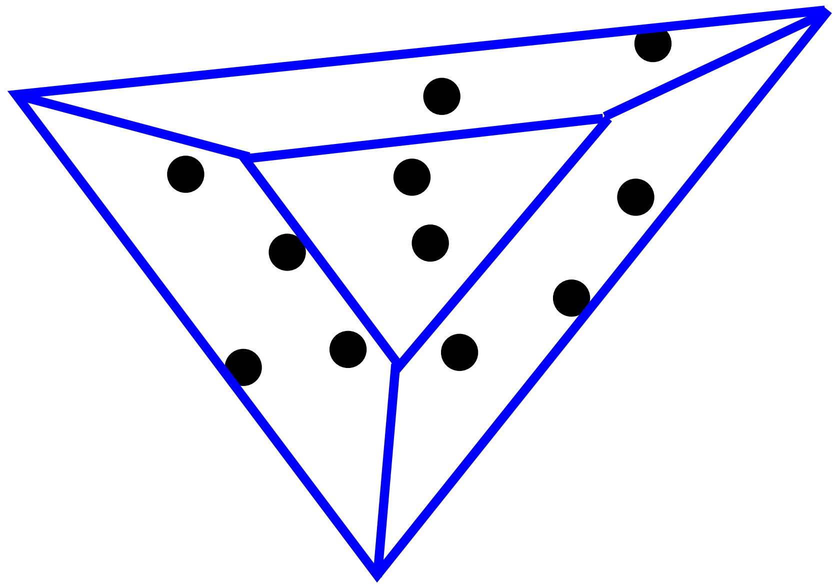
Bounding Box Hierarchies

1. Axis-aligned bounding boxes (AABB)
2. Oriented bounding boxes (OBB)
   - Arbitrary orientation for bounding hyperrectangles
3. Minimum bounding hyperspheres (sphere tree, SS-tree)
4. Combination of hyperspheres and hyperrectangles (SR-tree)
5. 3-dimensional pie slices (BOXTREE)
6. Truncated tetrahedra (prism tree)



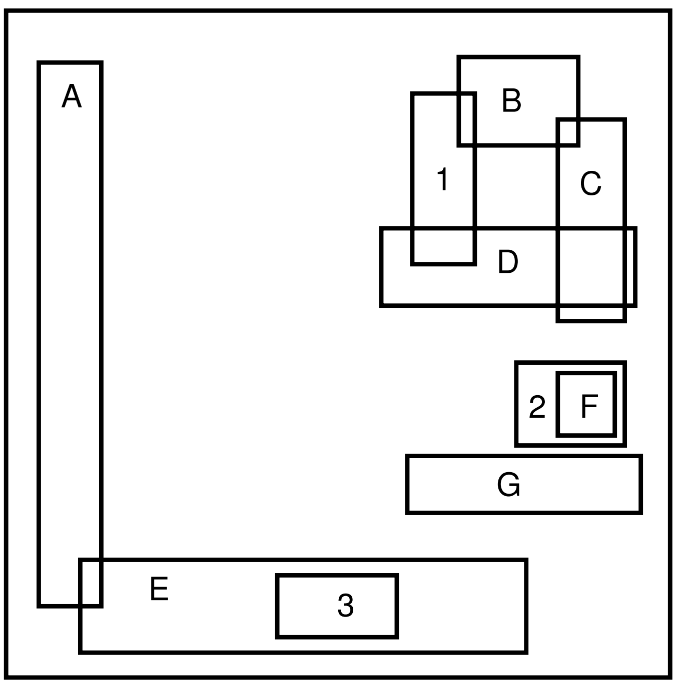
Bounding Box Hierarchies

1. Axis-aligned bounding boxes (AABB)
2. Oriented bounding boxes (OBB)
   - Arbitrary orientation for bounding hyperrectangles
3. Minimum bounding hyperspheres (sphere tree, SS-tree)
4. Combination of hyperspheres and hyperrectangles (SR-tree)
5. 3-dimensional pie slices (BOXTREE)
6. Truncated tetrahedra (prism tree)



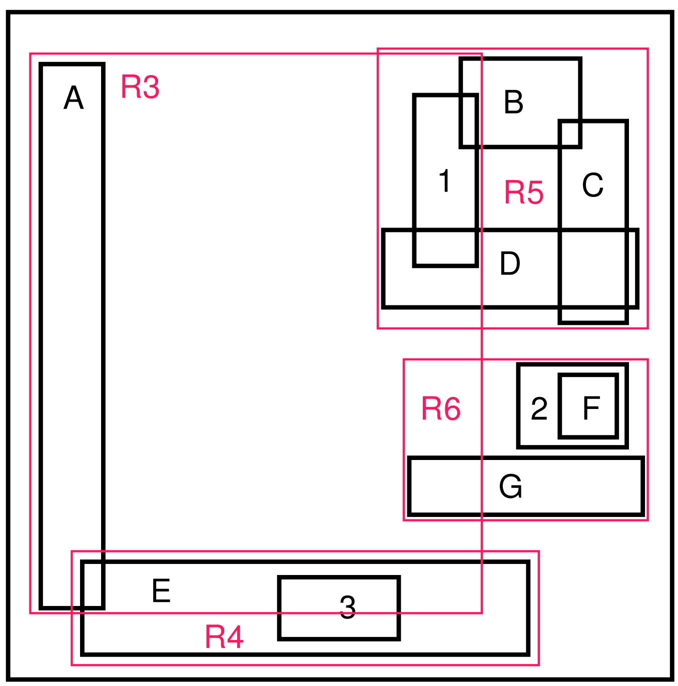
MINIMUM BOUNDING RECTANGLES

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



MINIMUM BOUNDING RECTANGLES

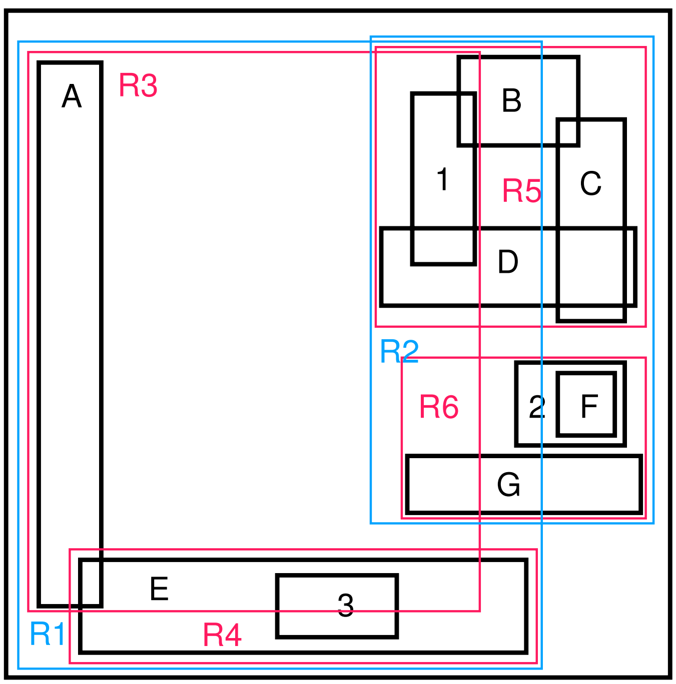
- Rectangles grouped into hierarchies, stored in another structure such as a B-tree
- Drawback: not a disjoint decomposition of space
- Rectangle has single bounding rectangle, yet area it spans may be included in several bounding rectangles
- May have to visit several rectangles to determine the presence/absence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



R3: | A | 1 | | R4: | E | 3 | | R5: | B | C | D | R6: | 2 | F | G |

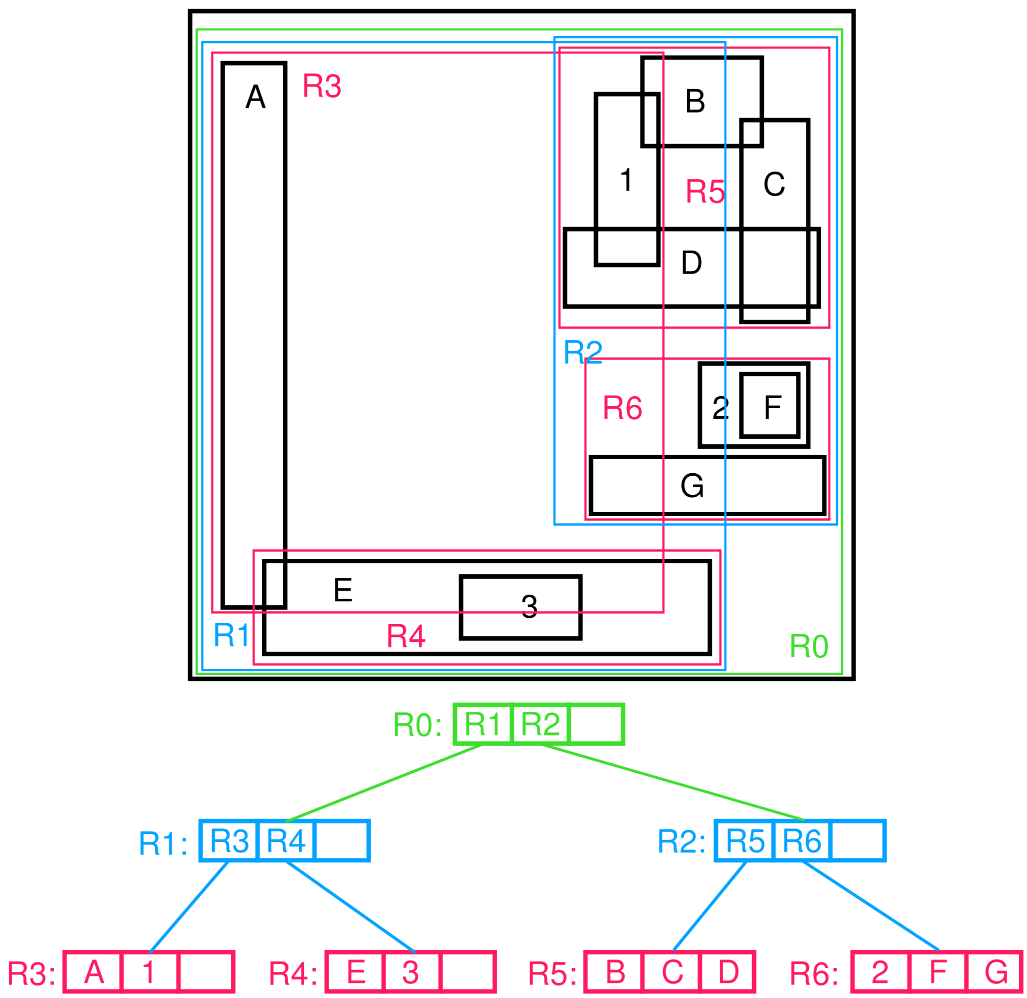
MINIMUM BOUNDING RECTANGLES

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



MINIMUM BOUNDING RECTANGLES

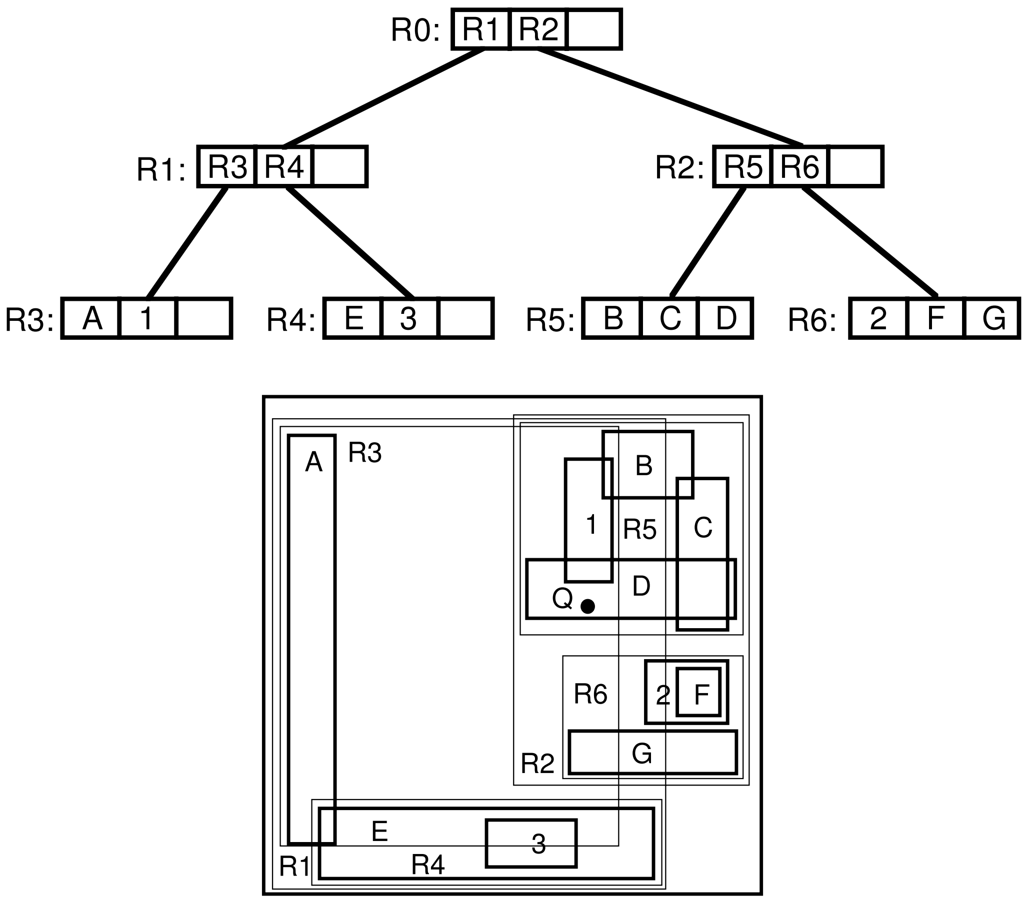
- Rectangles grouped into hierarchies, stored in another structure such as a B-tree
- Drawback: not a disjoint decomposition of space
- Rectangle has single bounding rectangle, yet area it spans may be included in several bounding rectangles
- May have to visit several rectangles to determine the presence/absence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



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

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

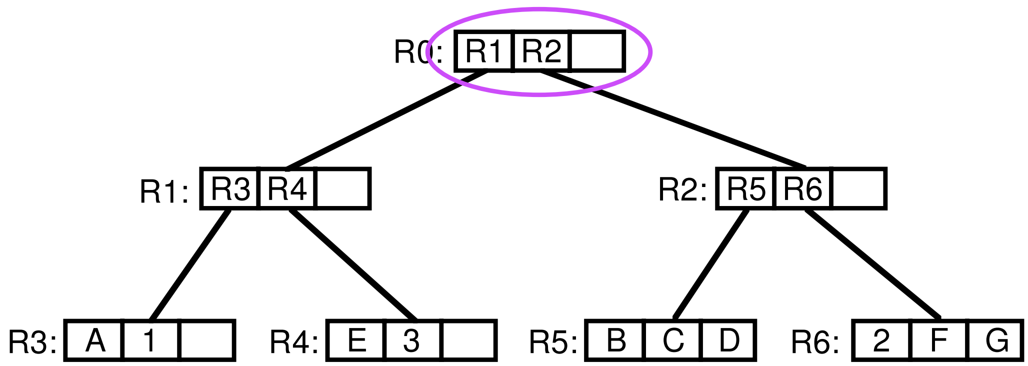
  Ex: Search for the rectangle containing point Q

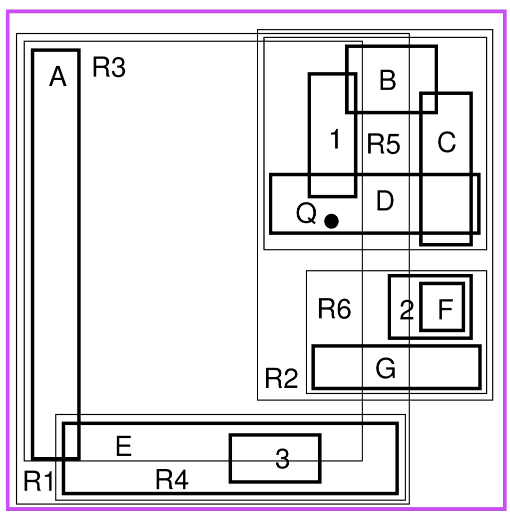


SEARCHING FOR A RECTANGLE CONTAINING A POINT IN AN R-TREE

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

  Ex: Search for the rectangle containing point Q



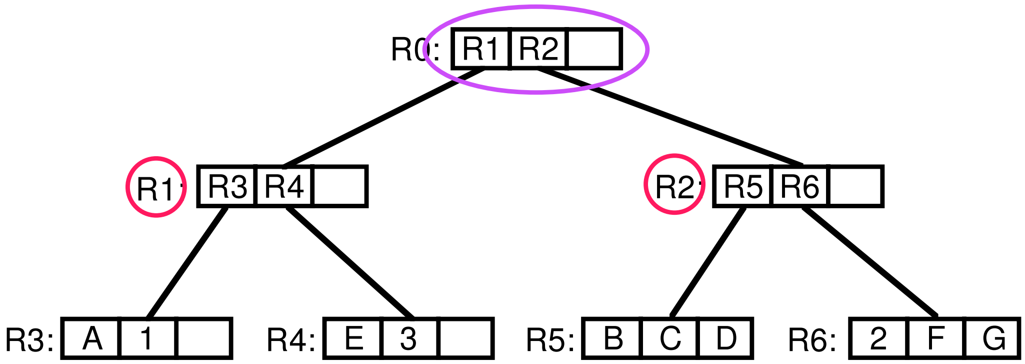


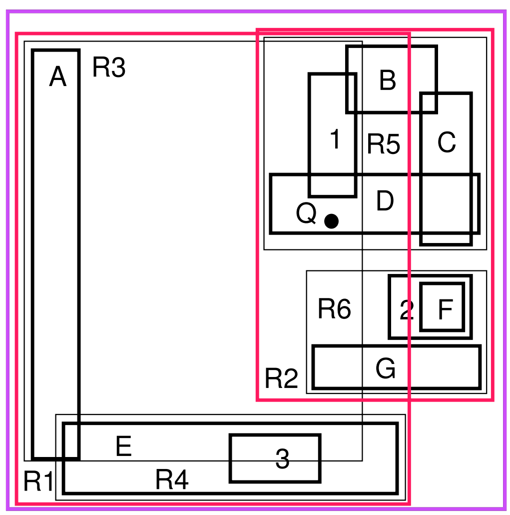
- Q is in R0

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

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

  Ex: Search for the rectangle containing point Q



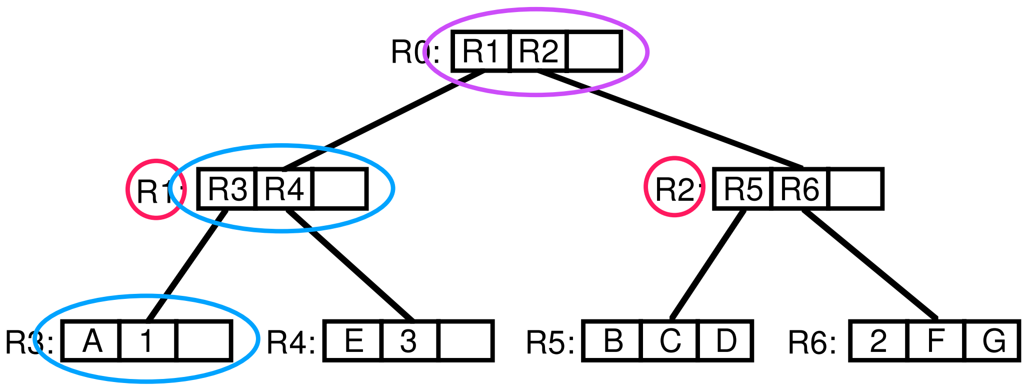


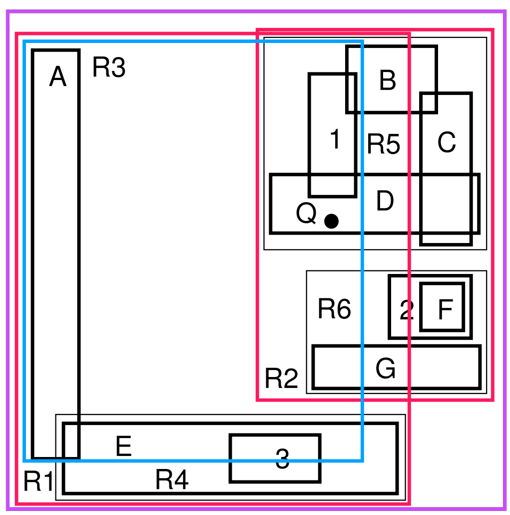
- Q is in R0
- Q can be in both R1 and R2

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

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

  Ex: Search for the rectangle containing point Q



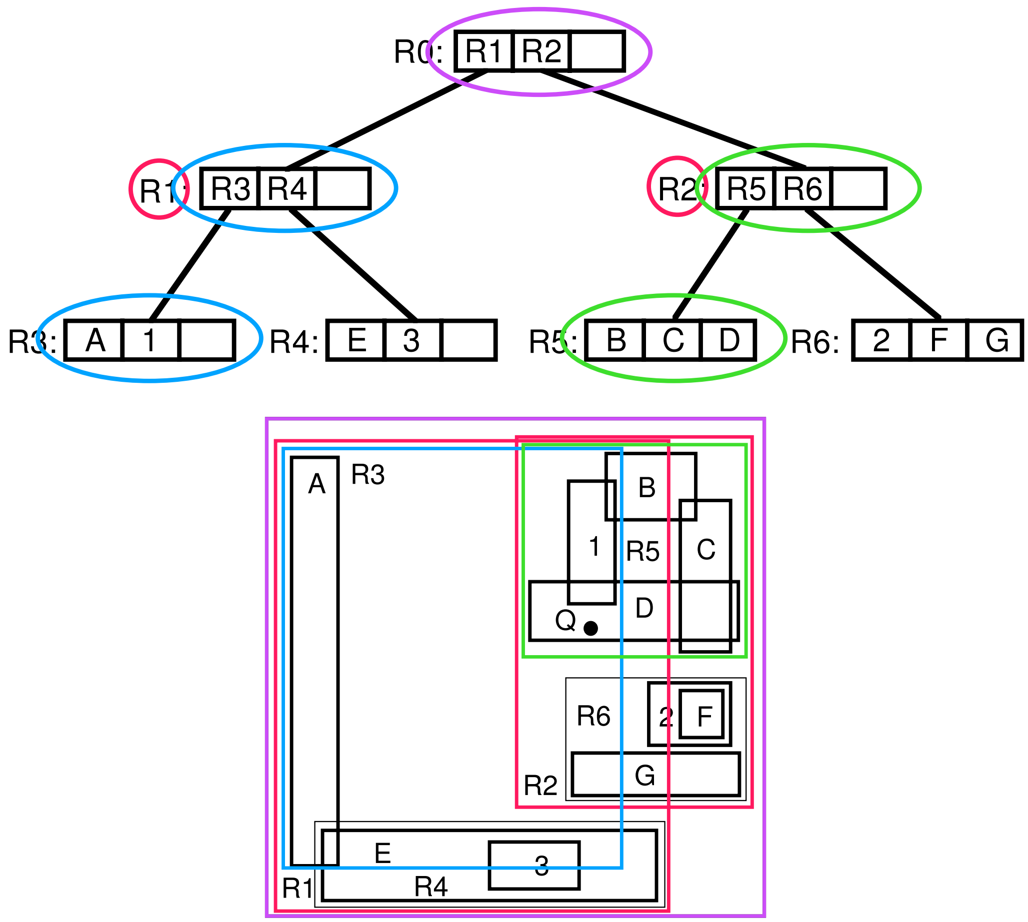


- Q is in R0
- Q can be in both R1 and R2
- Searching R1 first means that R3 is searched but this leads to failure even though Q is in a part of D which is in R3

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

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

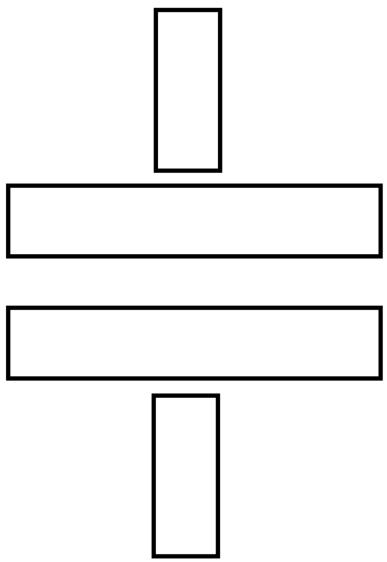
  Ex: Search for the rectangle containing point Q



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

Dynamic R-Tree Construction

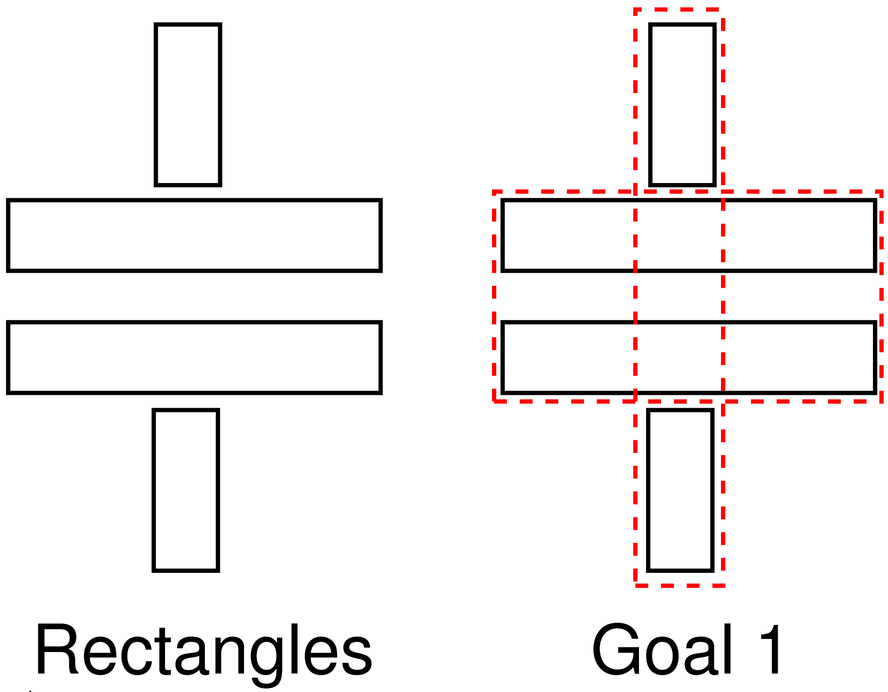
- Differ by how to split overflowing node p upon insertion
- Conflicting goals



Rectangles

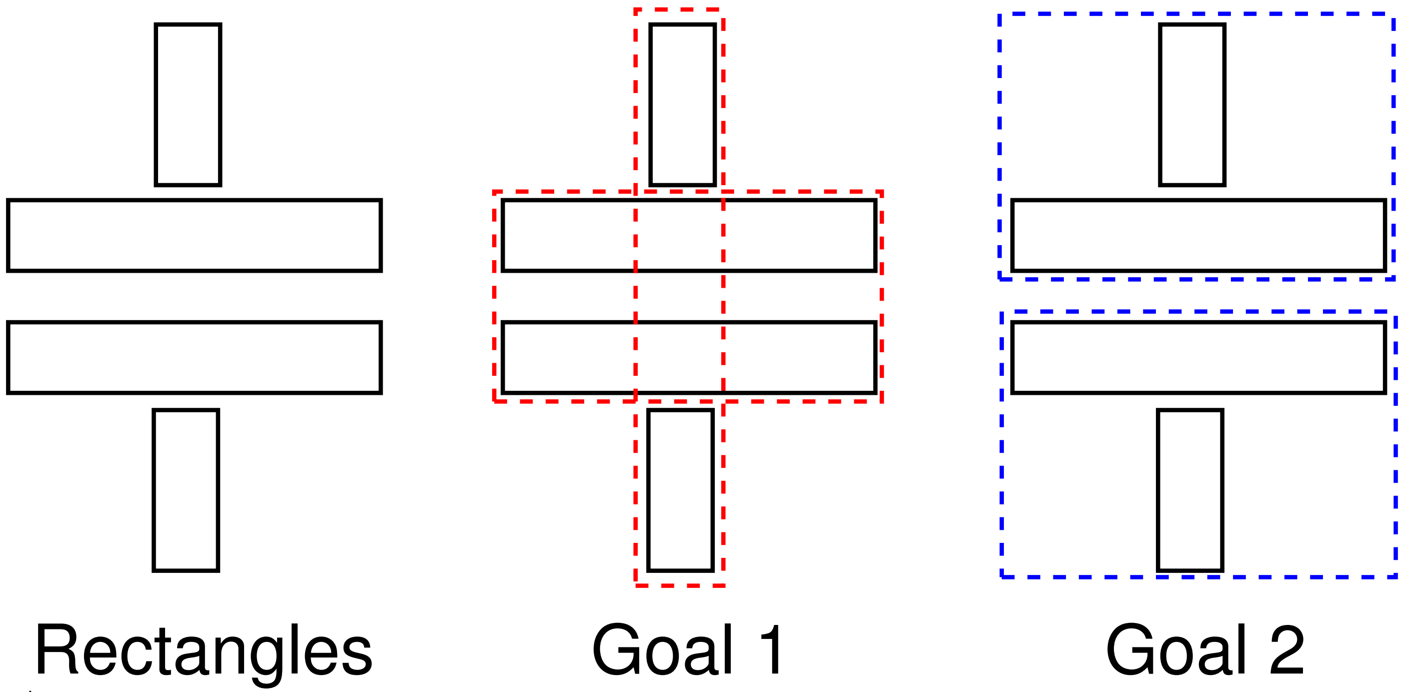
Dynamic R-Tree Construction

- Differ by how to split overflowing node p upon insertion
- Conflicting goals
  1. Reduce likelihood that each node q is visited by the search
     - achieve by minimizing total area spanned by bounding box of q (coverage)



Dynamic R-Tree Construction

- Differ by how to split overflowing node p upon insertion
- Conflicting goals
  1. Reduce likelihood that each node q is visited by the search
     - achieve by minimizing total area spanned by bounding box of q (coverage)
  2. minimize number of children of p that must be visited by search operations
     - achieve by minimizing area common to children so that the area that they span is not visited a multiple number of times (overlap)



EXAMPLE DYNAMIC SPLITTING METHODS

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

R-TREE OVERFLOW NODE SPLITTING POLICIES

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

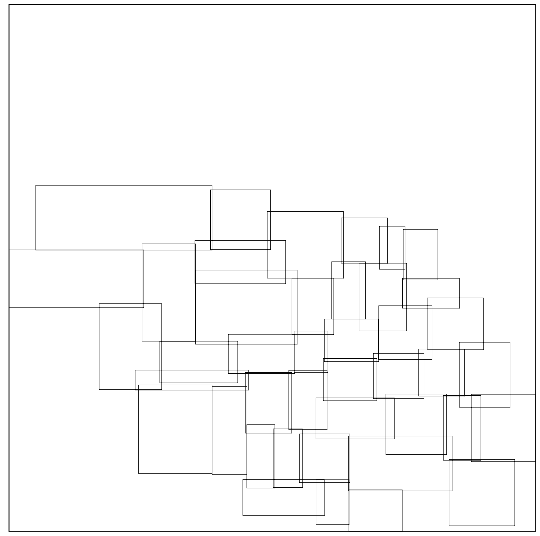
R*-TREE

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

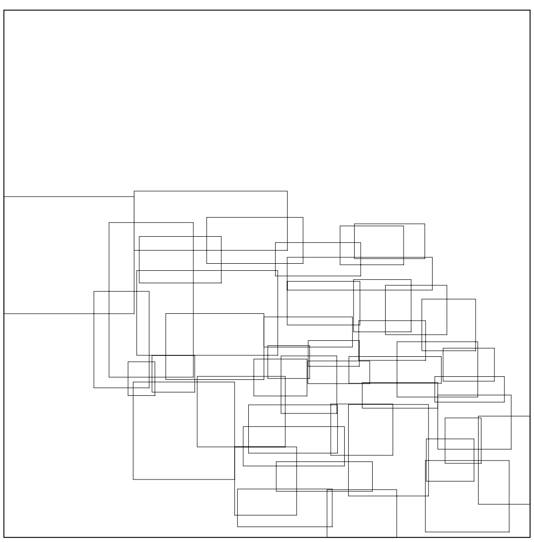
EXAMPLE OF R-TREE NODE SPLITTING POLICIES

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

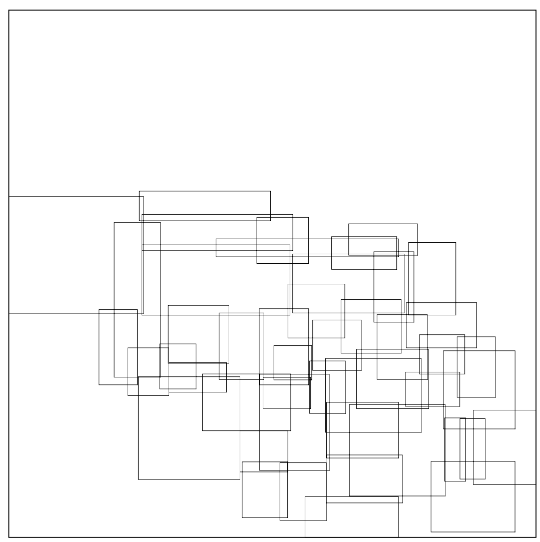
Collection of lines



R*-tree



Linear



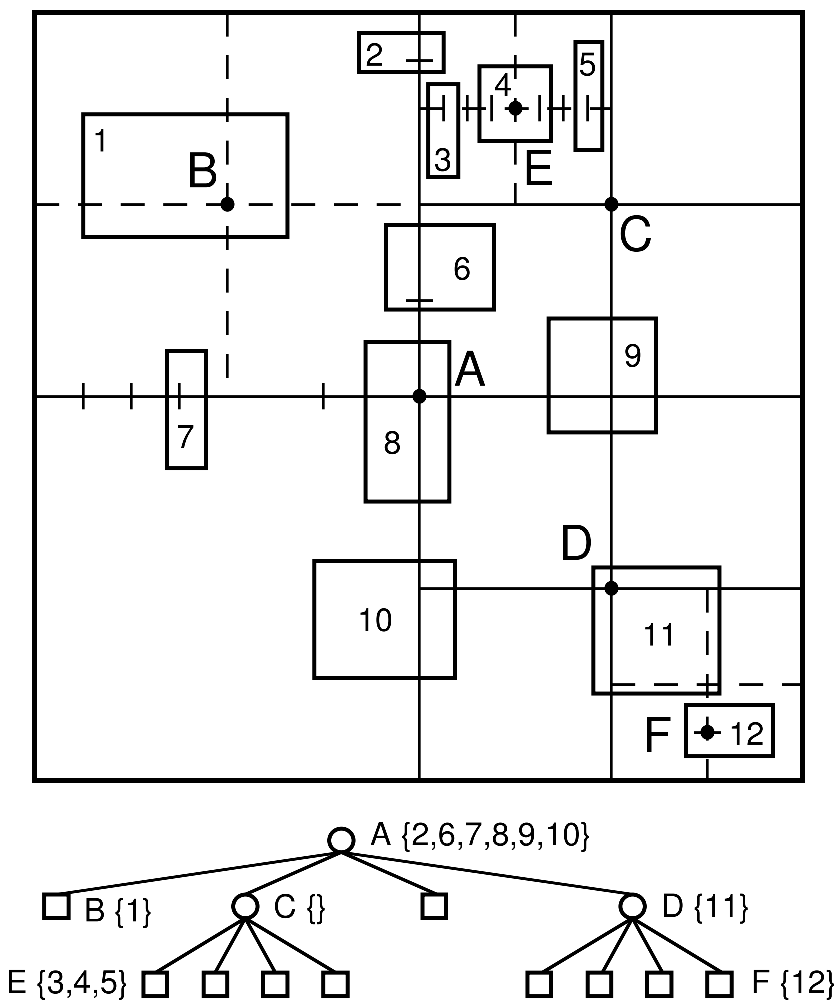
Quadratic

Outline

1. Introduction
2. Points
3. Lines
4. Regions, Volumes, and Surfaces
5. Bounding Box Hierarchies
6. Rectangles
7. Surfaces and Volumes
8. Metric Data
9. Operations
10. Indexing Spatiotextual Data
11. Example system

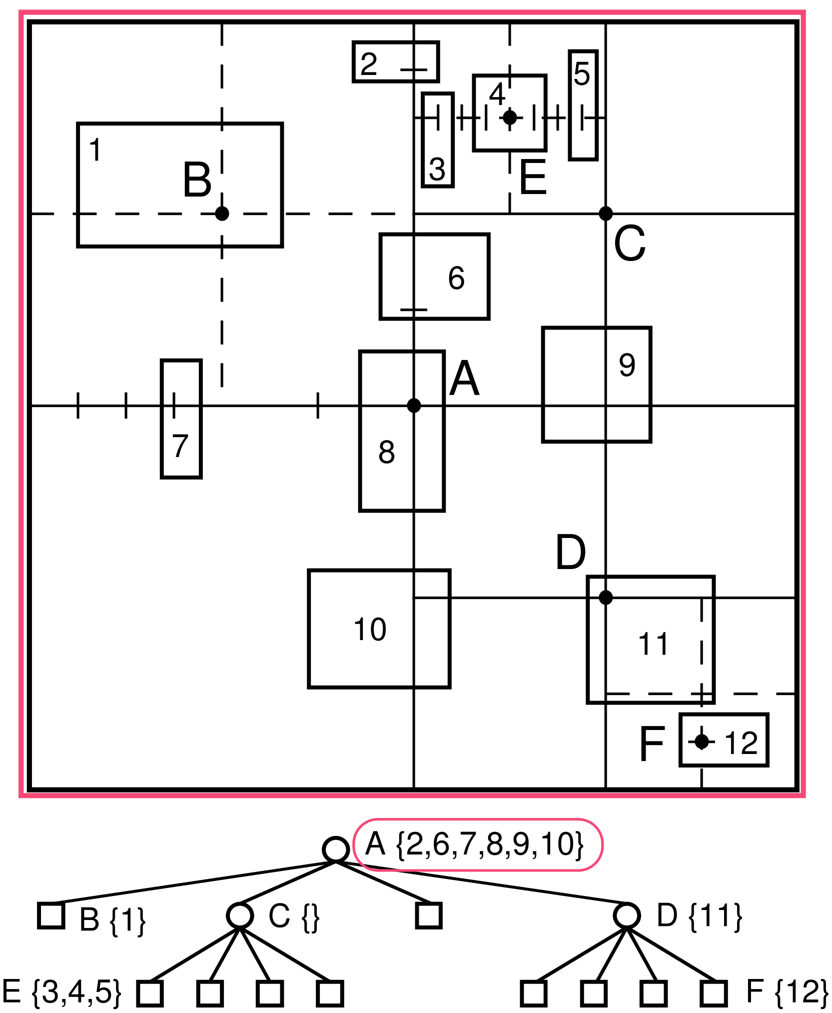
MX-CIF QUADTREE (Kedem)

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



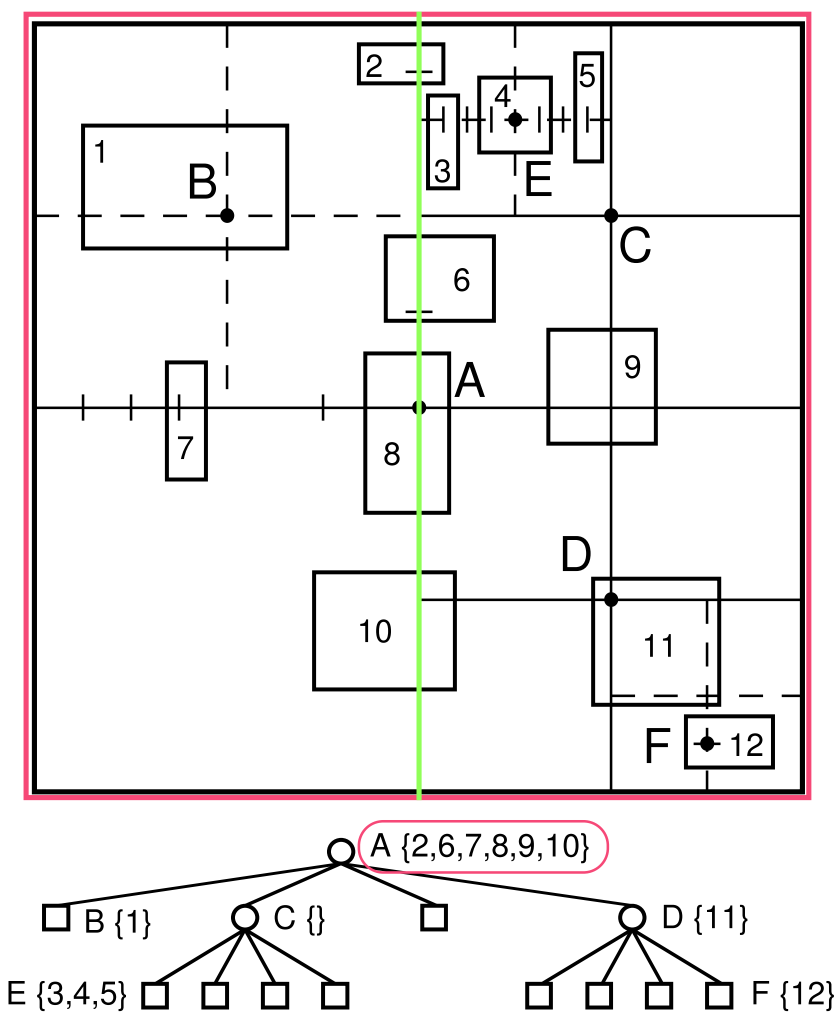
MX-CIF QUADTREE (Kedem)

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

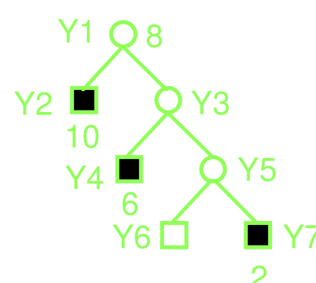


MX-CIF QUADTREE (Kedem)

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

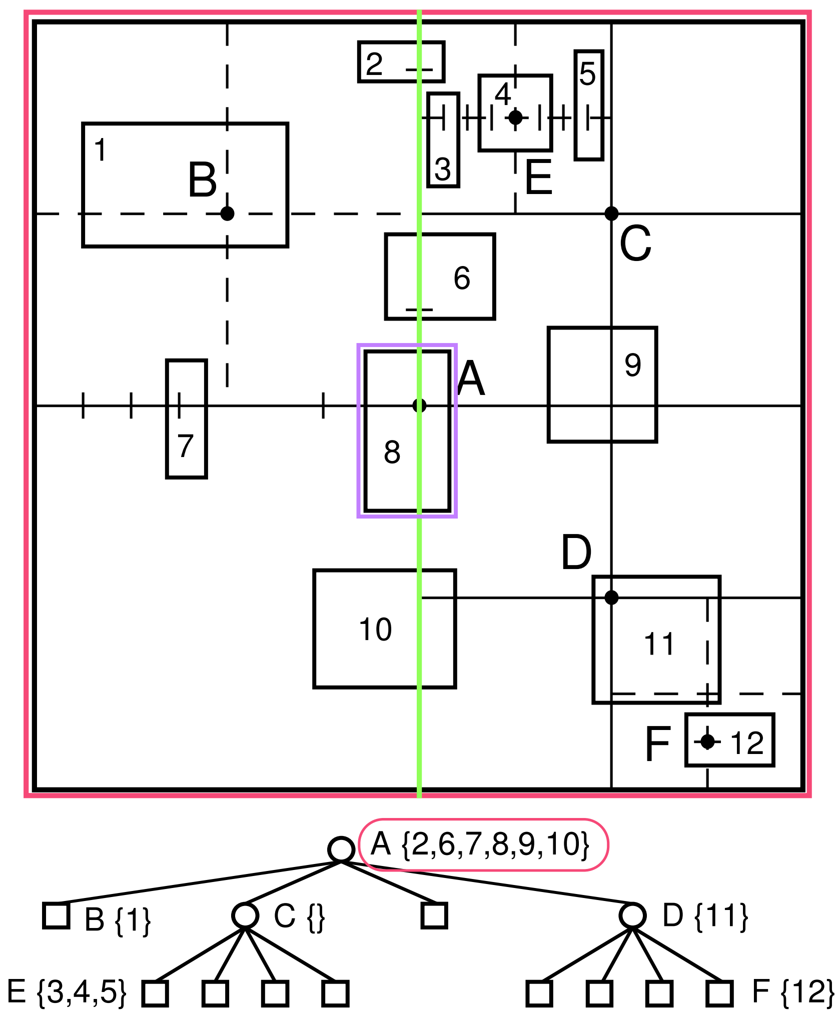


Binary tree for *y*-axis through A

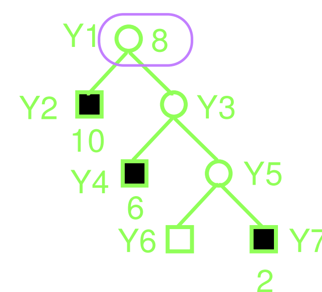


MX-CIF QUADTREE (Kedem)

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

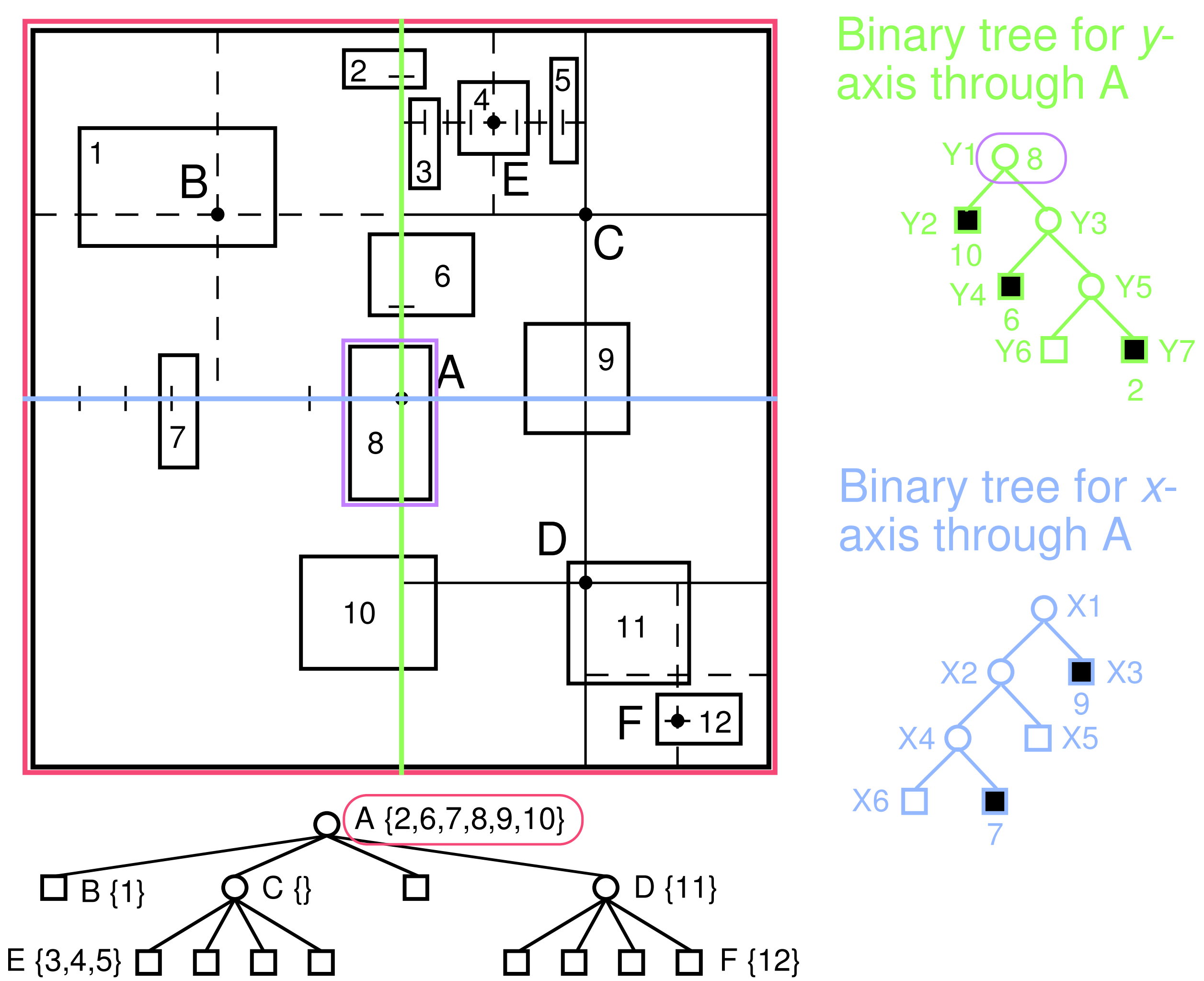


Binary tree for *y*-axis through A



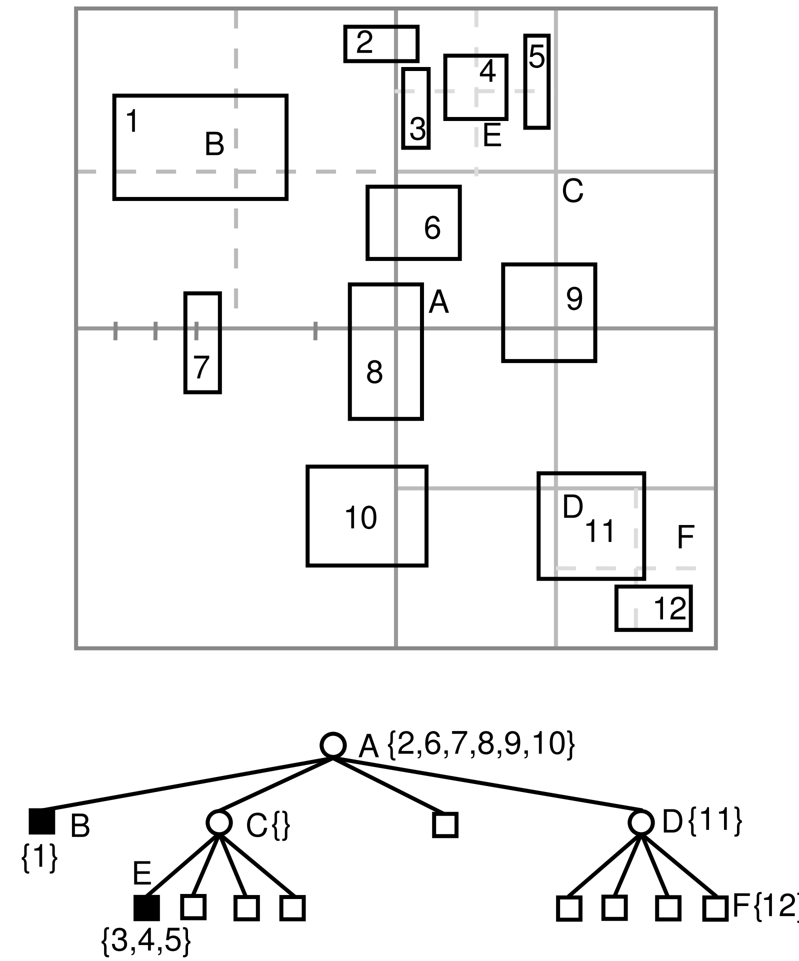
MX-CIF QUADTREE (Kedem)

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



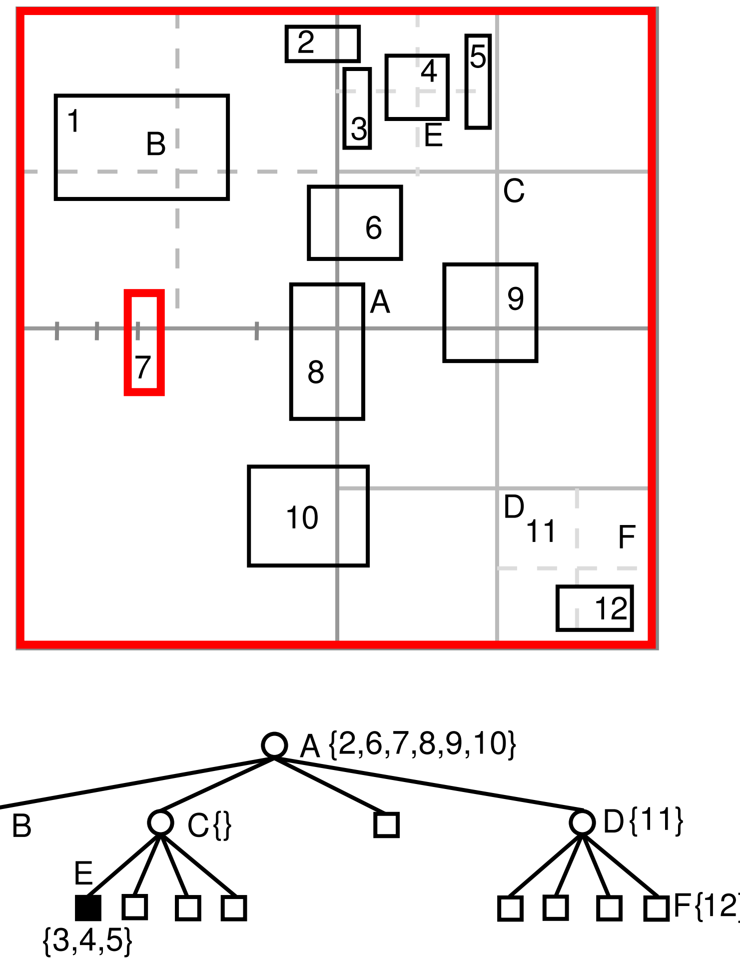
Loose Quadtree (Octree)/Cover Fieldtree

- 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



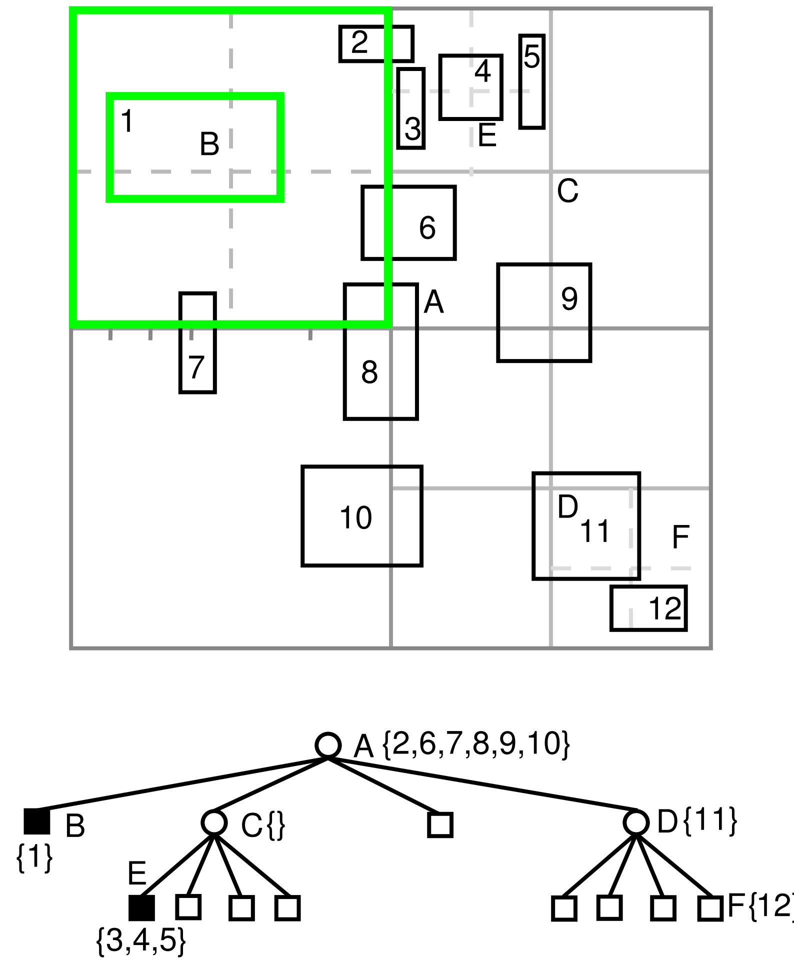
Loose Quadtree (Octree)/Cover Fieldtree

- 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



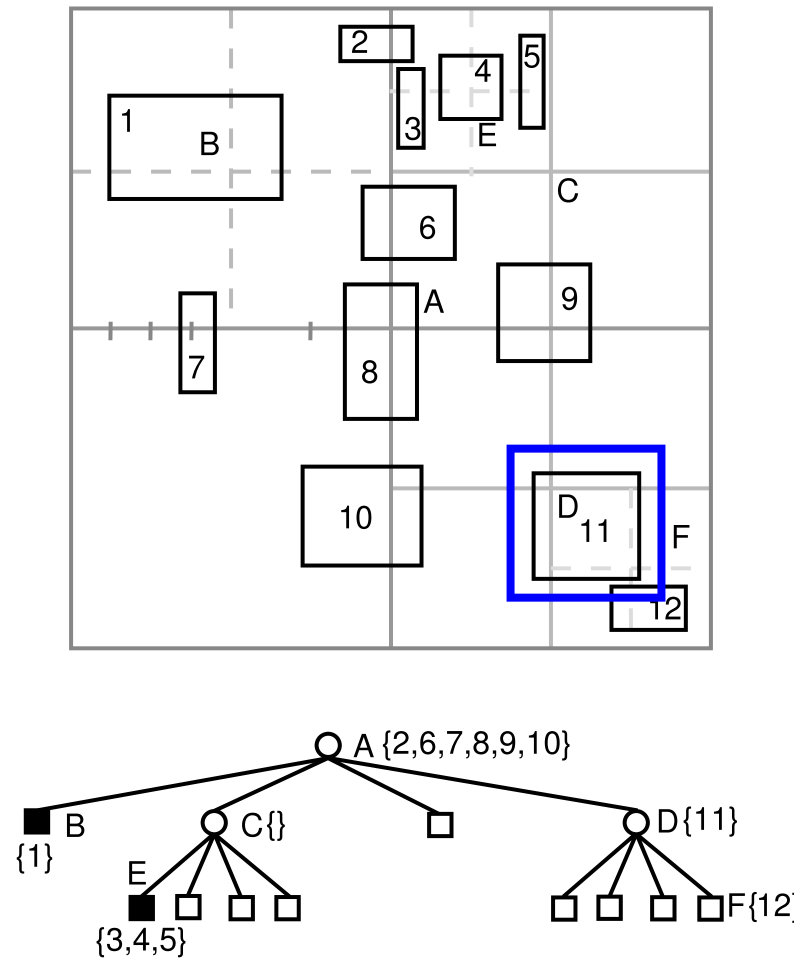
Loose Quadtree (Octree)/Cover Fieldtree

- 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

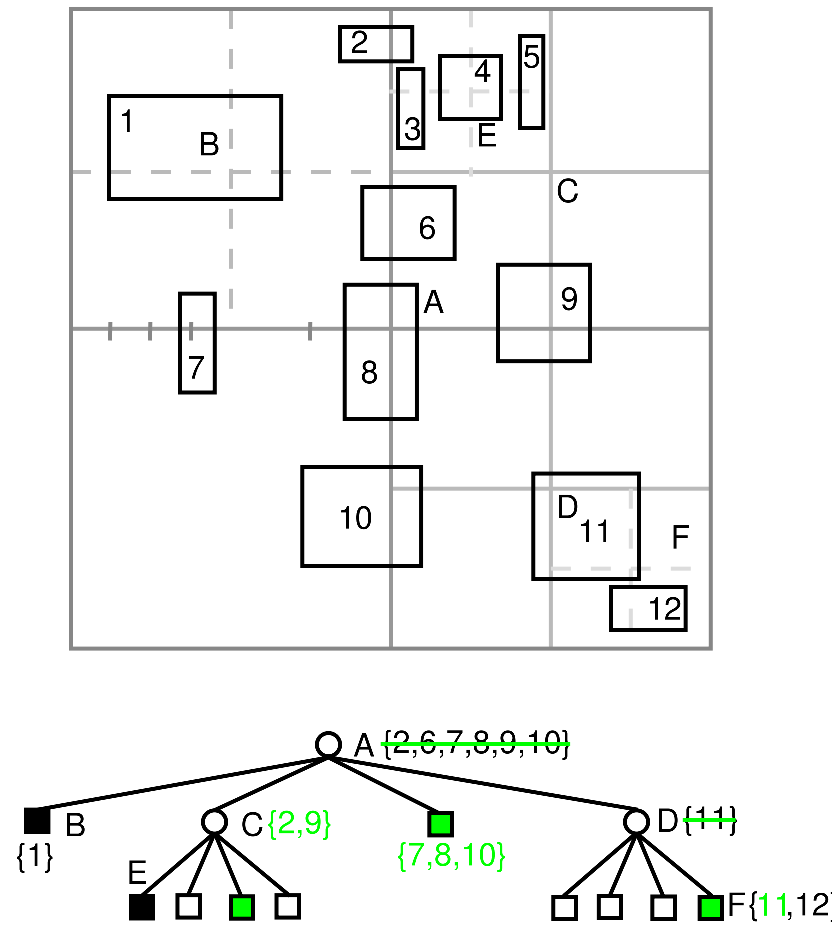
- Overcomes drawback of MX-CIF quadtree that the width w of the minimum enclosing quadtree block of a rectangle o is not a function of the size of o
- Instead, it depends on the position of the centroid of o and often considerably larger than o
- Solution: expand size of space spanned by each quadtree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

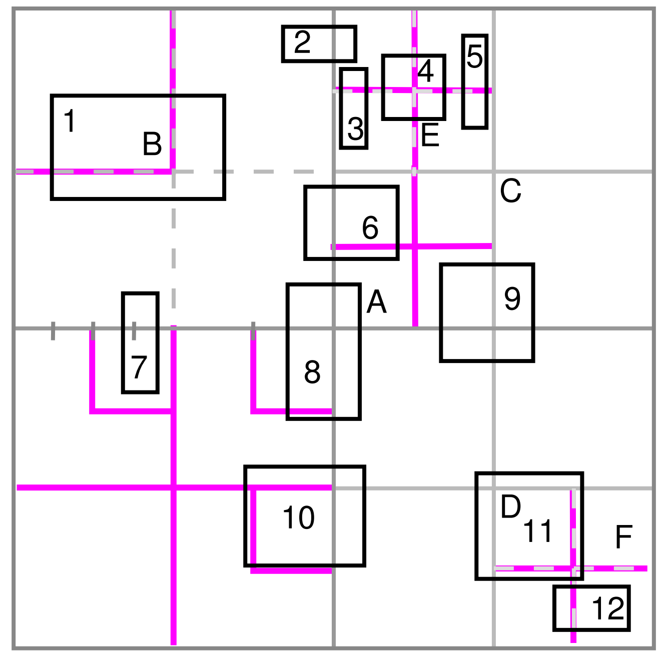
  1. $p = 0.3$

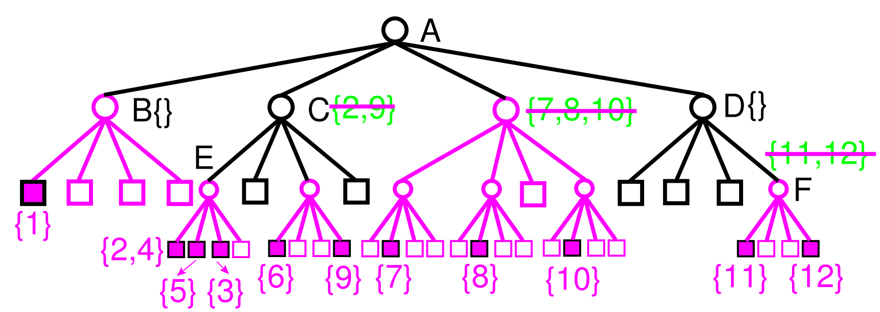


Loose Quadtree (Octree)/Cover Fieldtree

- 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 $(1+p)w$

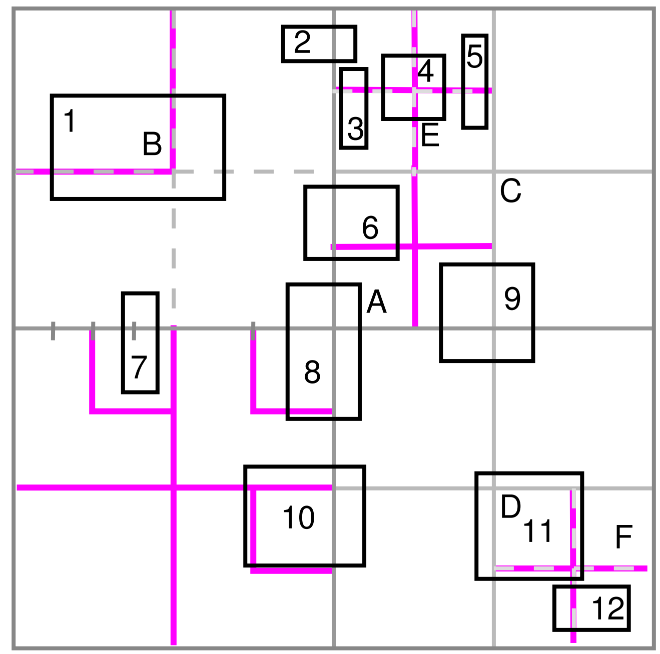
  1. $p = 0.3$
  2. $p = 1.0$

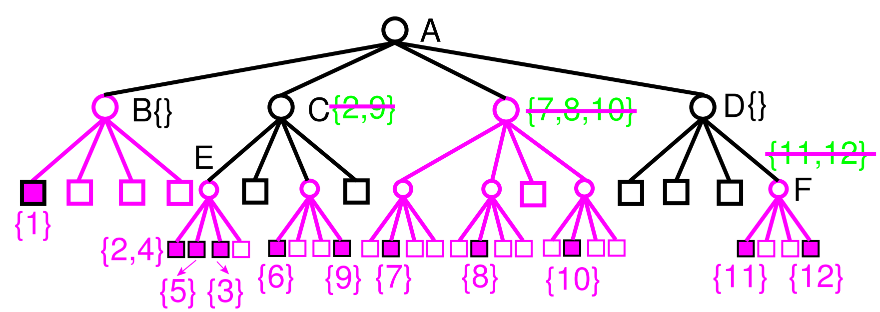




Loose Quadtree (Octree)/Cover Fieldtree

- 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

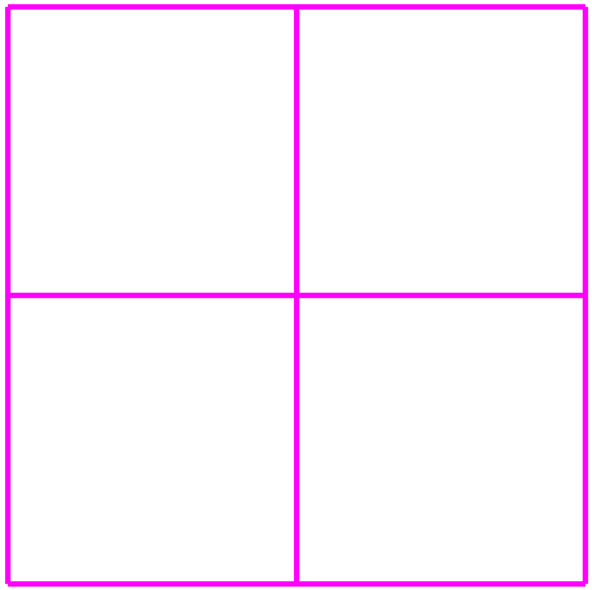
- Alternative to loose quadtree (octree)/cover fieldtree at overcoming drawback of MX-CIF quadtree that the width w of the minimum enclosing quadtree block of a rectangle o is not a function of the size of o

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

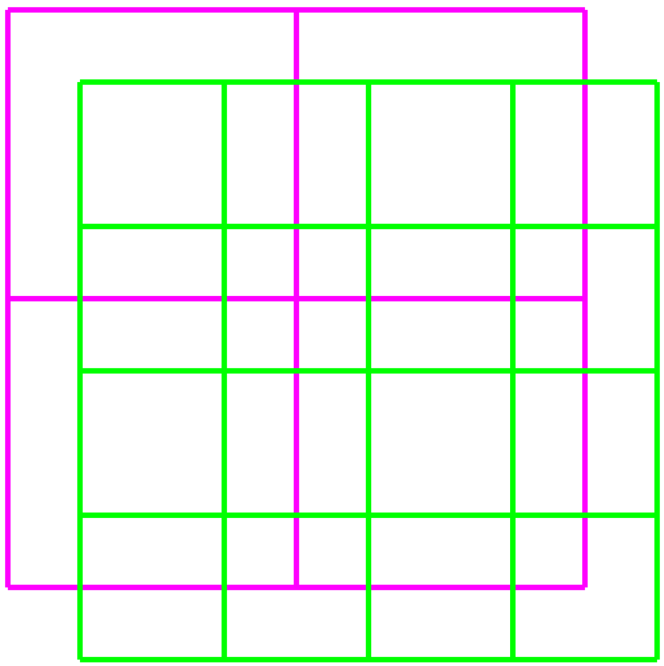
Partition Fieldtree

- 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



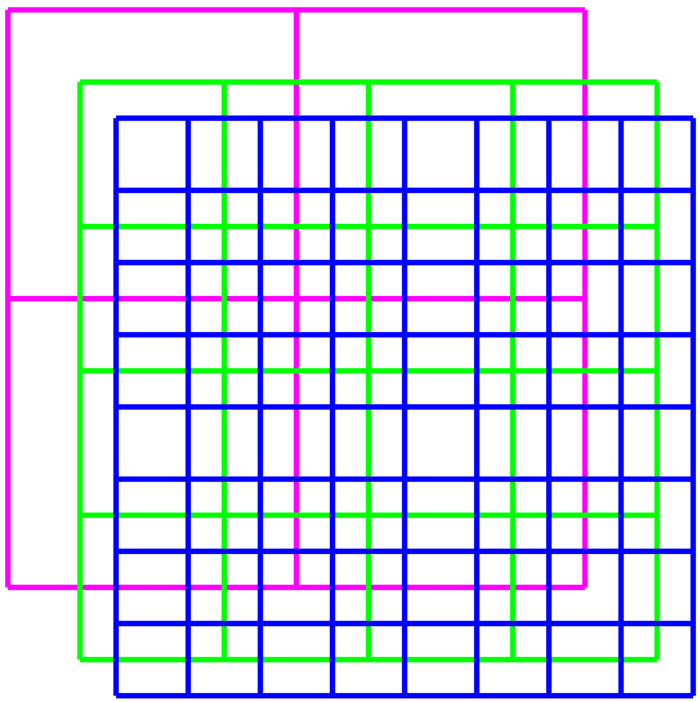
Partition Fieldtree

- 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



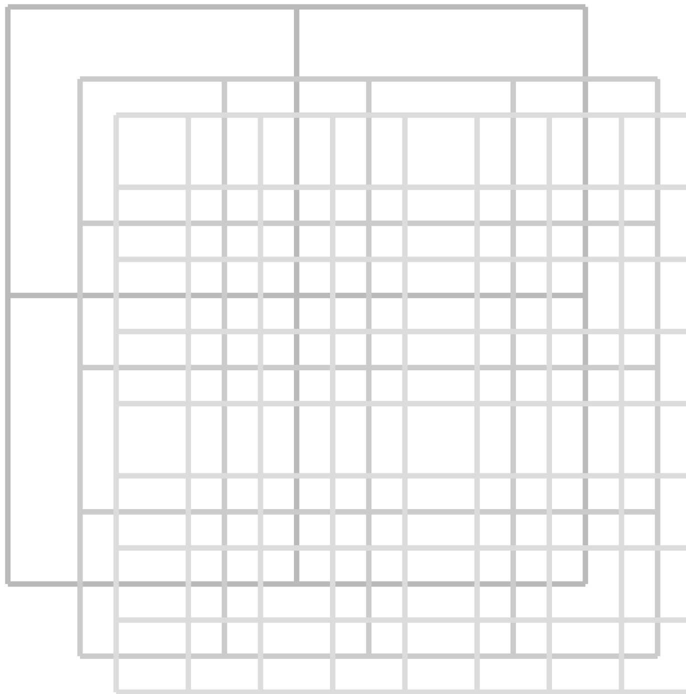
Partition Fieldtree

- 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



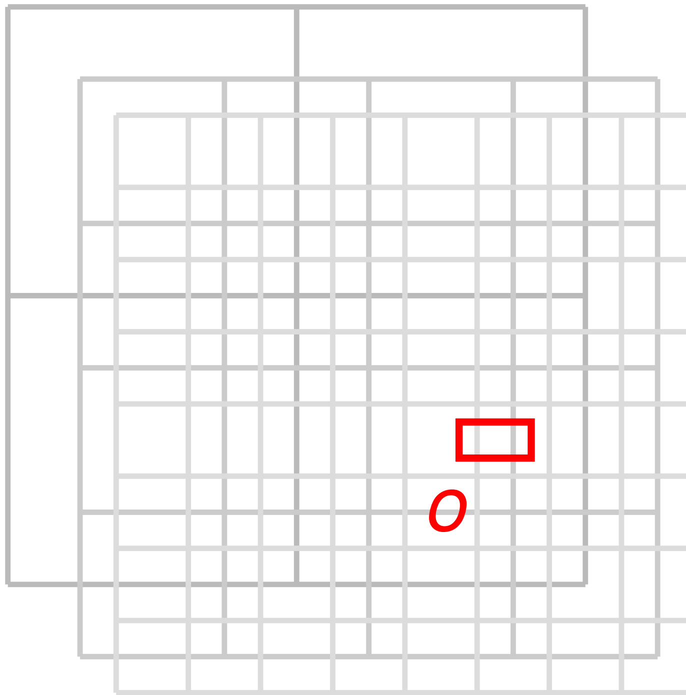
Partition Fieldtree

- 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



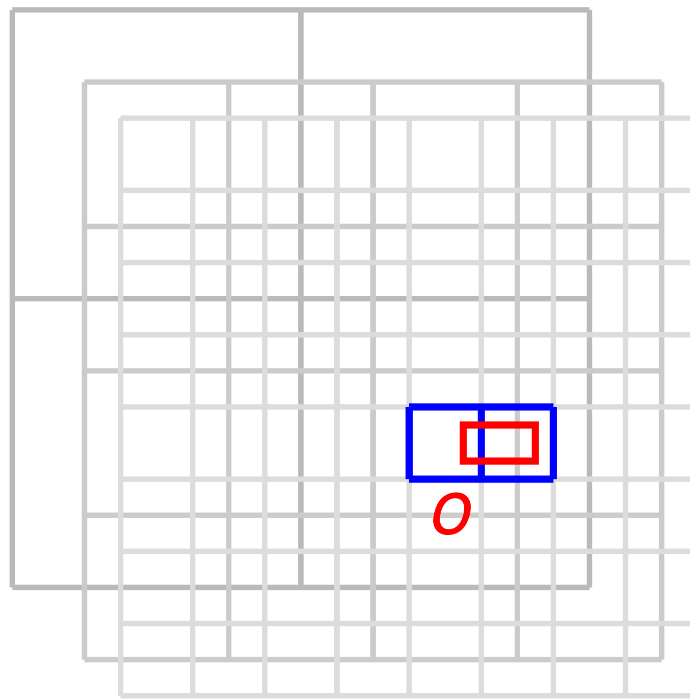
Partition Fieldtree

- 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



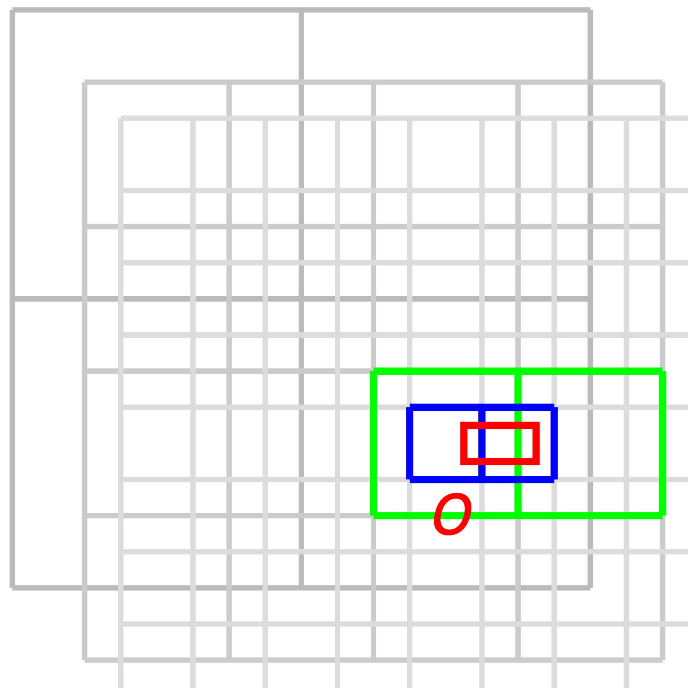
Partition Fieldtree

- 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



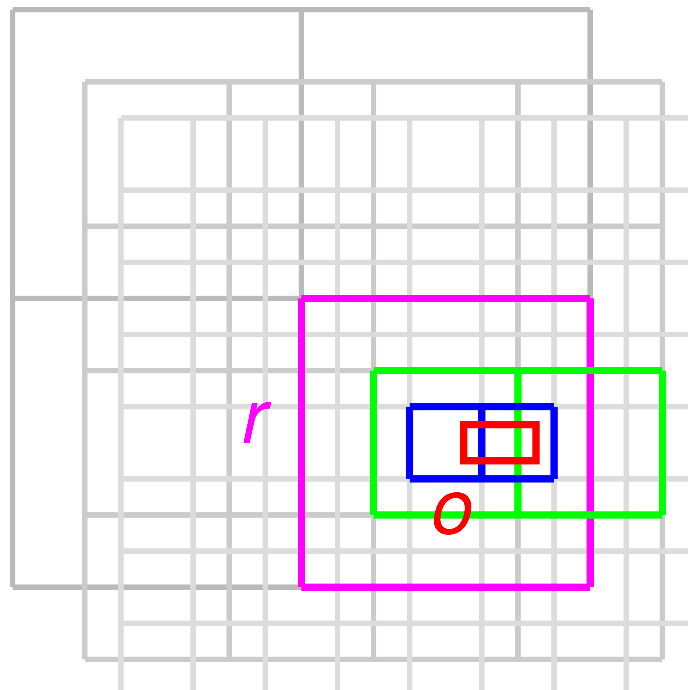
Partition Fieldtree

- 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



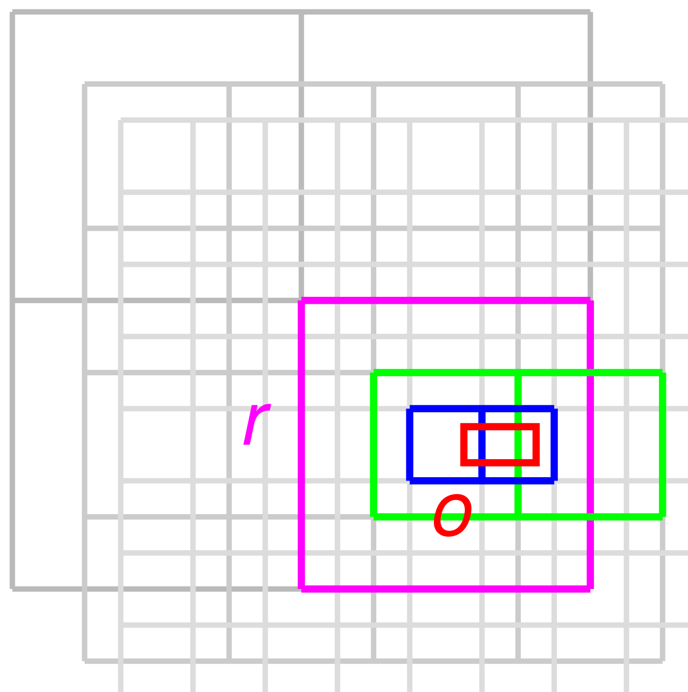
Partition Fieldtree

- 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



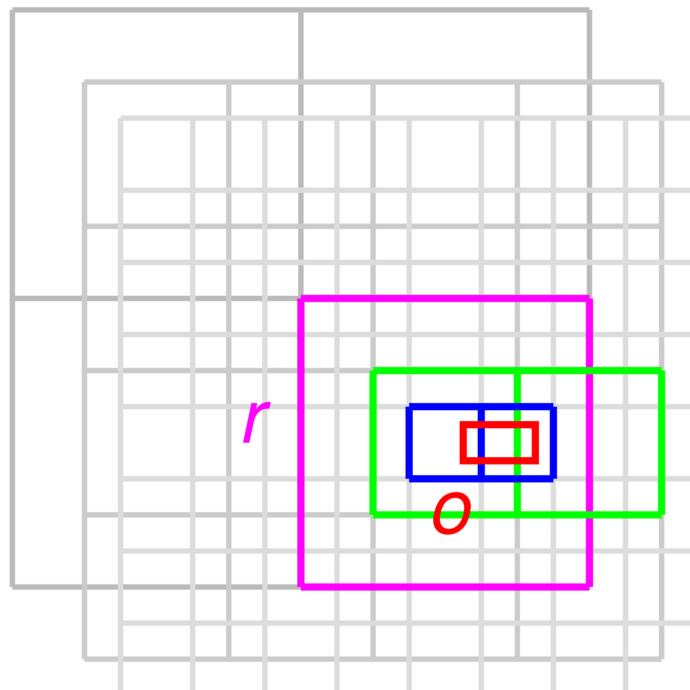
Partition Fieldtree

- 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

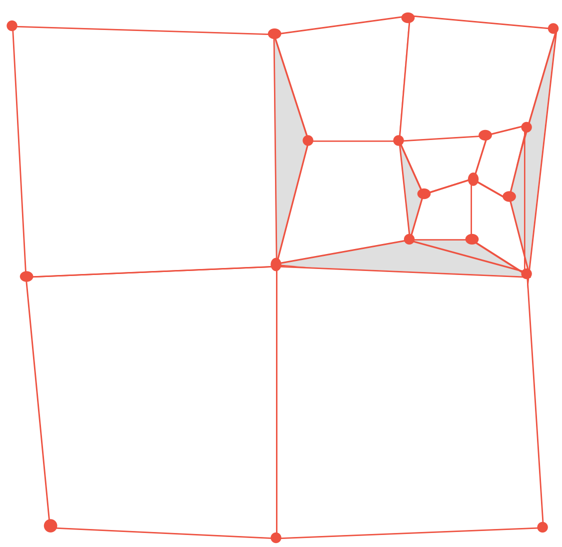


Outline

1. Introduction
2. Points
3. Lines
4. Regions, Volumes, and Surfaces
5. Bounding Box Hierarchies
6. Rectangles
7. Surfaces and Volumes
8. Metric Data
9. Operations
10. Indexing Spatiotextual Data
11. Example system

HIERARCHICAL RECTANGULAR DECOMPOSITION

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

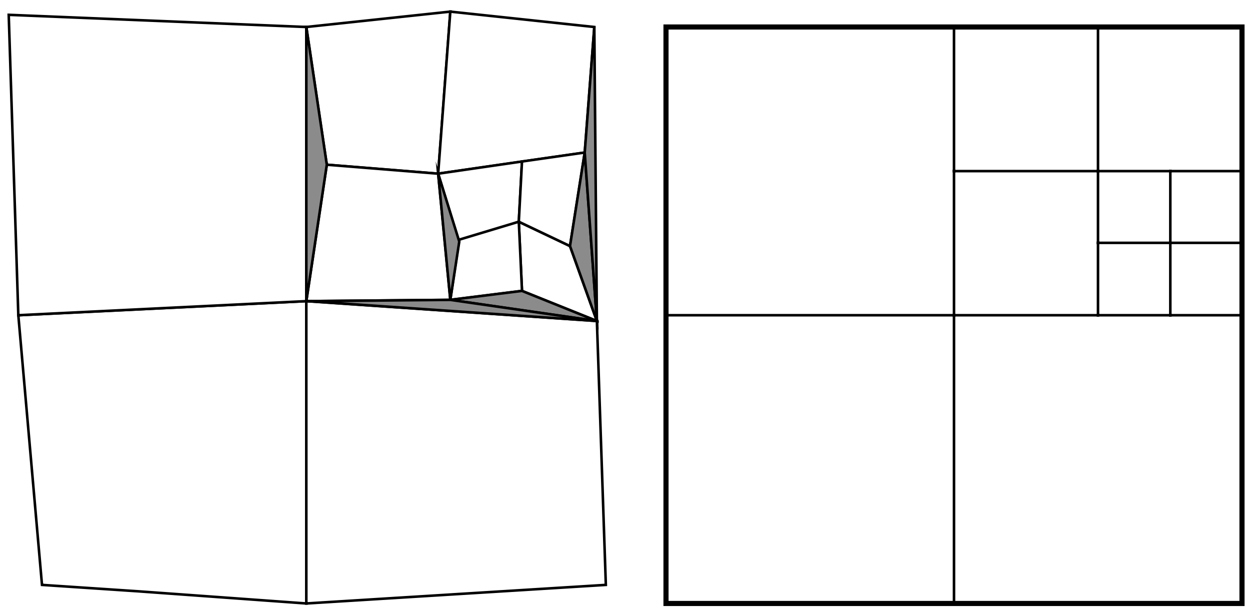


- Overcoming the presence of cracks
  1. use the interpolated point instead of the true point (Barrera and Hinjosa)
  2. triangulate the squares (Von Herzen and Barr)
     - can split into 2, 4, or 8 triangles depending on how many lines are drawn through the midpoint
     - if split into 2 triangles,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

RESTRICTED QUADTREE (VON HERZEN/BARR)

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

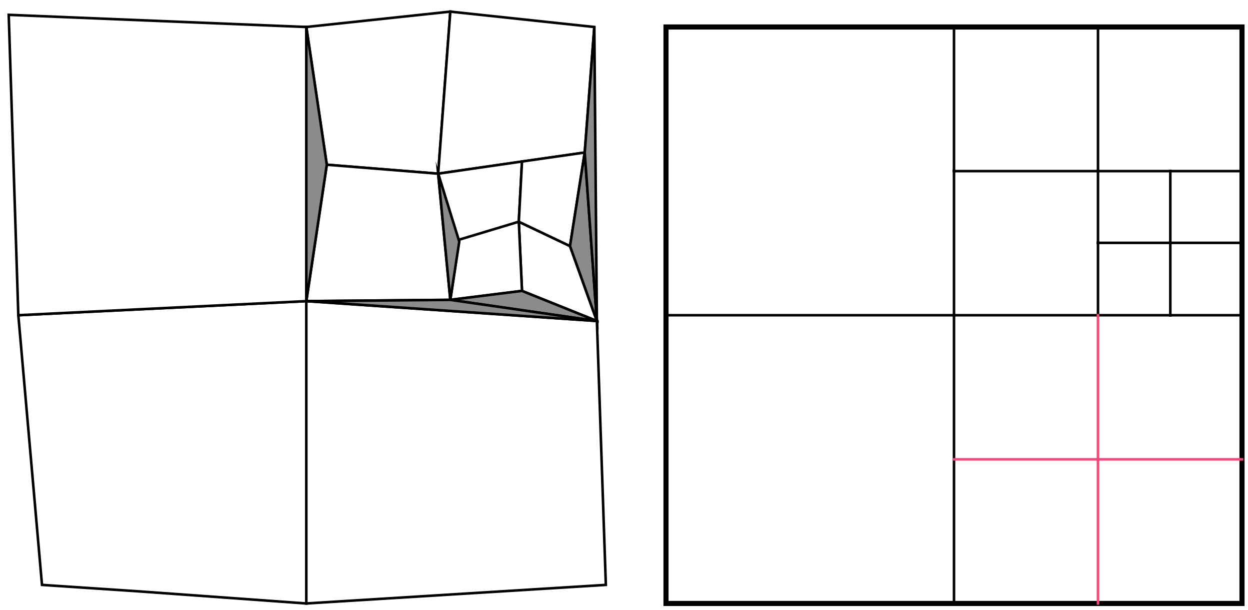
  Note: also used in finite element analysis to adptively refine an element as well as to achieve element compatibility (termed *h-refinement* by Kela, Perucchio, and Voelcker)



RESTRICTED QUADTREE (VON HERZEN/BARR)

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

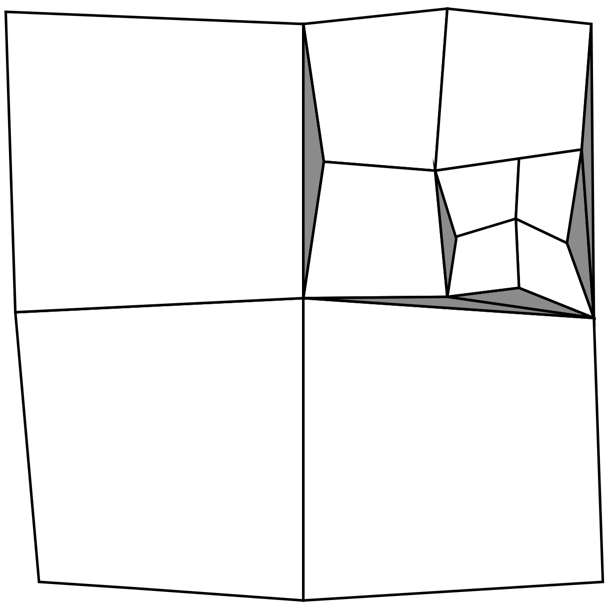
  Note: also used in finite element analysis to adptively refine an element as well as to achieve element compatibility (termed *h-refinement* by Kela, Perucchio, and Voelcker)

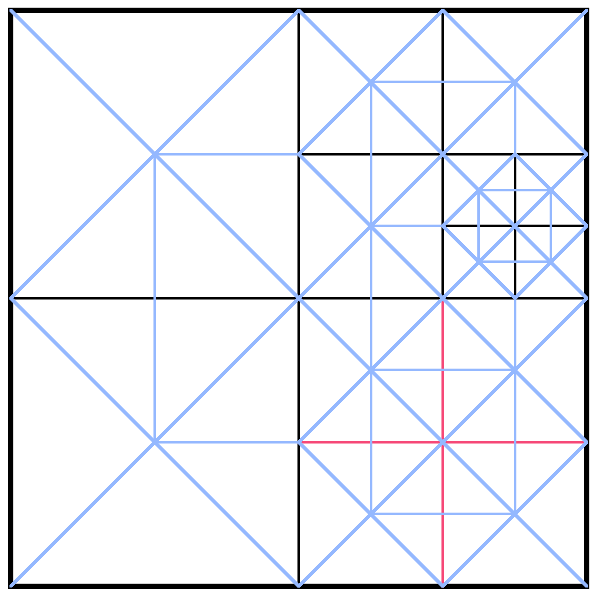


RESTRICTED QUADTREE (VON HERZEN/BARR)

- 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)



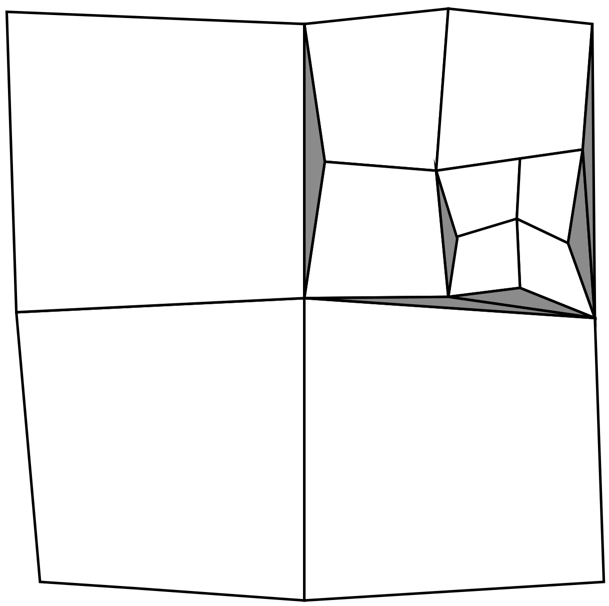
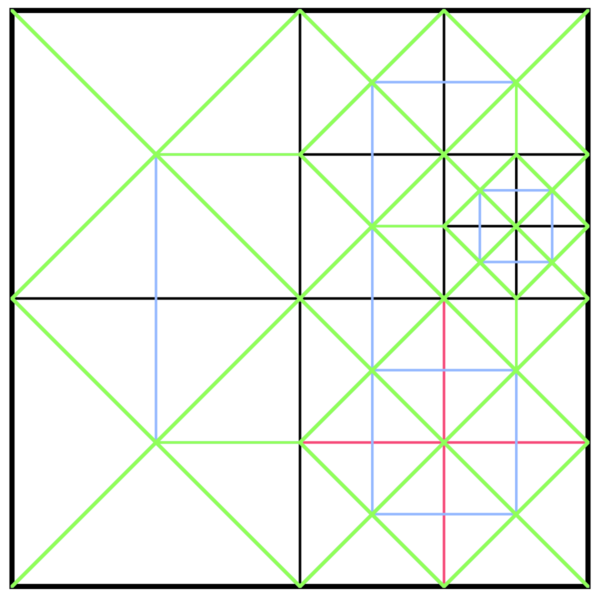


- 8-triangle decomposition rule
  1. decompose each block into 8 triangles (i.e., 2 triangles per edge)
  2. unless the edge is shared by a larger block
  3. in which case only 1 triangle is formed

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)

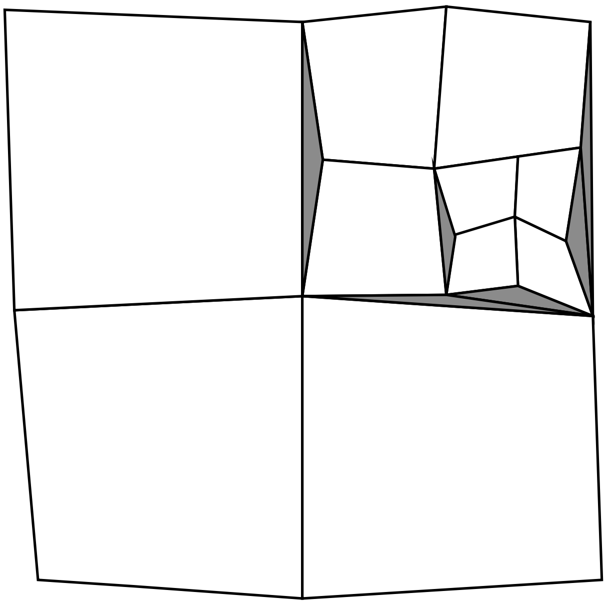



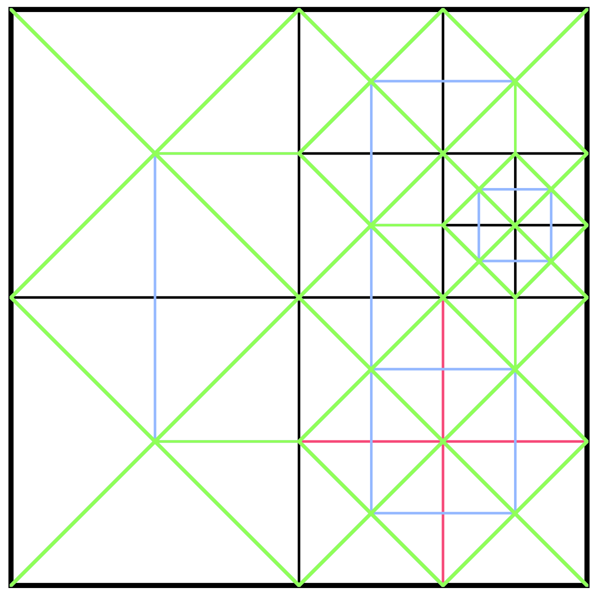
- 8-triangle decomposition rule
  1. decompose each block into 8 triangles (i.e., 2 triangles per edge)
  2. unless the edge is shared by a larger block
  3. in which case only 1 triangle is formed
- 4-triangle decomposition rule
  1. decompose each block into 4 triangles (i.e., 1 triangle per edge)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



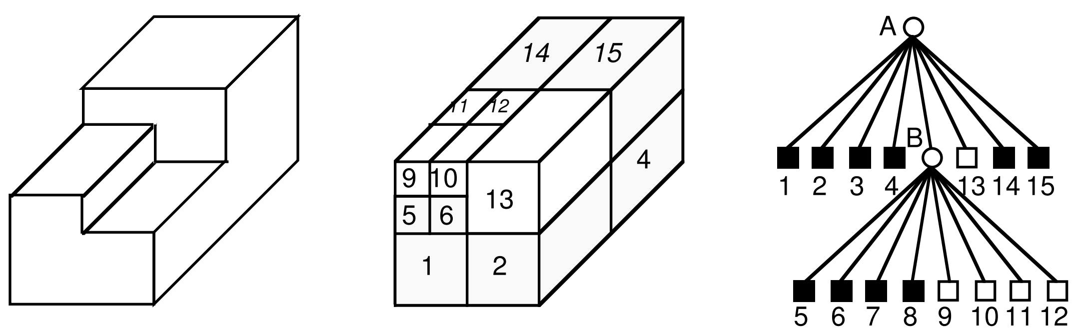


- 8-triangle decomposition rule
  1. decompose each block into 8 triangles (i.e., 2 triangles per edge)
  2. unless the edge is shared by a larger block
  3. in which case only 1 triangle is formed
- 4-triangle decomposition rule
  1. decompose each block into 4 triangles (i.e., 1 triangle per edge)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

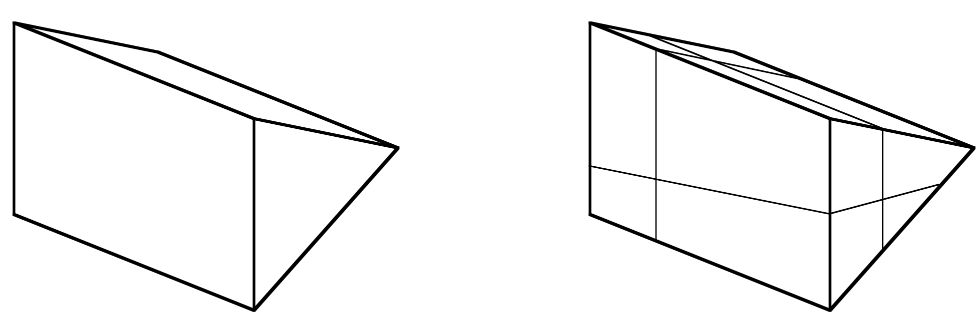
OCTREES

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 (PM octrees)
   - 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 spatial index on a boundary model (BRep)



Outline

Basic Definitions

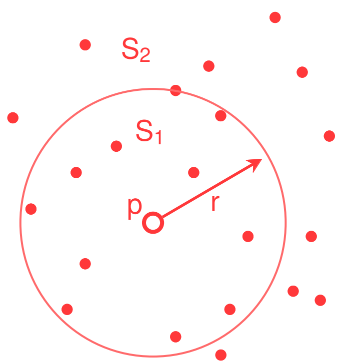
1. Often only information available is a distance function indicating degree of similarity (or dis-similarity) between all pairs of N data objects
2. Distance metric d: objects must reside in finite metric space (S, d) where for o_1, o_2, o_3 in S, d must satisfy
   - $d(o_1, o_2) = d(o_2, o_1)$ (symmetry)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

Pivots

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

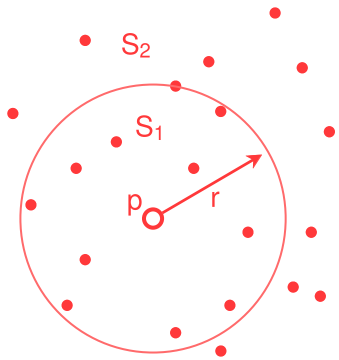
Pivots

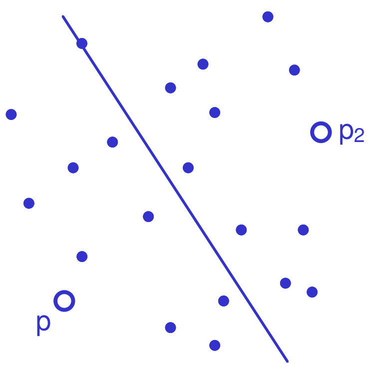
- Identify a distinguished object or subset of the objects termed *pivots* or *vantage points*
  1. sort remaining objects based on
     a. distances from the pivots, or
     b. which pivot is the closest
  2. and build index
  3. use index to achieve pruning of other objects during search
- Given pivot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



Pivots

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



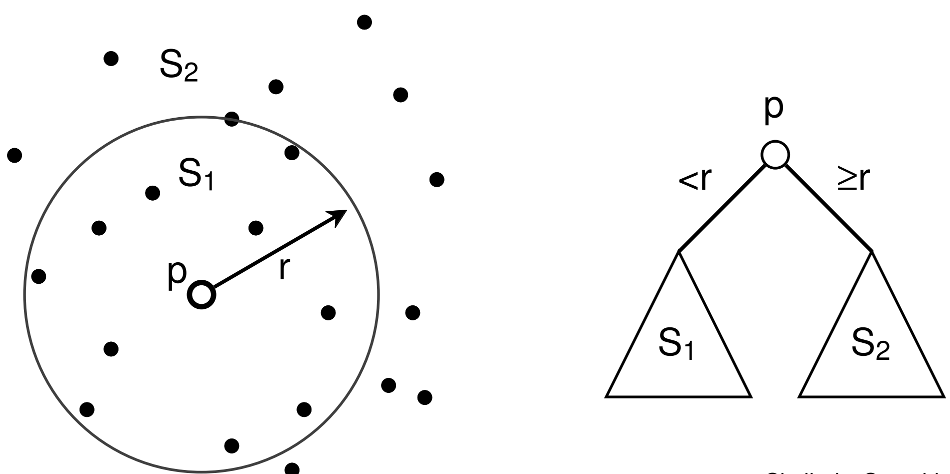


vp-Tree (Metric Tree; Uhlmann|Yianilos)

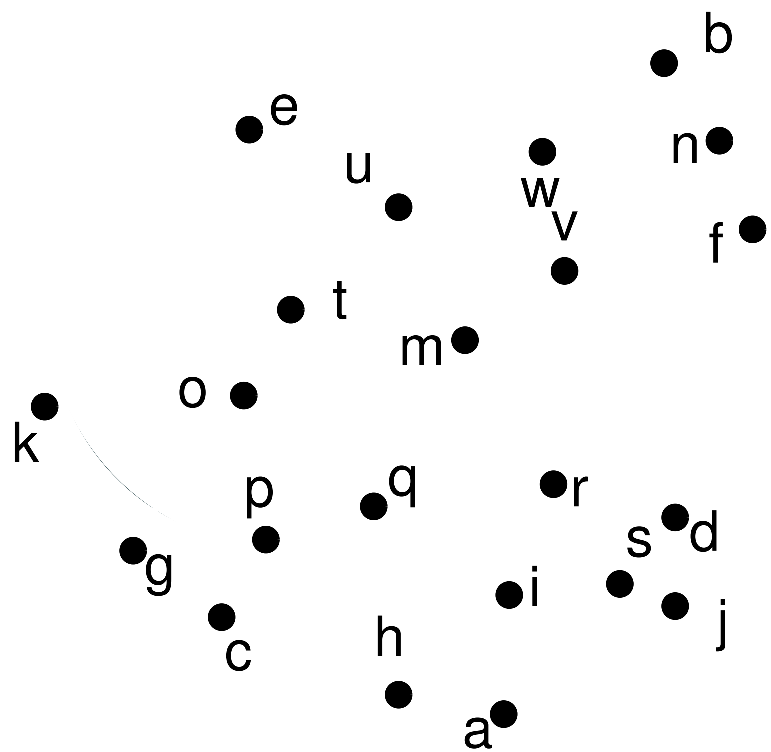
- Ball partitioning method
- Pick p from S and let r be median of distances of other objects from p
- Partition S into two sets S_1 and S_2 where:

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

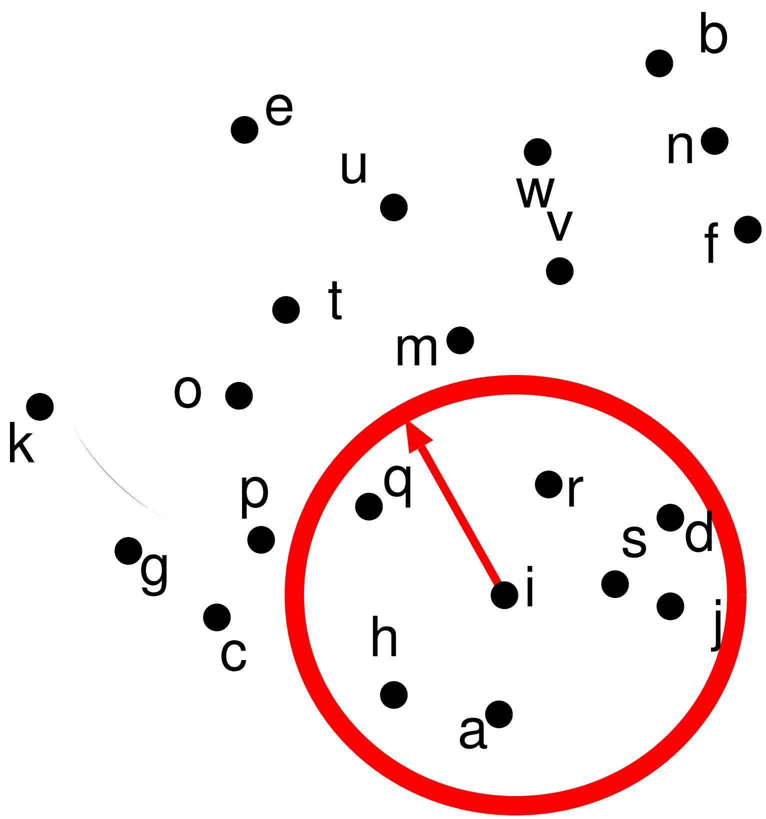
- Apply recursively, yielding a binary tree with pivot and radius values at internal nodes
- Choosing pivots
  1. simplest is to pick at random
  2. choose a random sample and then select median

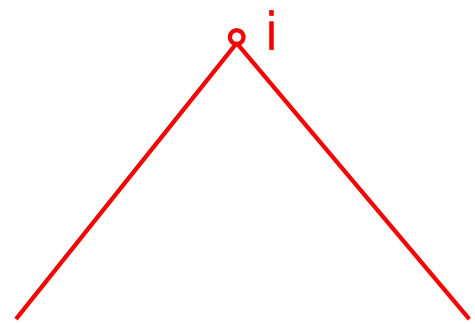


vp-Tree Example

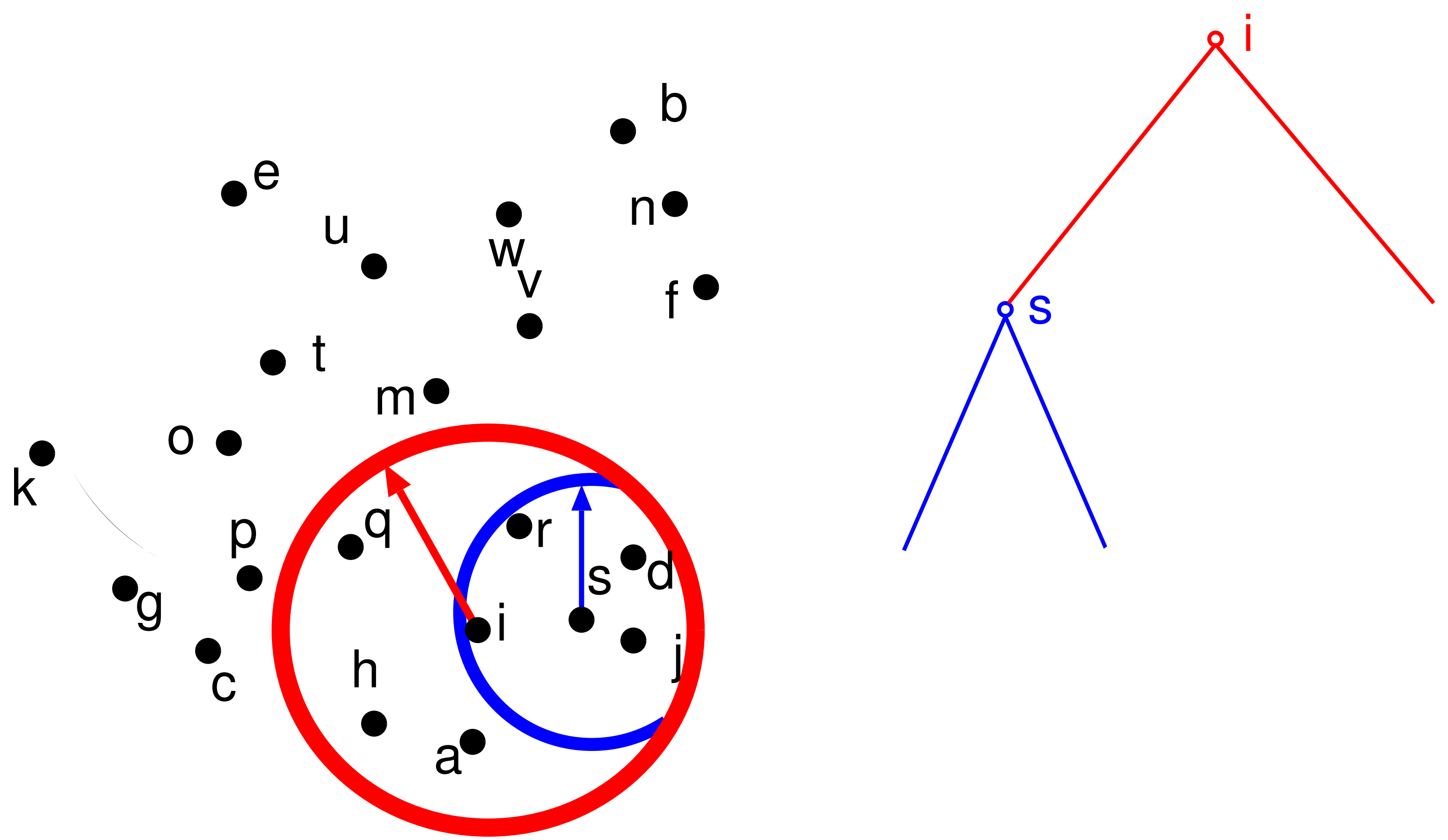


vp-Tree Example

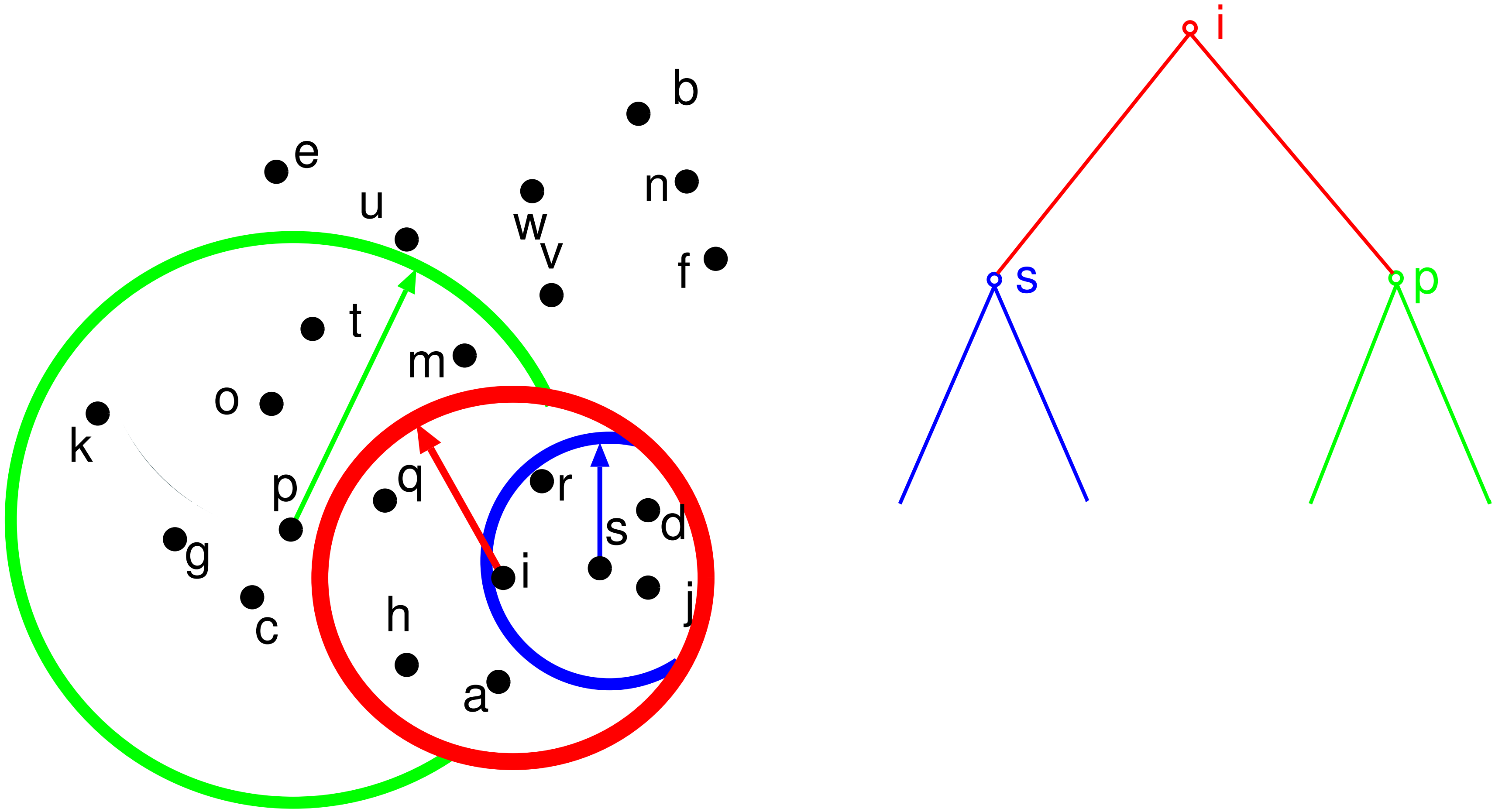




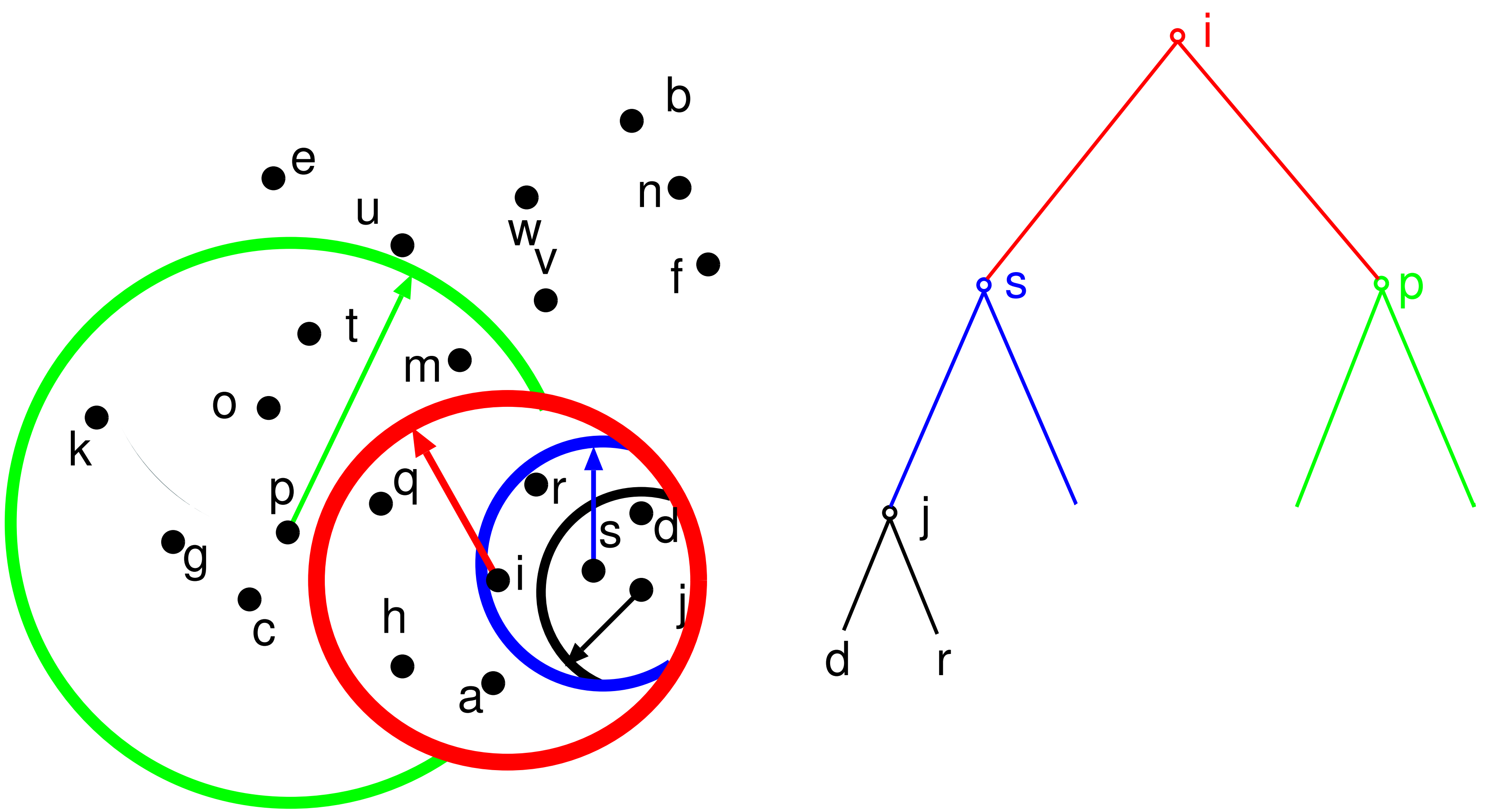
vp-Tree Example



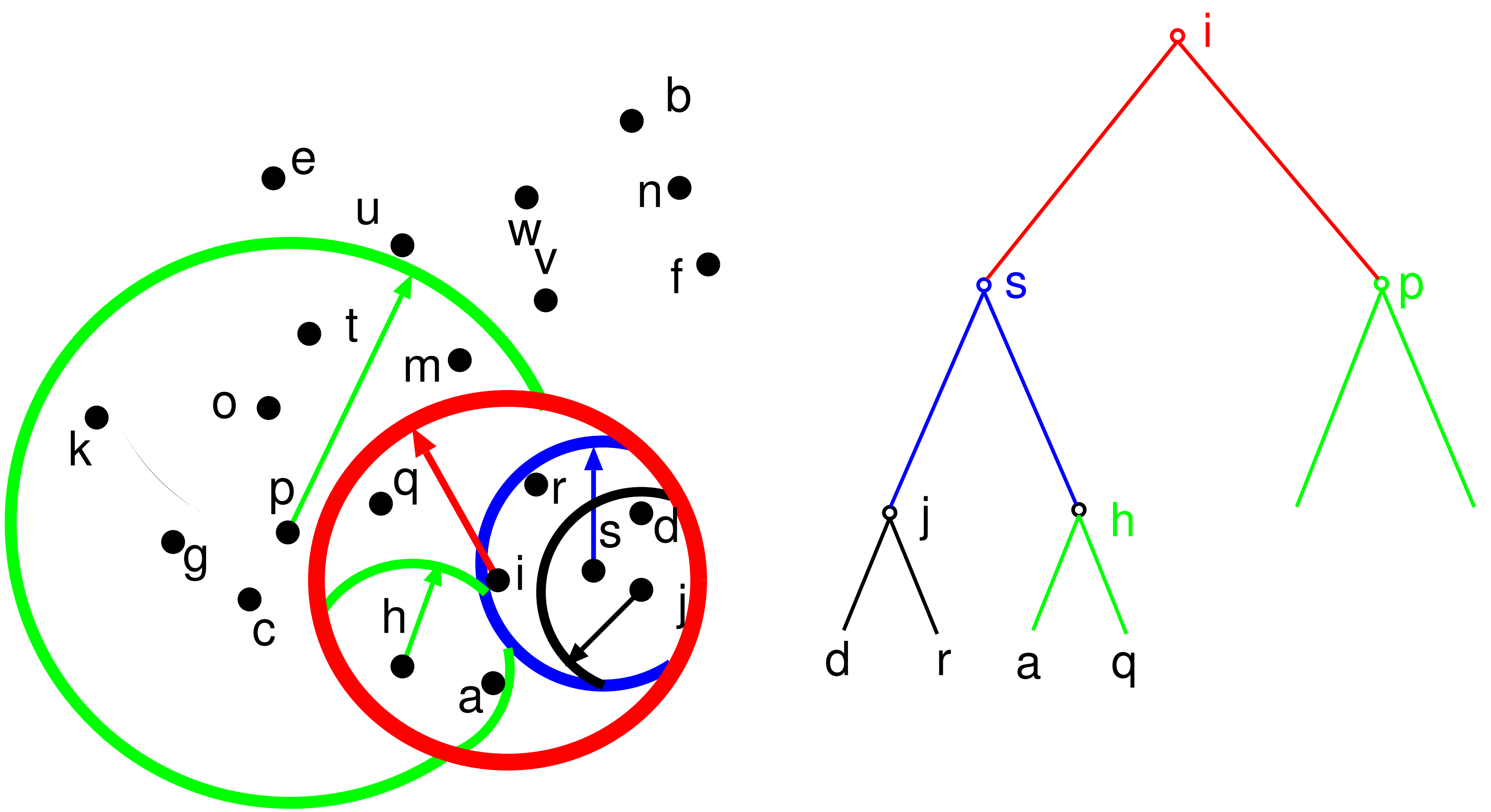
vp-Tree Example



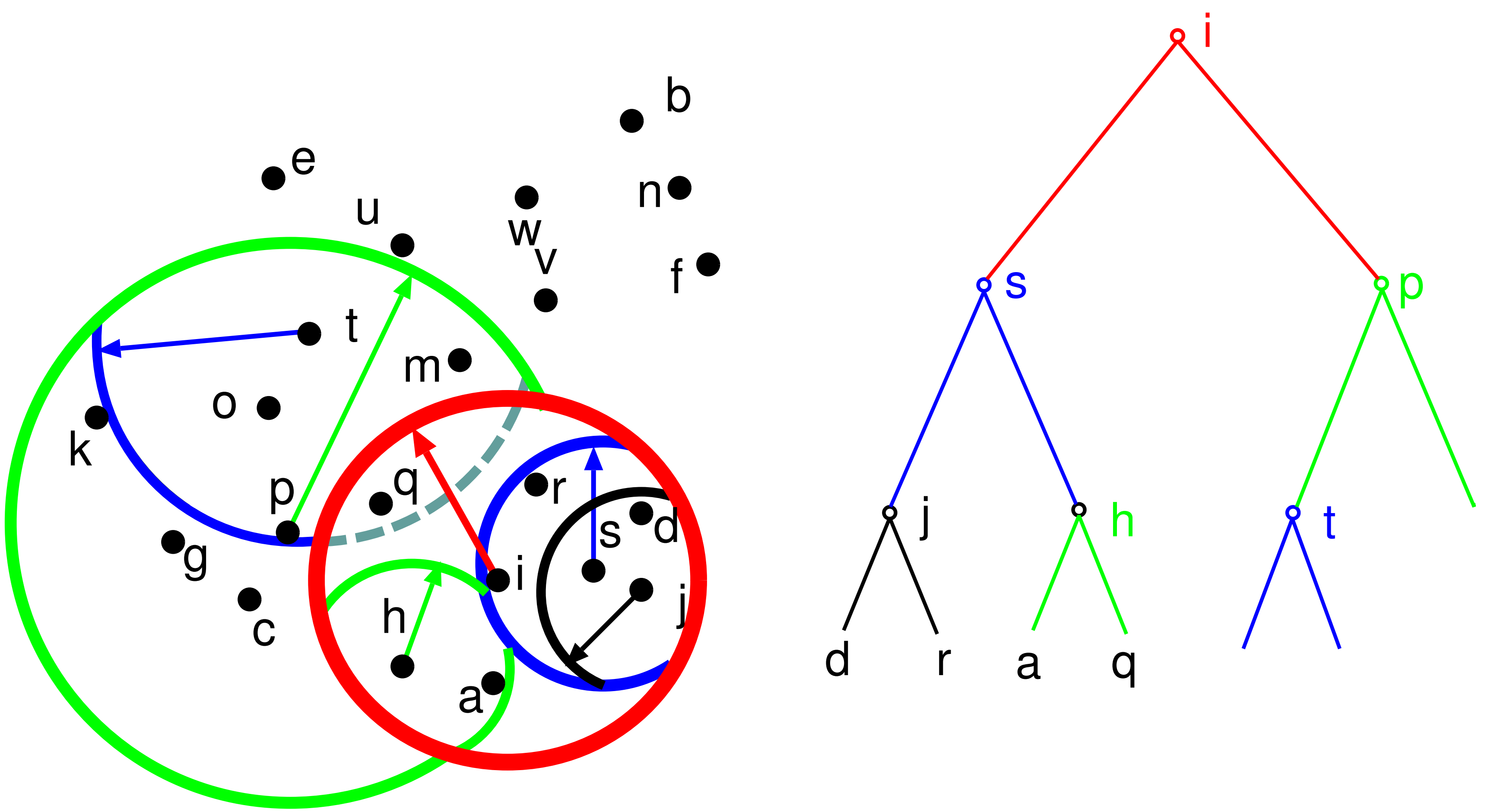
vp-Tree Example



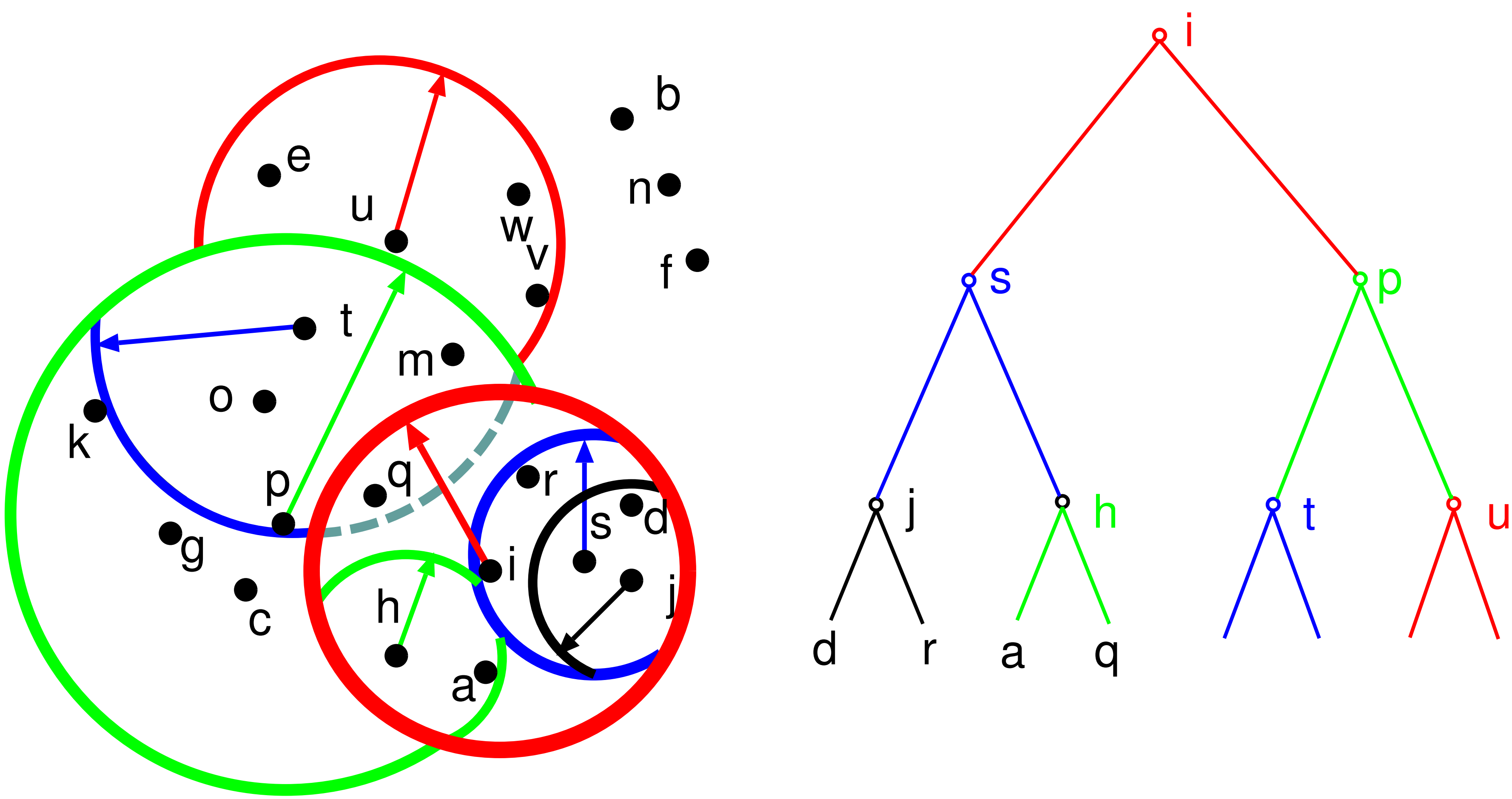
vp-Tree Example



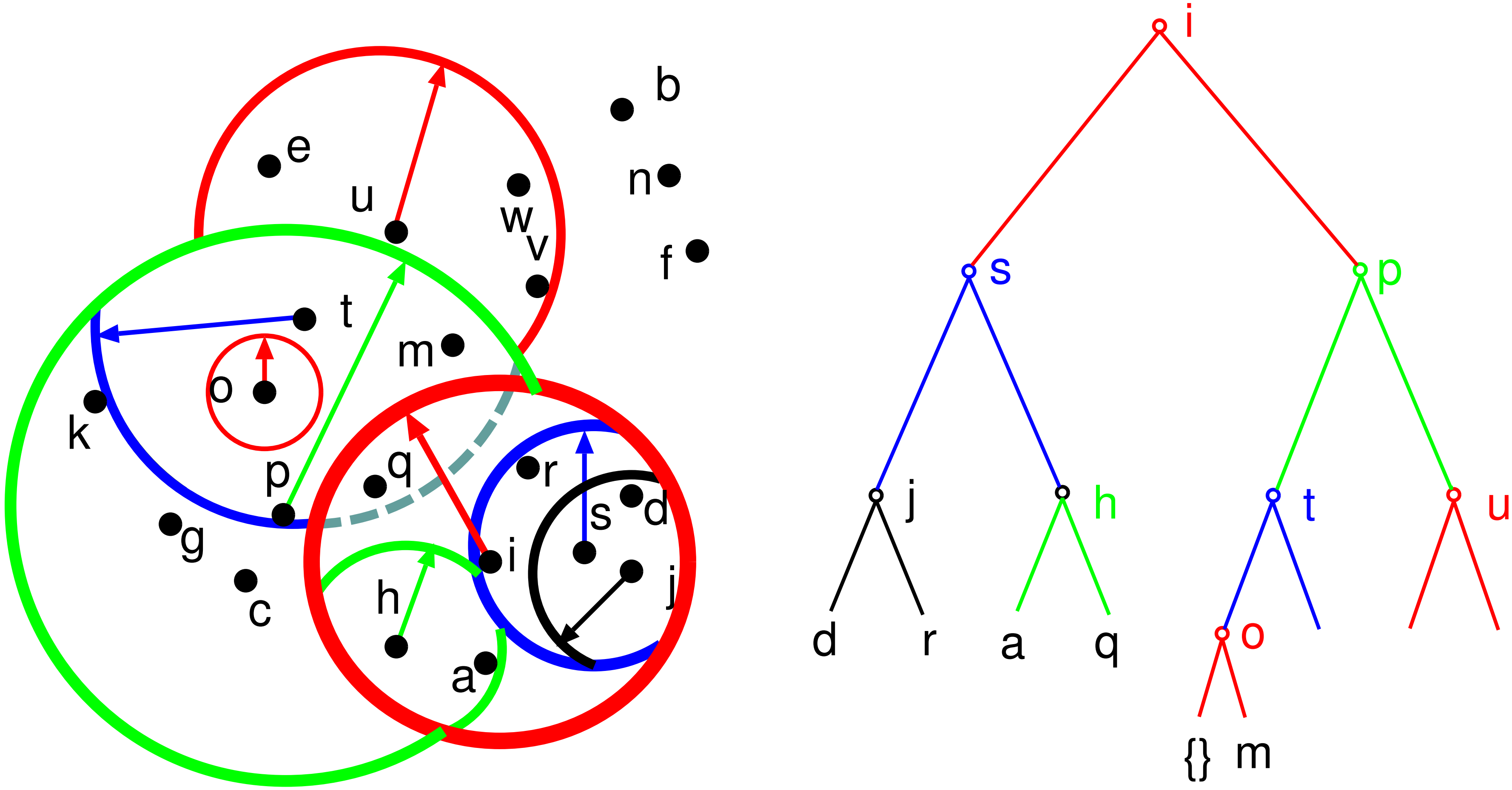
vp-Tree Example



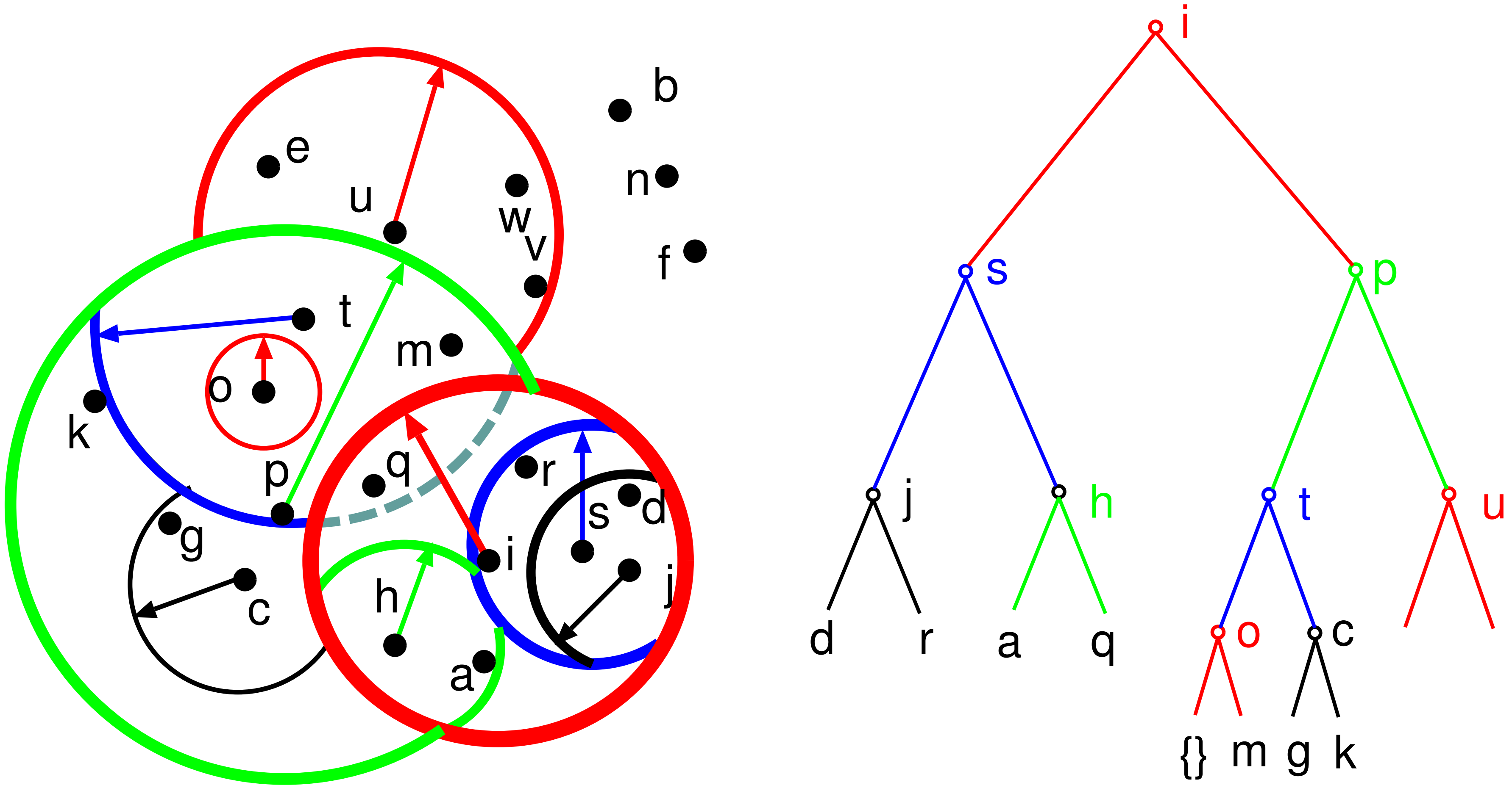
vp-Tree Example



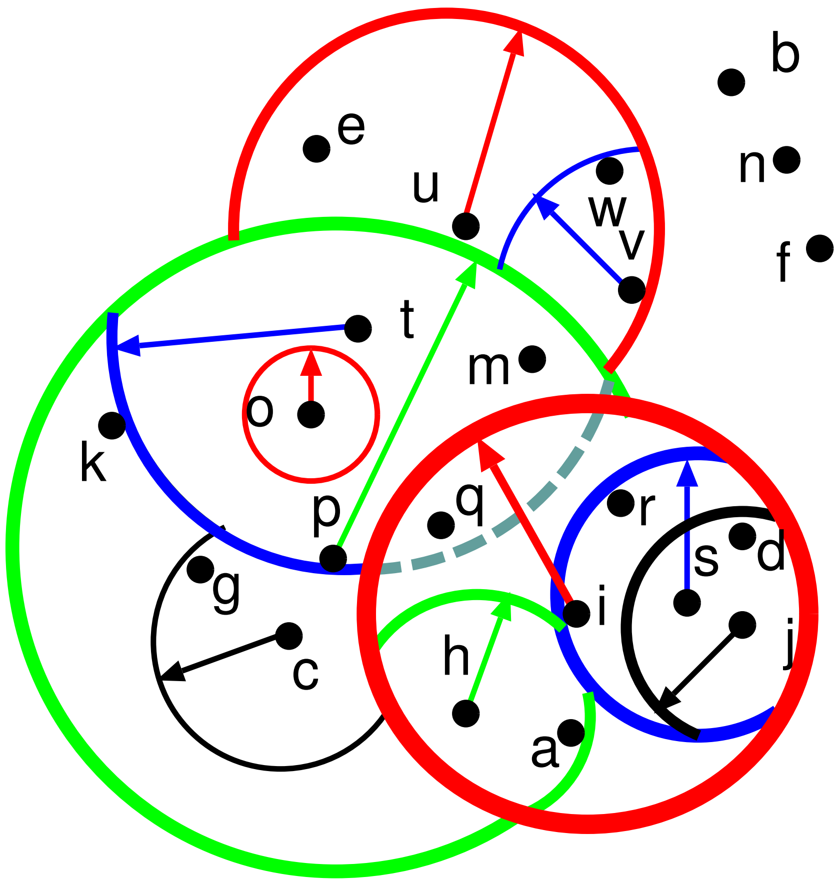
vp-Tree Example

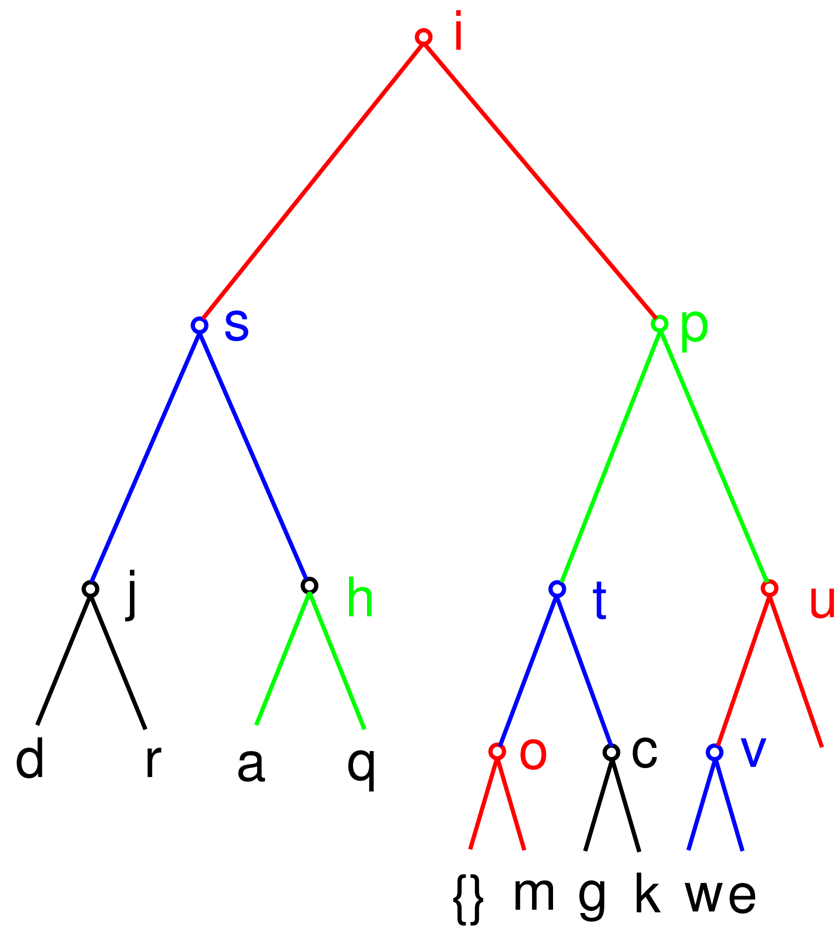


vp-Tree Example

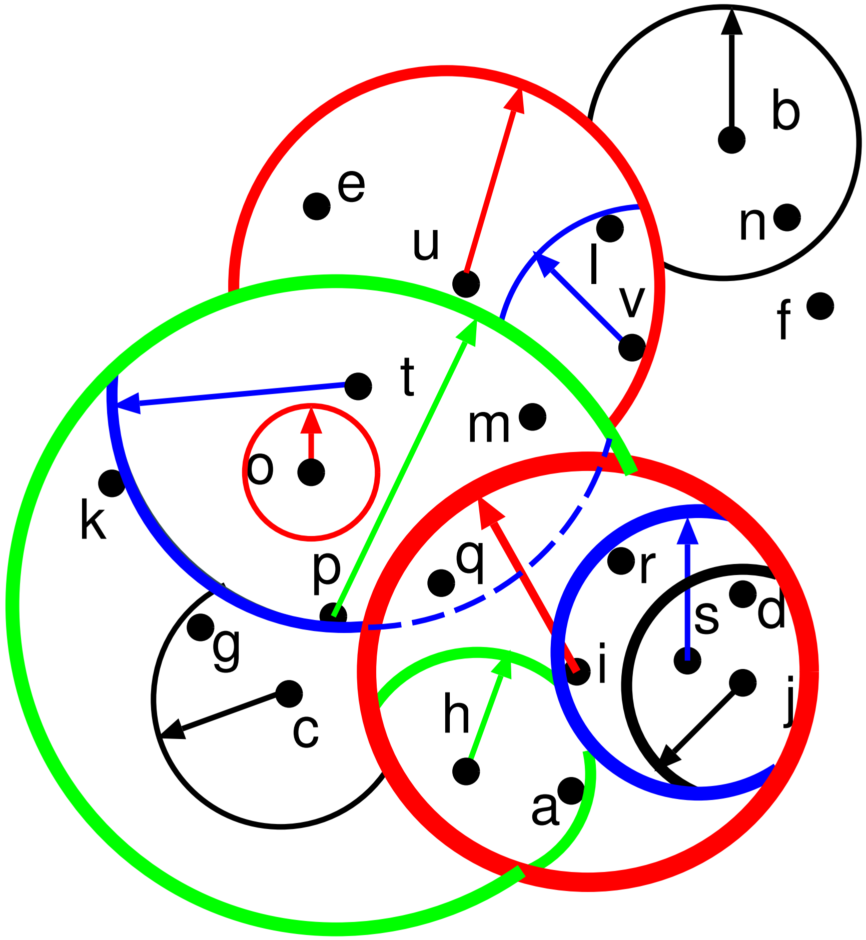


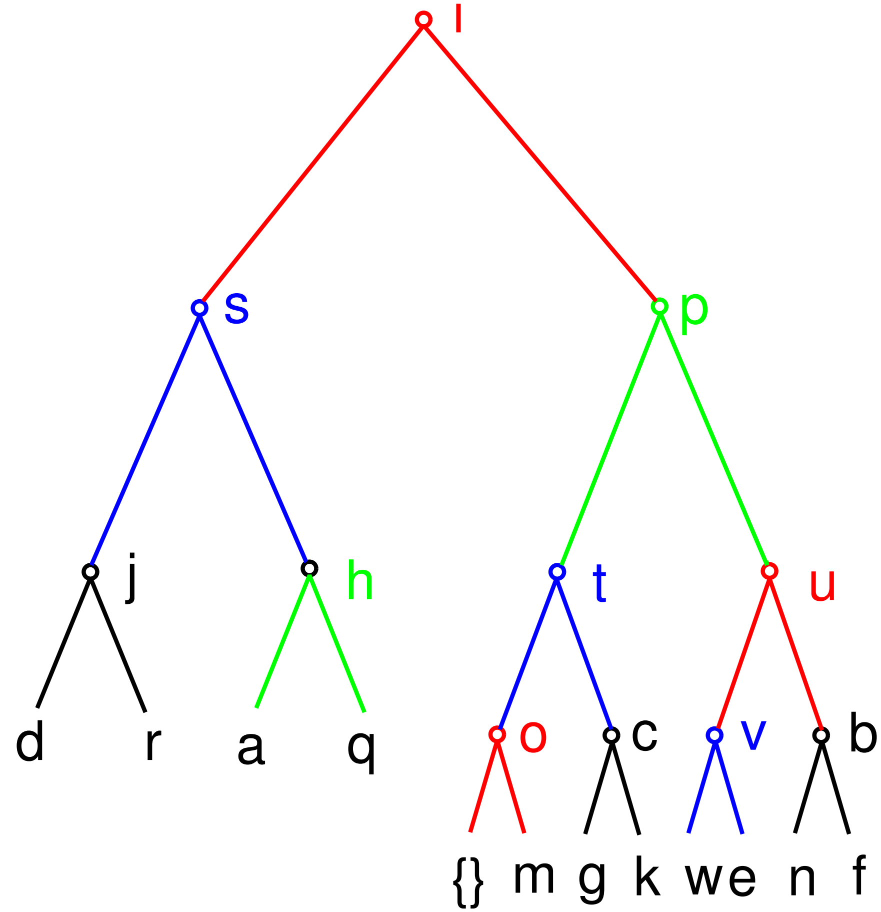
vp-Tree Example





vp-Tree Example



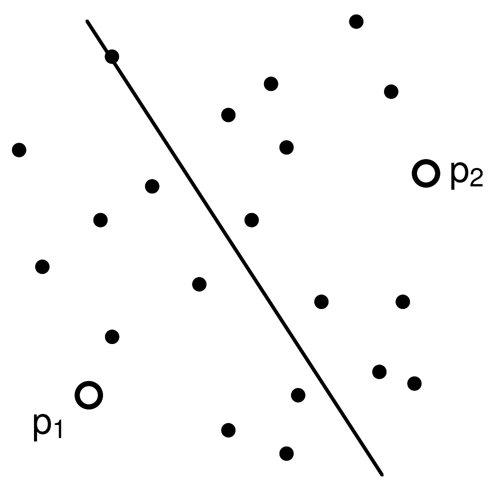


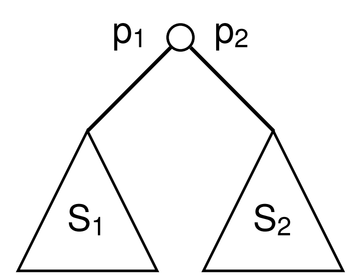
gh-Tree (Metric Tree; Uhlmann)

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

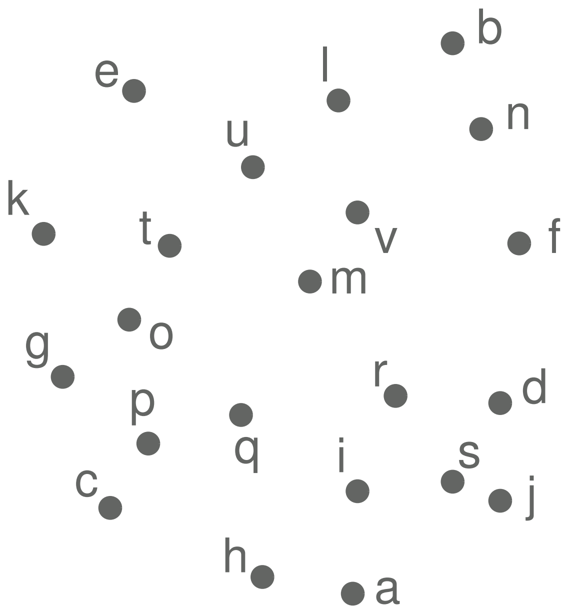
$$
\begin{aligned}
S_1 &= \{o \in S \setminus \{p_1, p_2\} \mid d(p_1, o) \le d(p_2, o)\} \\
S_2 &= \{o \in S \setminus \{p_1, p_2\} \mid d(p_2, o) < d(p_1, o)\}
\end{aligned}
$$

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





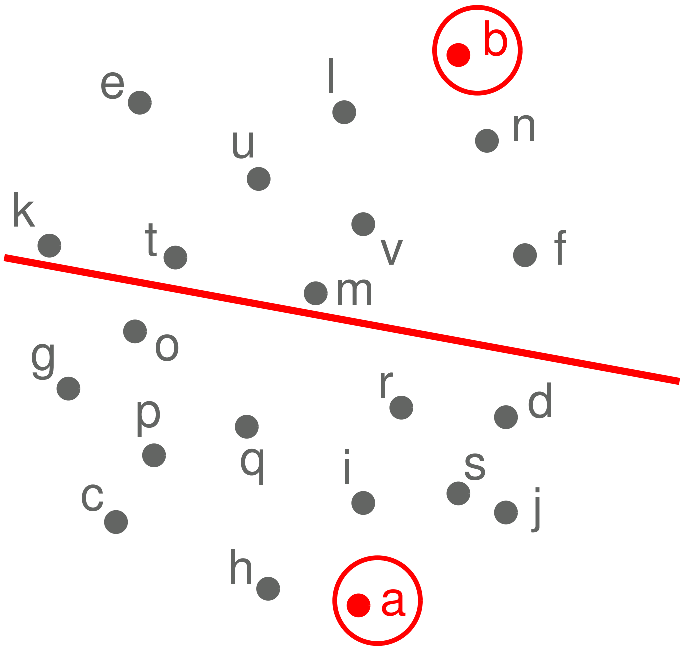
gh-Tree Example



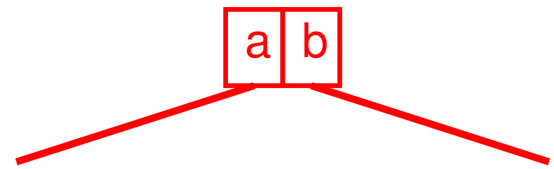
(a)

(b)

gh-Tree Example

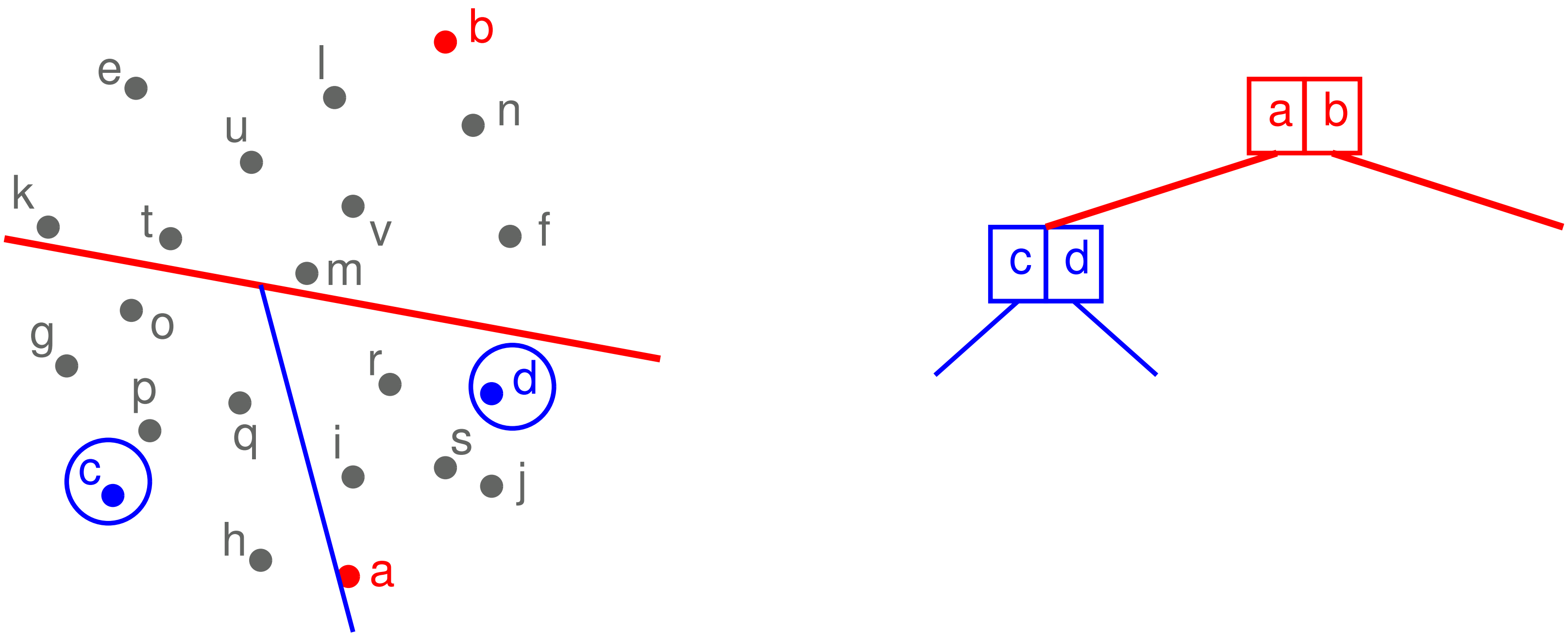


(a)



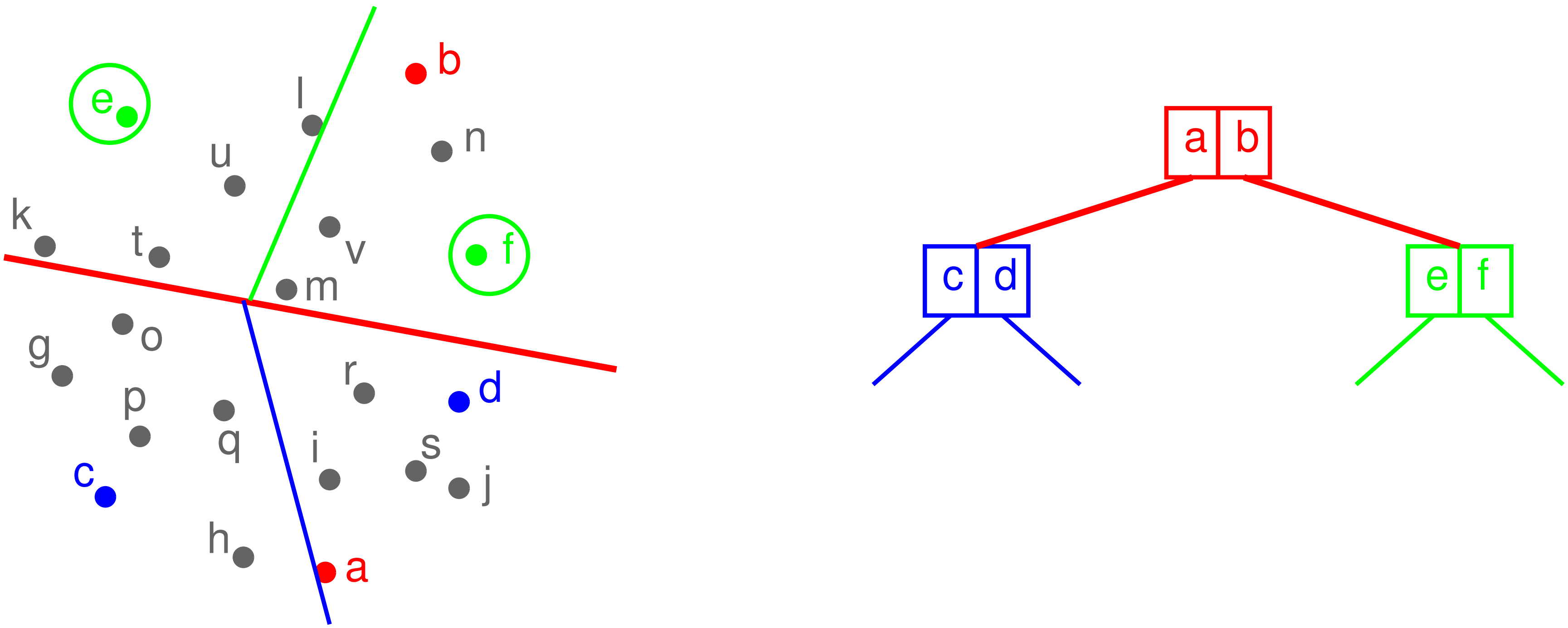
(b)

gh-Tree Example



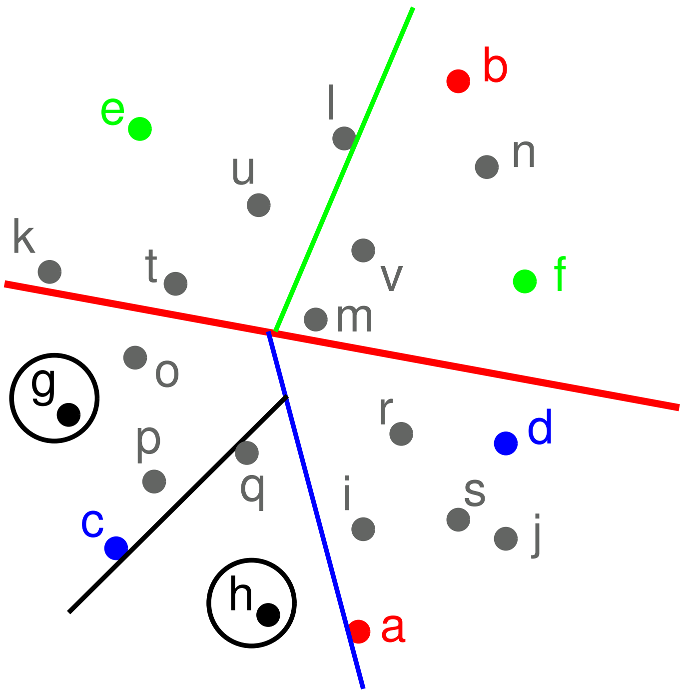
(a) (b)

gh-Tree Example

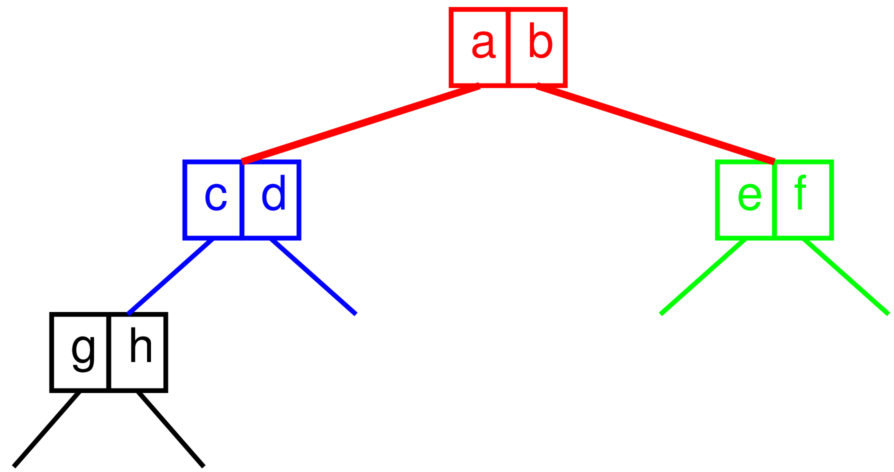


(a) (b)

gh-Tree Example

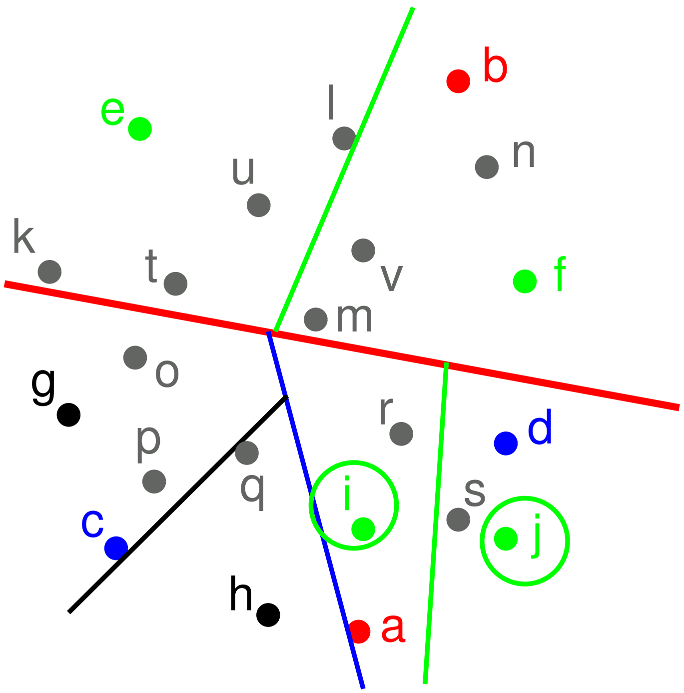


(a)

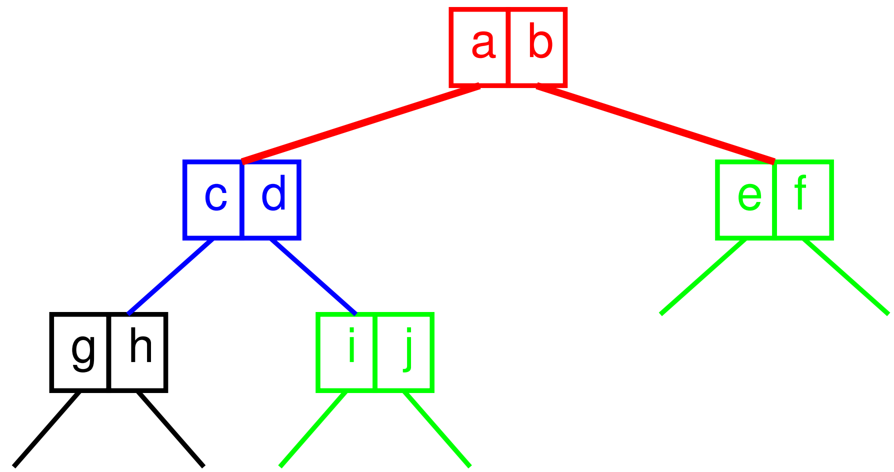


(b)

gh-Tree Example

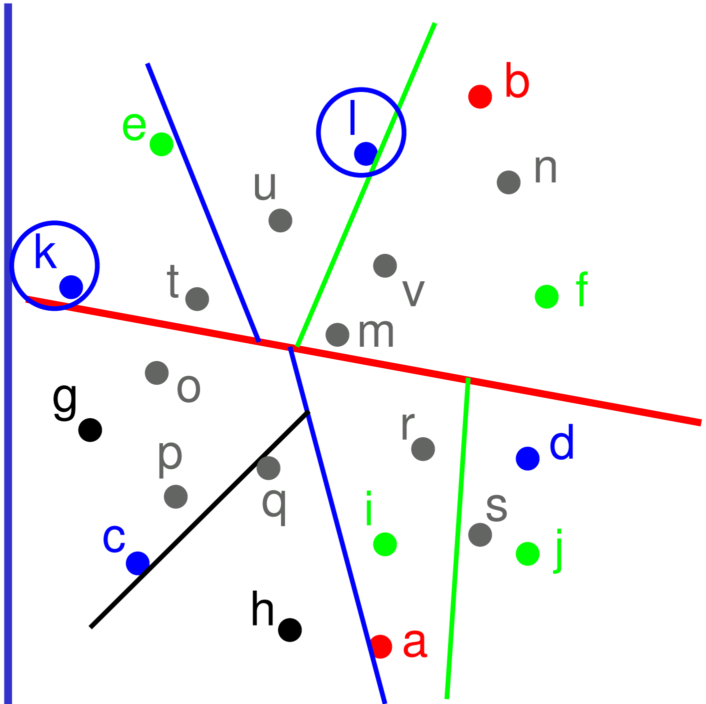


(a)

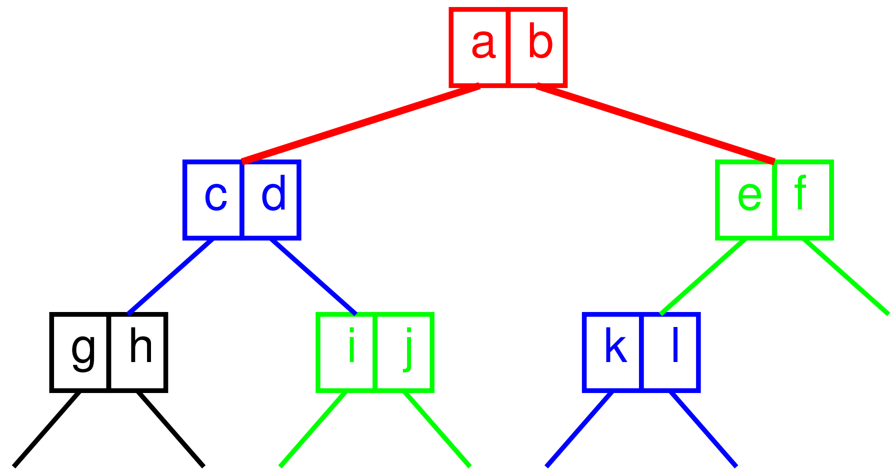


(b)

gh-Tree Example

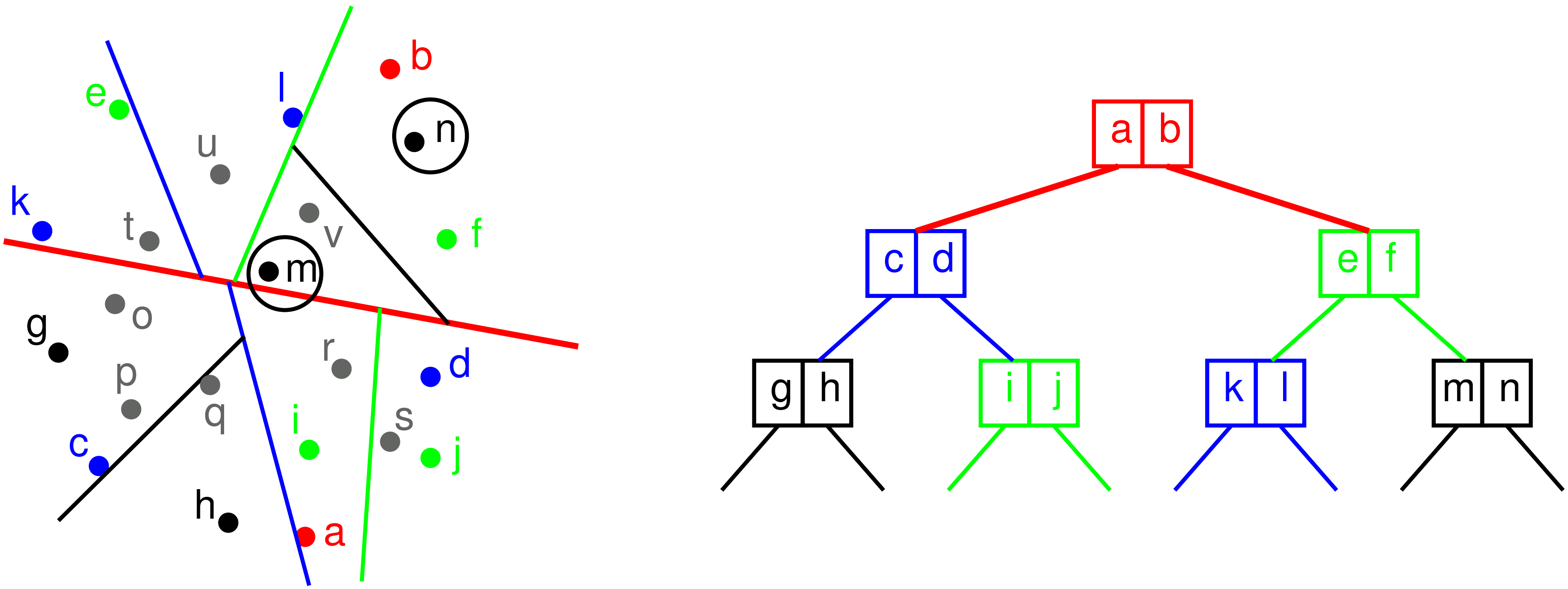


(a)



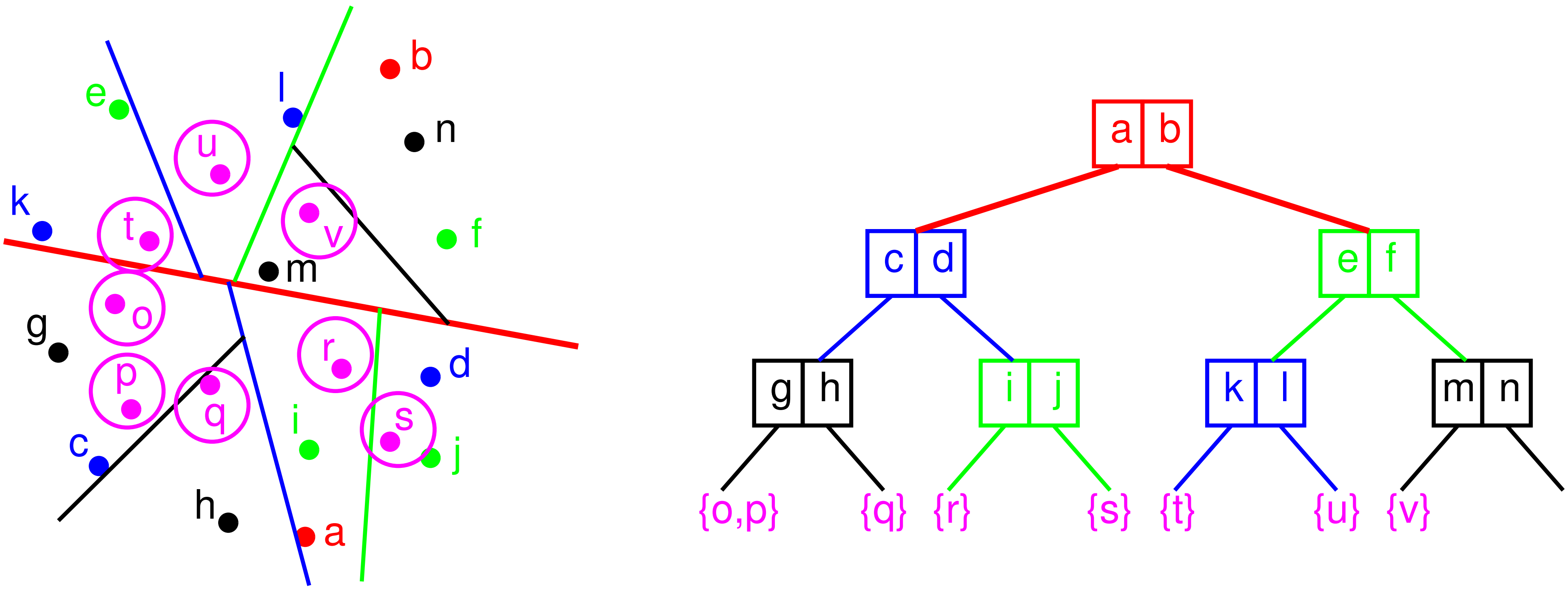
(b)

gh-Tree Example



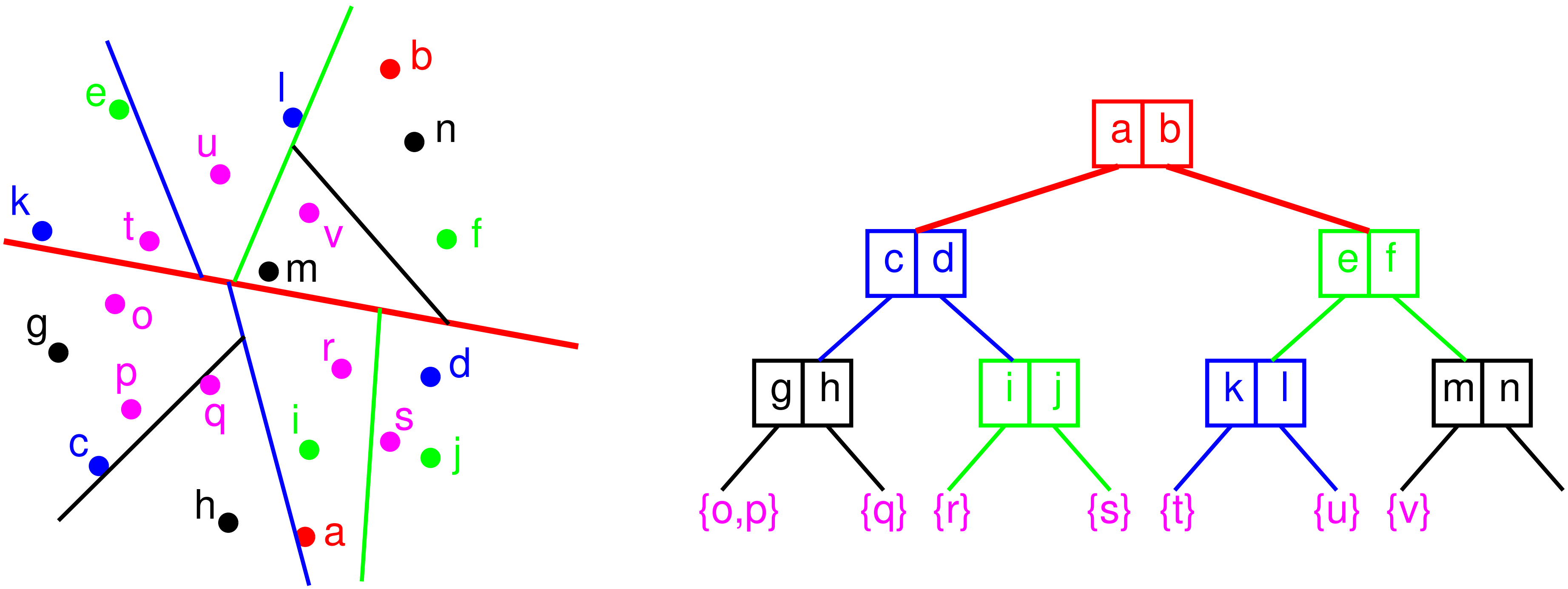
(a) (b)

gh-Tree Example



(a) (b)

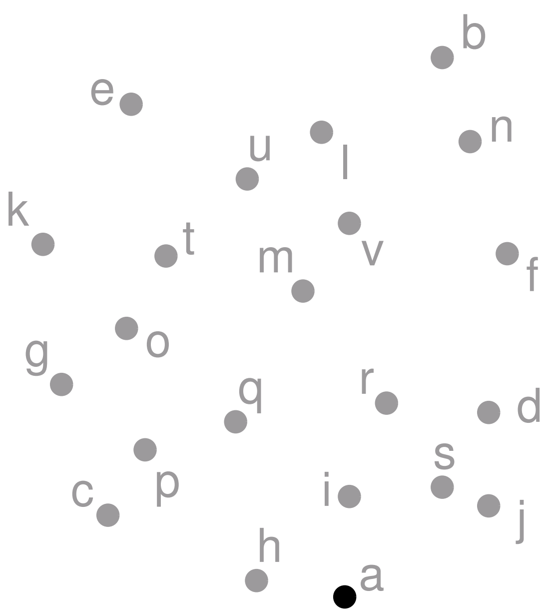
gh-Tree Example



(a) (b)

mb-Tree (Dehne/Noltemeier)

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

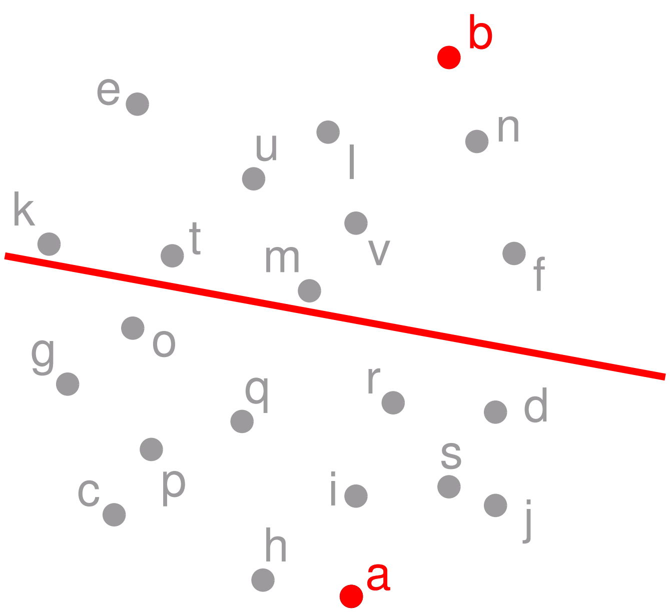


(a)

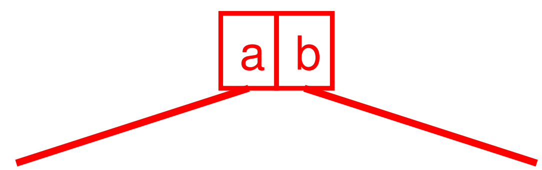
(b)

mb-Tree (Dehne/Noltemeier)

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



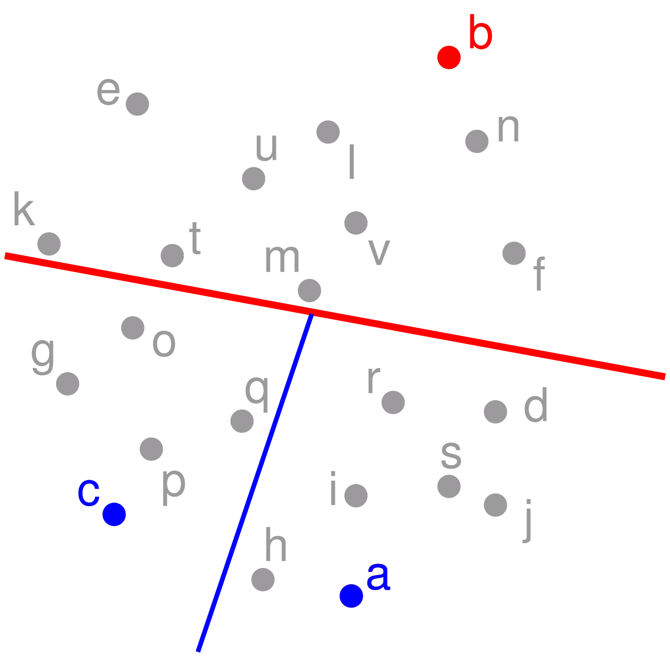
(a)



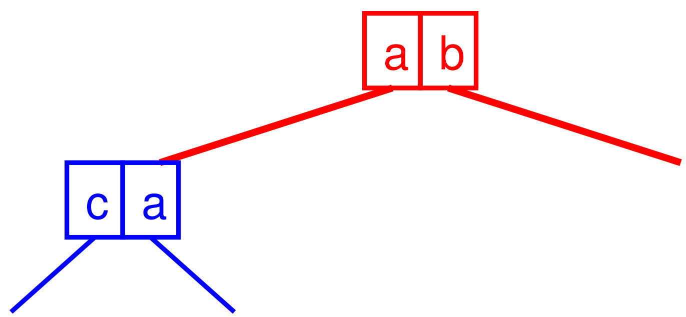
(b)

mb-Tree (Dehne/Noltemeier)

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



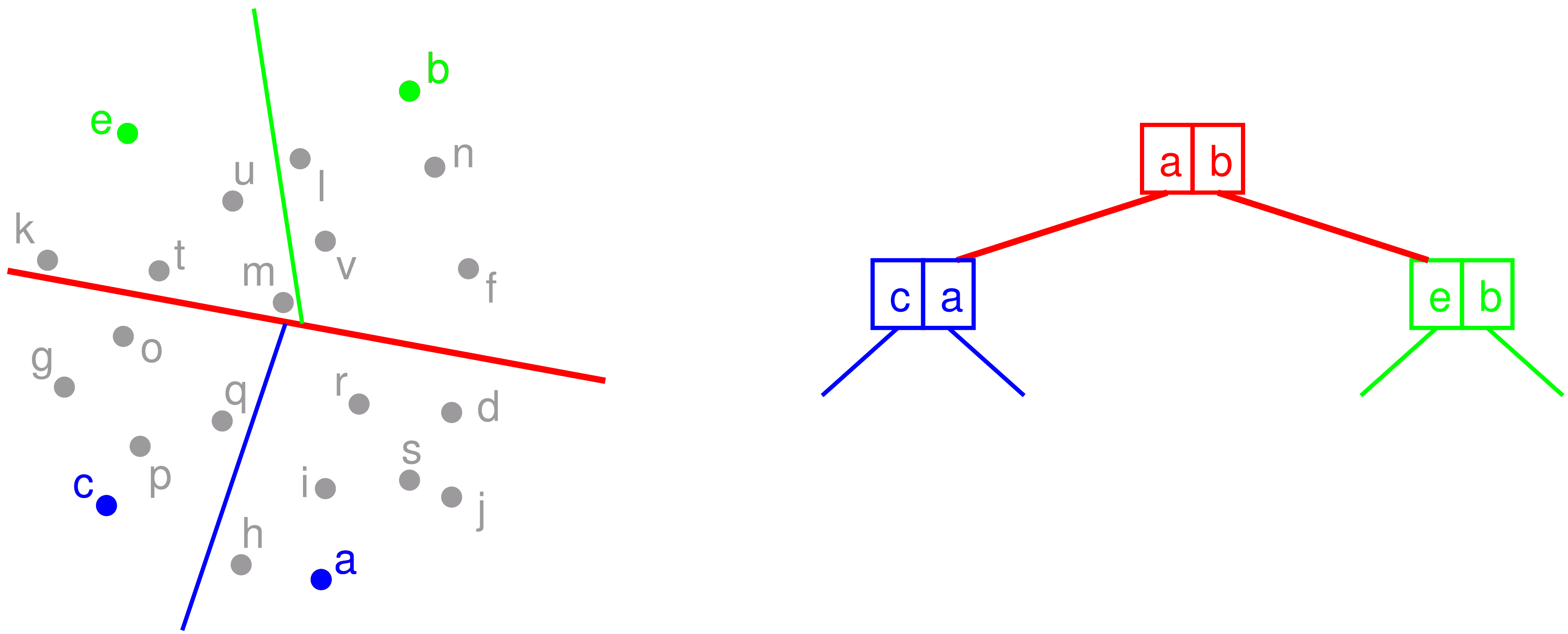
(a)



(b)

mb-Tree (Dehne/Noltemeier)

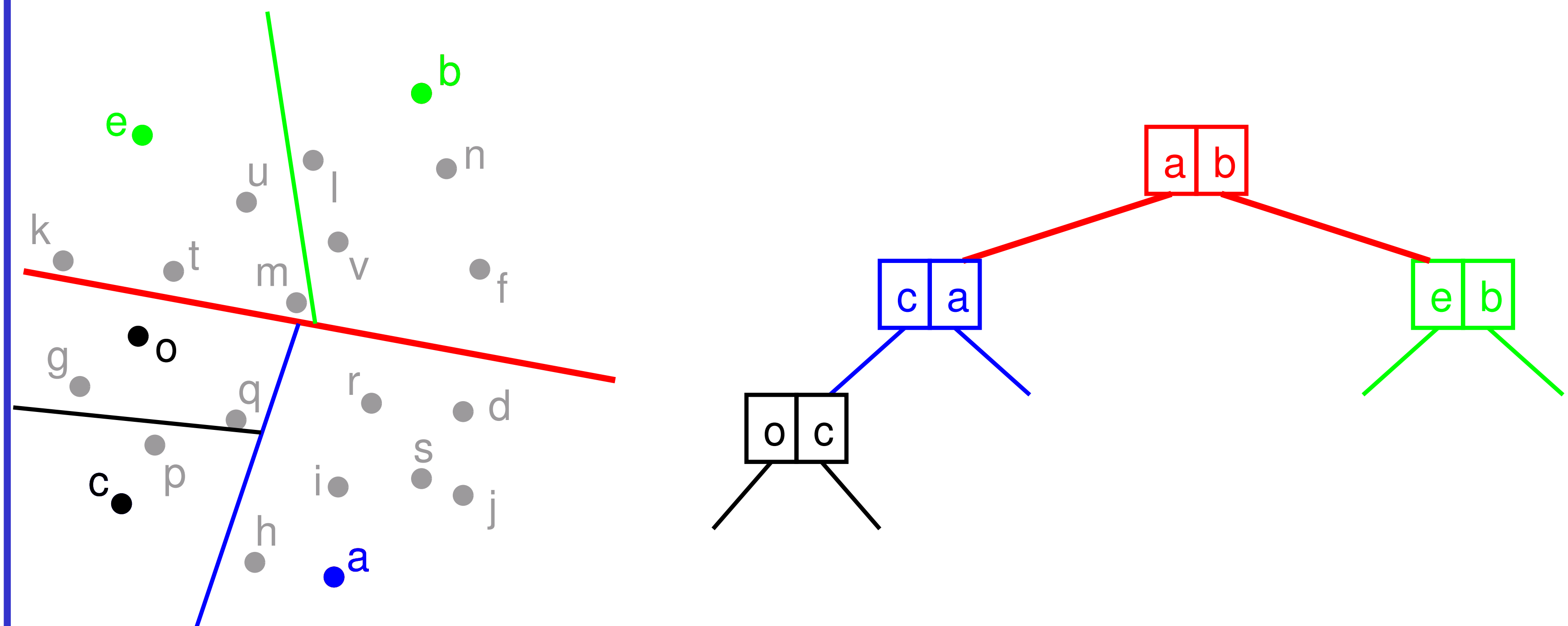
1. Inherit one pivot from ancestor node
2. Fewer pivots and fewer distance computations but perhaps deeper tree
3. Like bucket (k) PR k-d tree as split whenever region has $k > 1$ objects but region partitions are implicit (defined by pivot objects) instead of explicit



(a) (b)

mb-Tree (Dehne/Noltemeier)

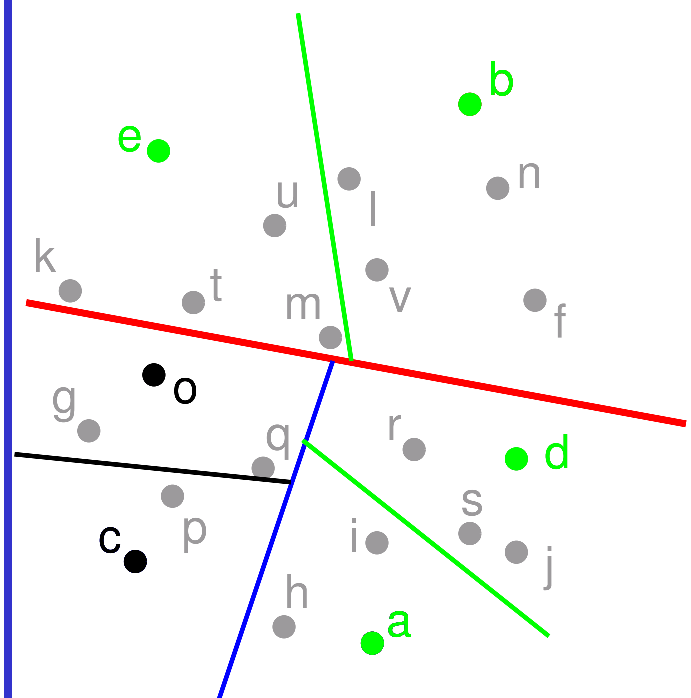
1. Inherit one pivot from ancestor node
2. Fewer pivots and fewer distance computations but perhaps deeper tree
3. Like bucket (k) PR k-d tree as split whenever region has $k > 1$ objects but region partitions are implicit (defined by pivot objects) instead of explicit



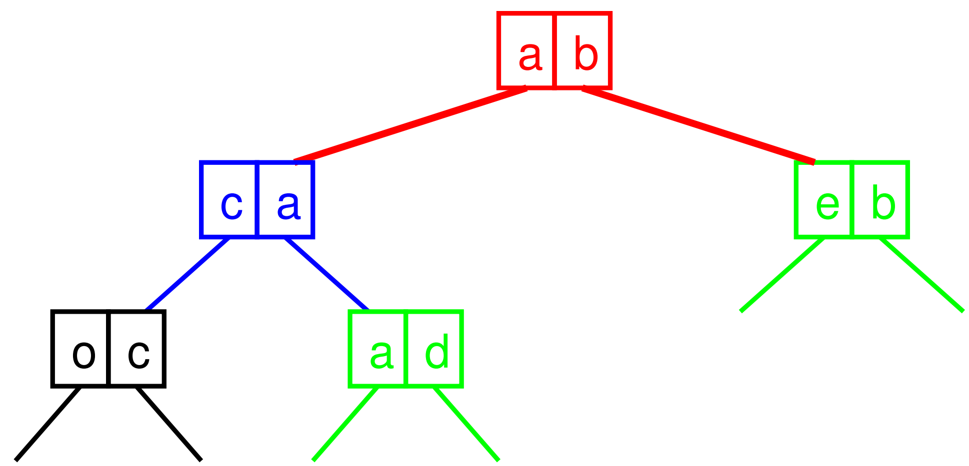
(a) (b)

mb-Tree (Dehne/Noltemeier)

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



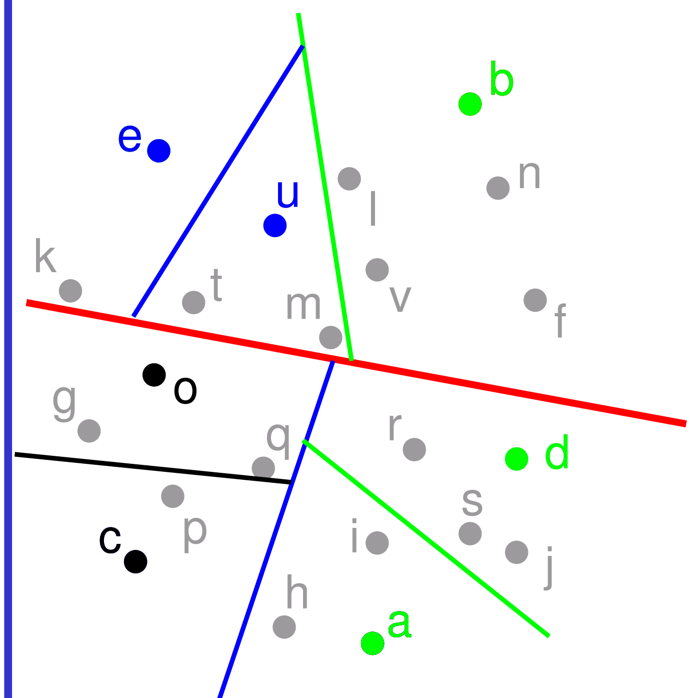
(a)



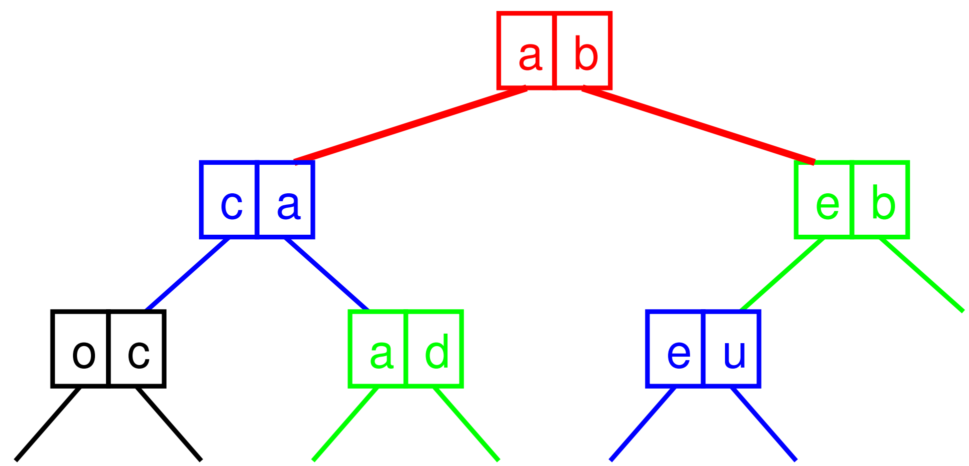
(b)

mb-Tree (Dehne/Noltemeier)

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



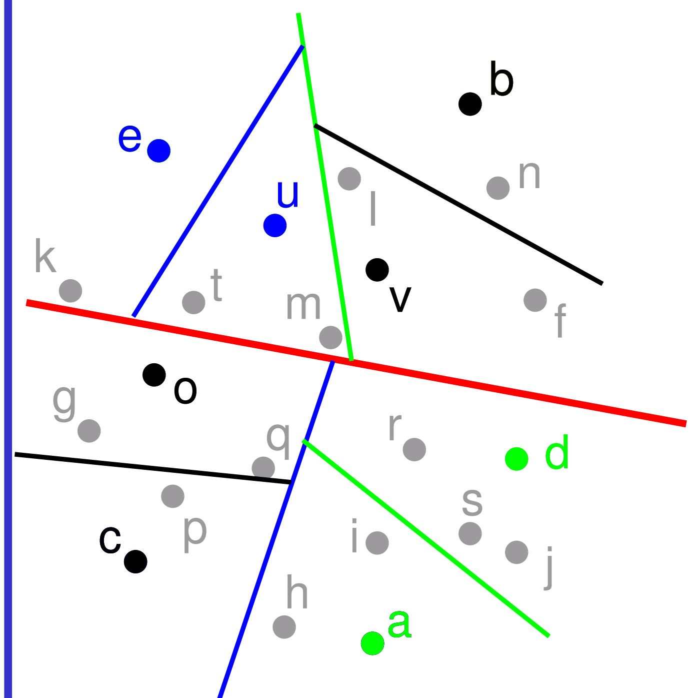
(a)



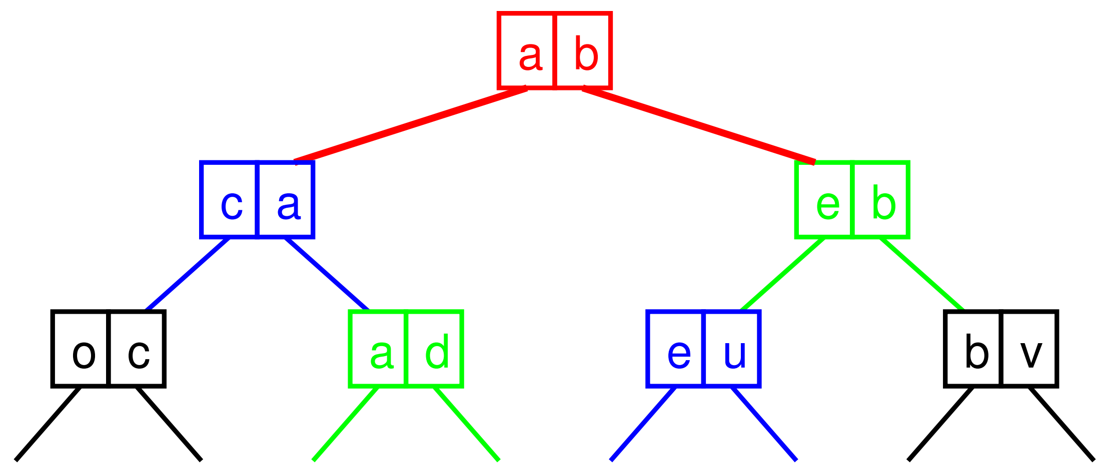
(b)

mb-Tree (Dehne/Noltemeier)

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



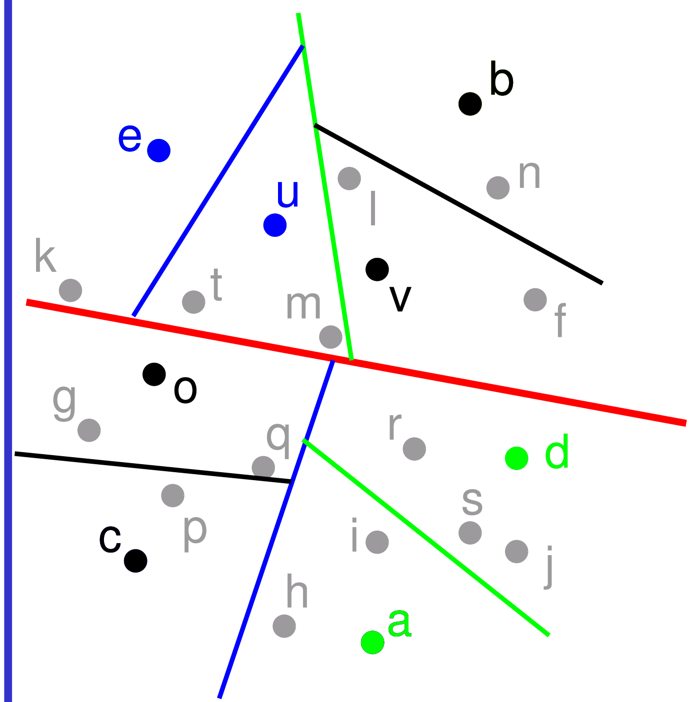
(a)



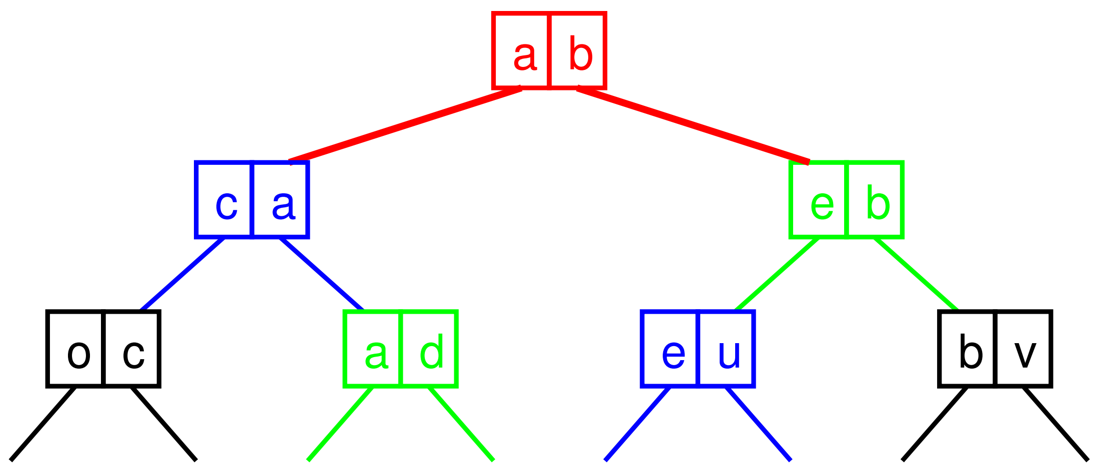
(b)

mb-Tree (Dehne/Noltemeier)

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



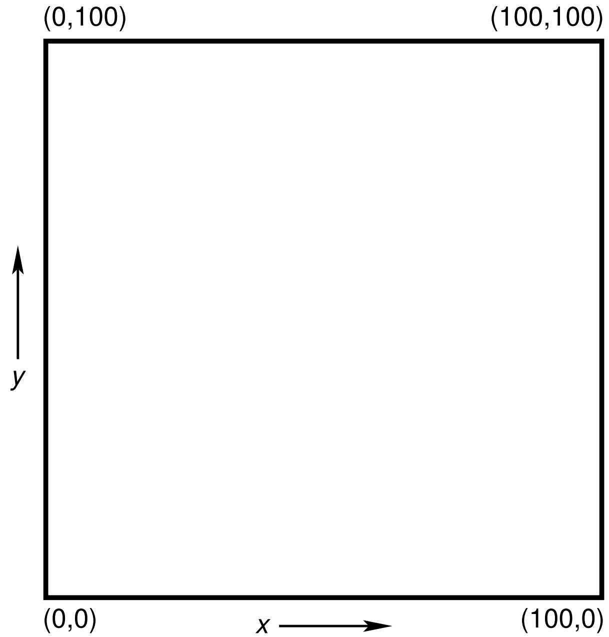
(a)



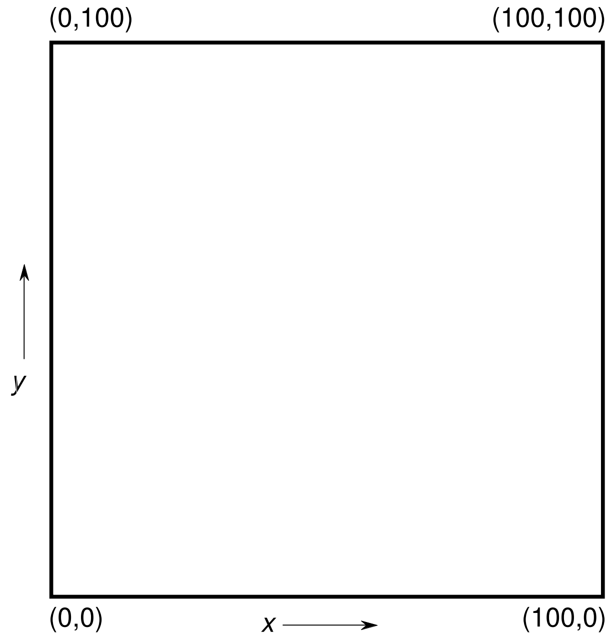
(b)

Comparison of mb-tree (BSP tree) and PR k-d tree

PR k-d tree

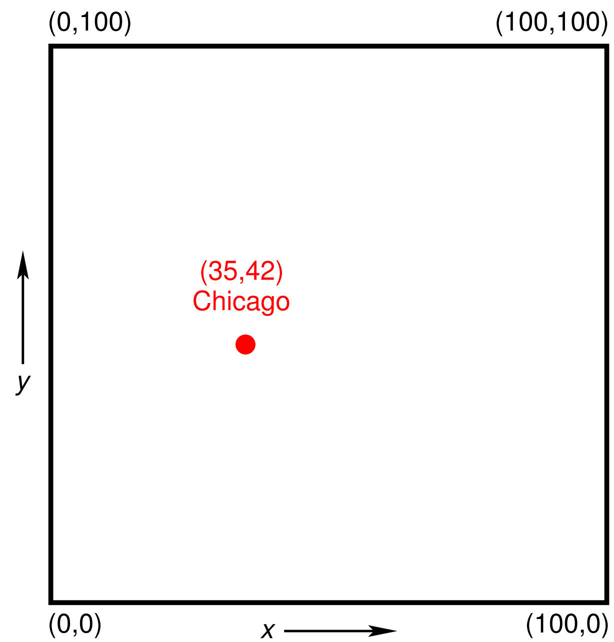


mb-tree

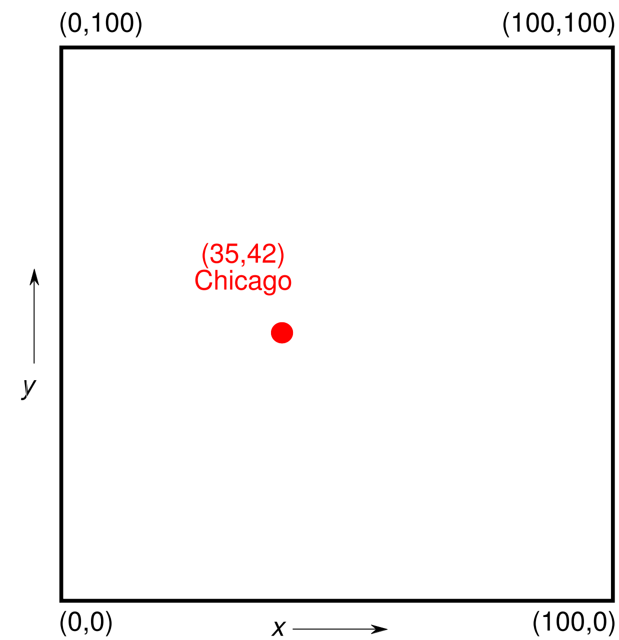


Comparison of mb-tree (BSP tree) and PR k-d tree

PR k-d tree

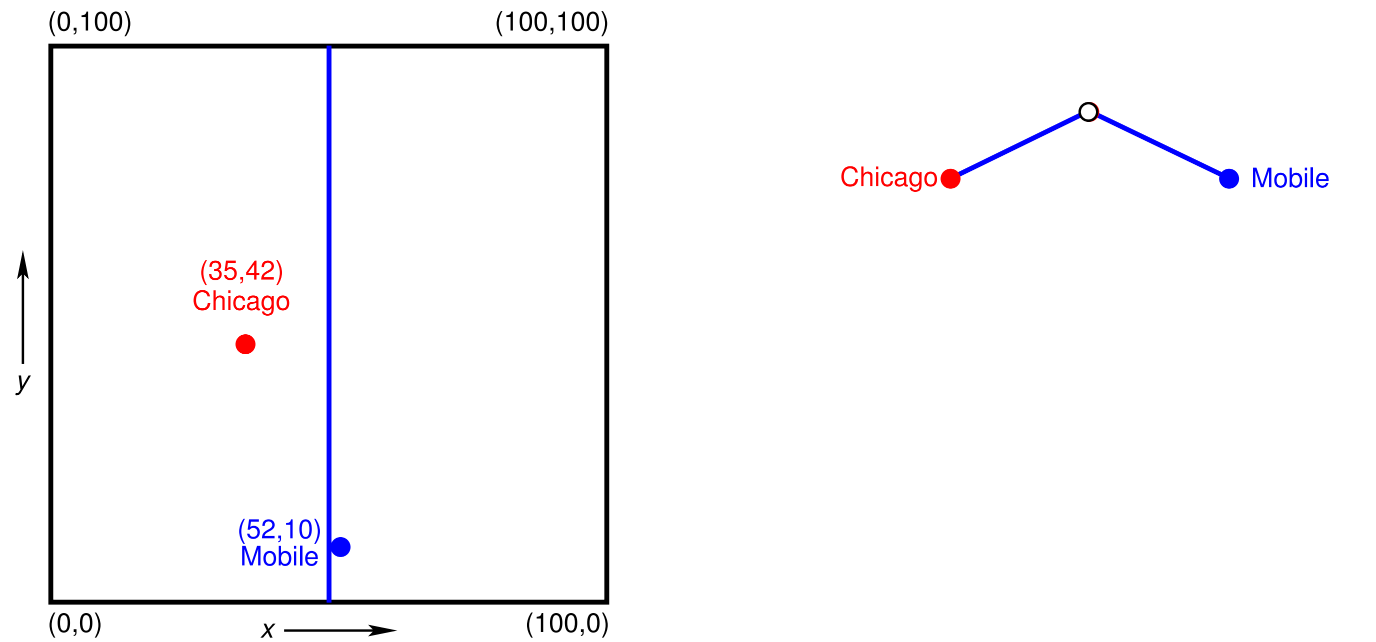


mb-tree

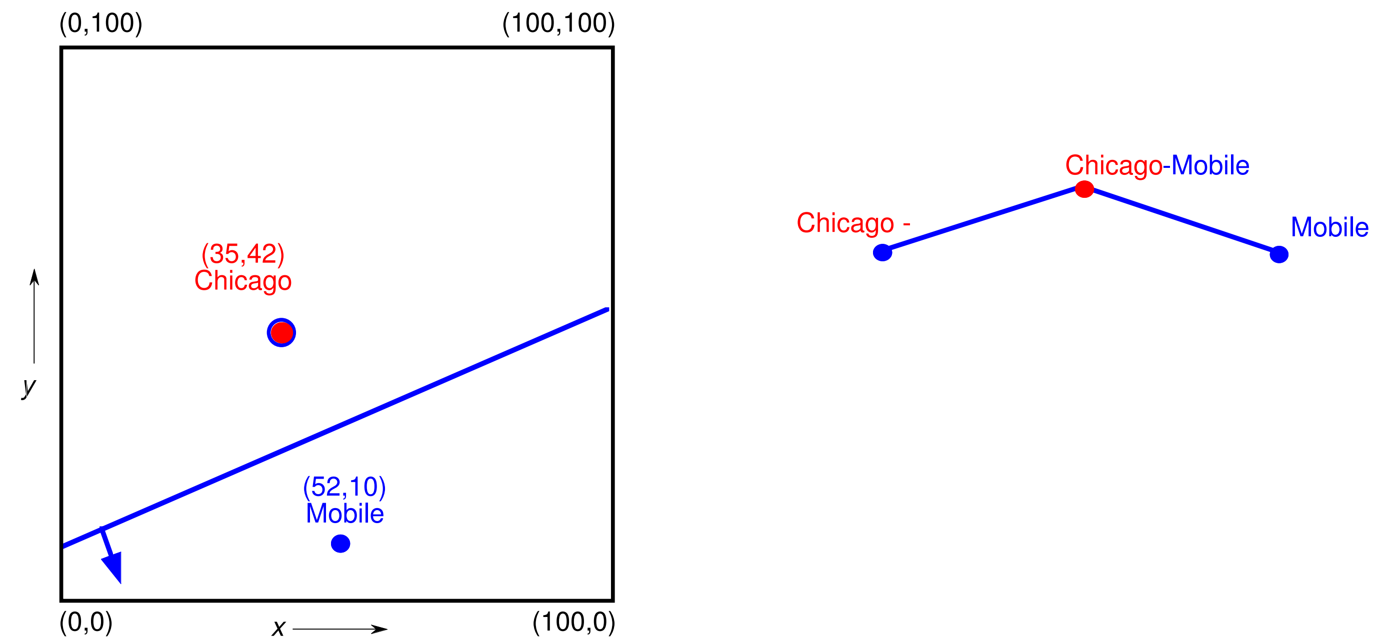


Comparison of mb-tree (BSP tree) and PR k-d tree

PR k-d tree

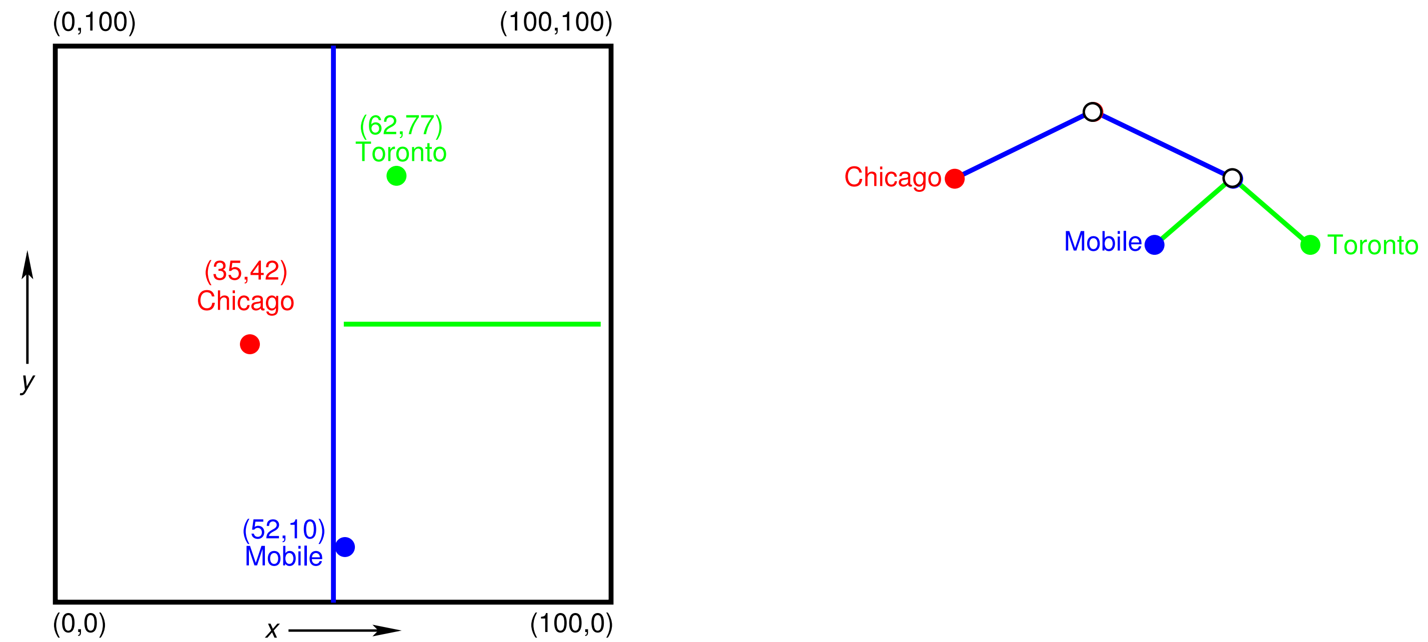


mb-tree

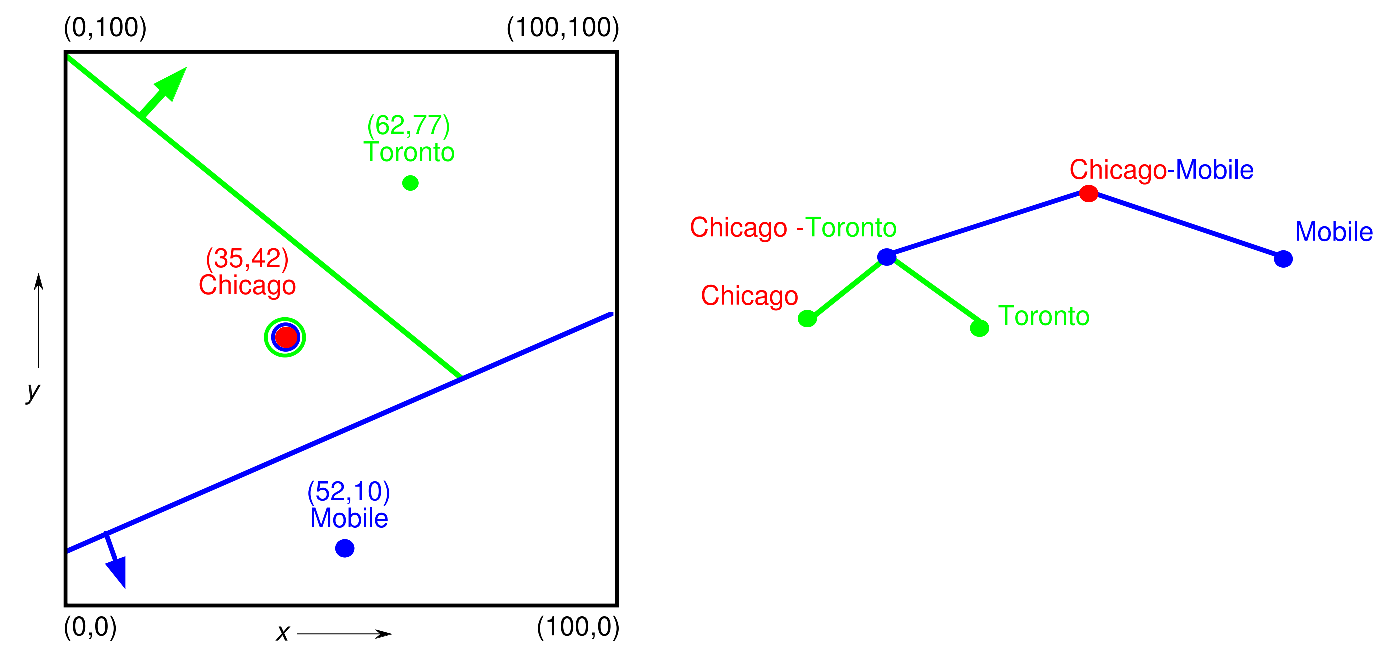


Comparison of mb-tree (BSP tree) and PR k-d tree

PR k-d tree

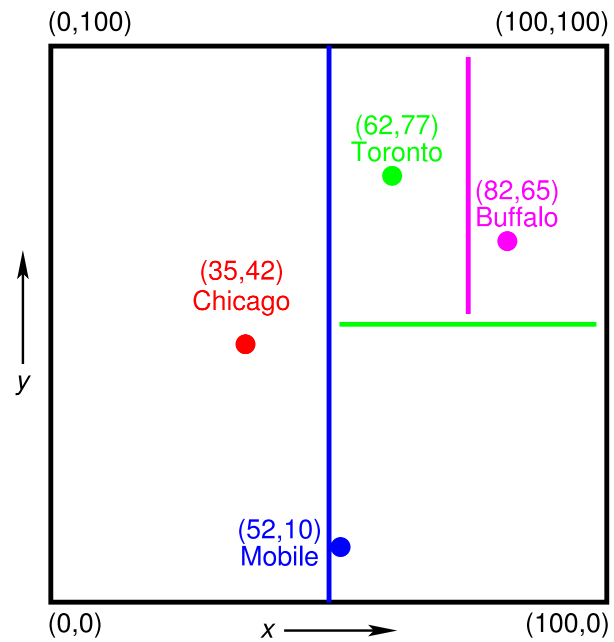


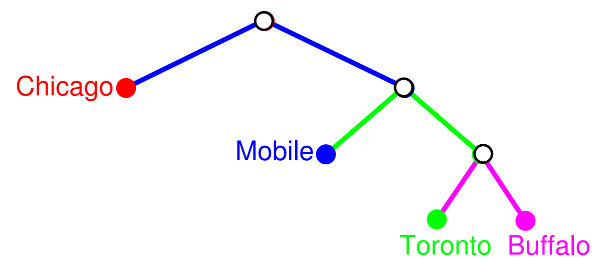
mb-tree



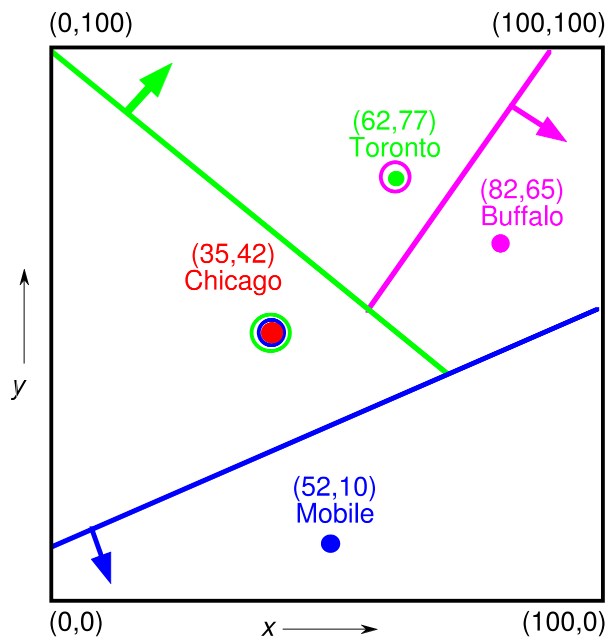
Comparison of mb-tree (BSP tree) and PR k-d tree

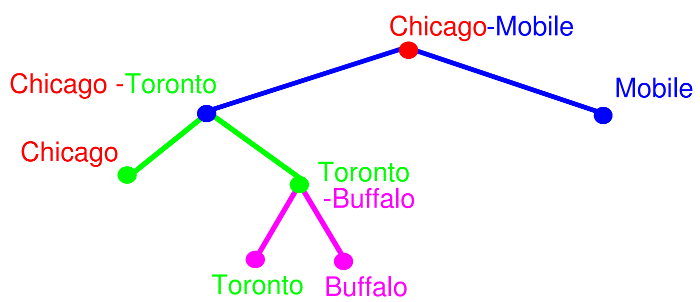
PR k-d tree





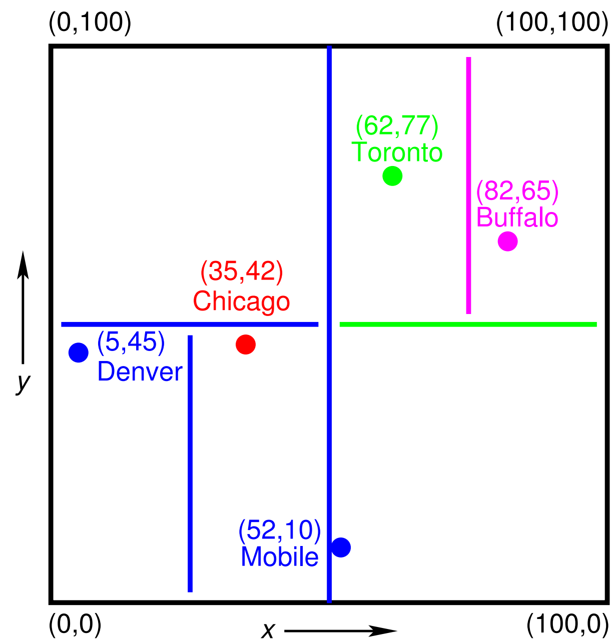
mb-tree

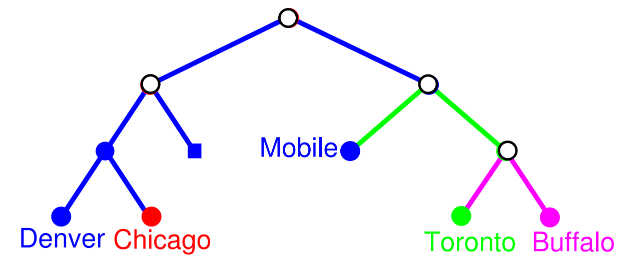




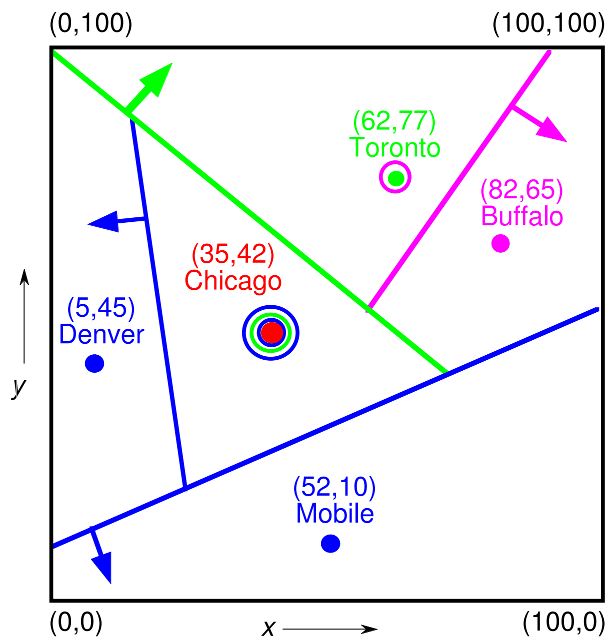
Comparison of mb-tree (BSP tree) and PR k-d tree

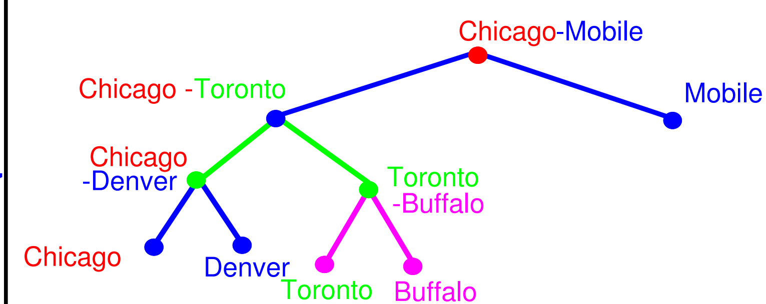
PR k-d tree





mb-tree

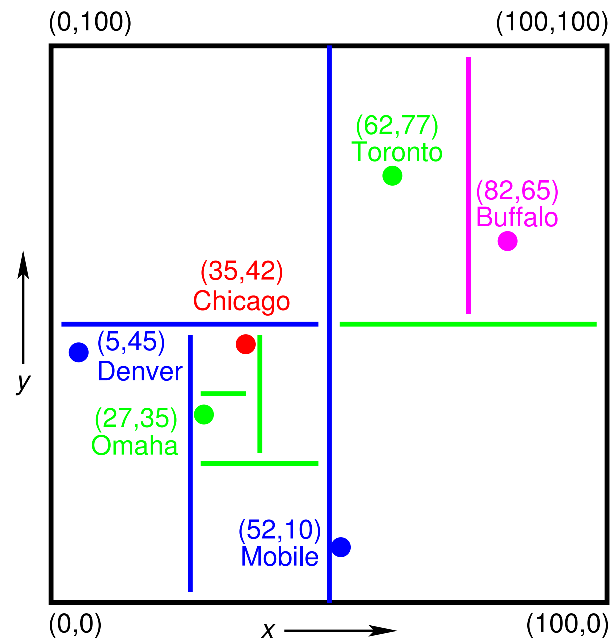


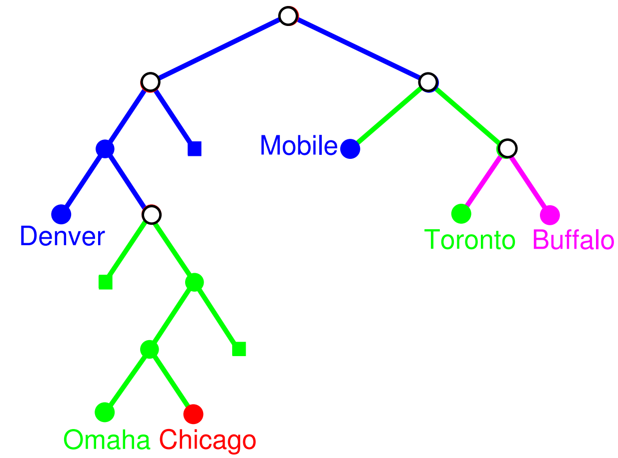


618-622 -- Mb-tree

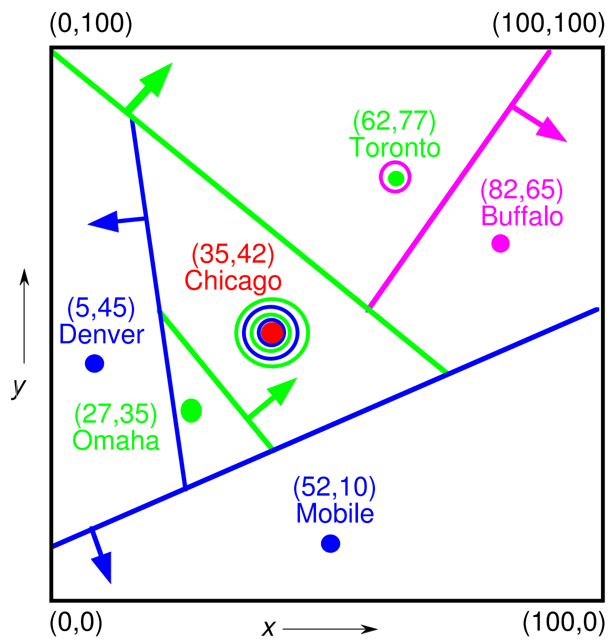
Comparison of mb-tree (BSP tree) and PR k-d tree

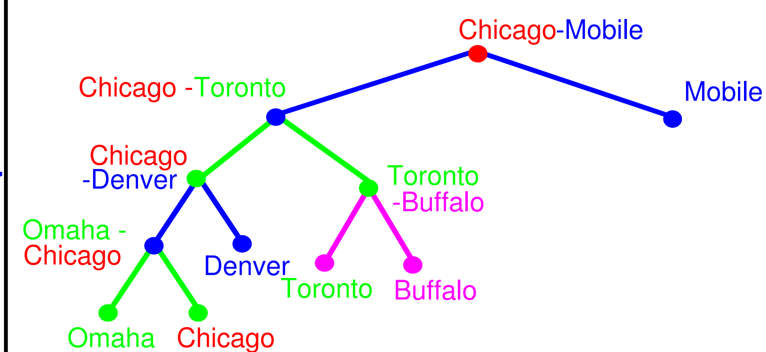
PR k-d tree





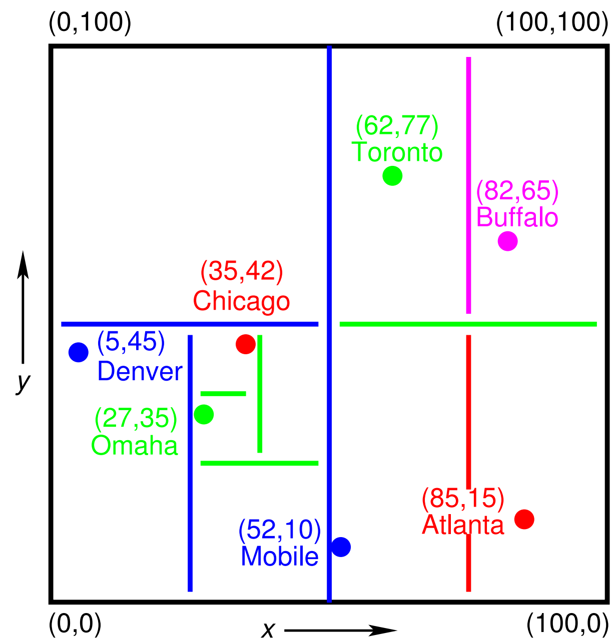
mb-tree

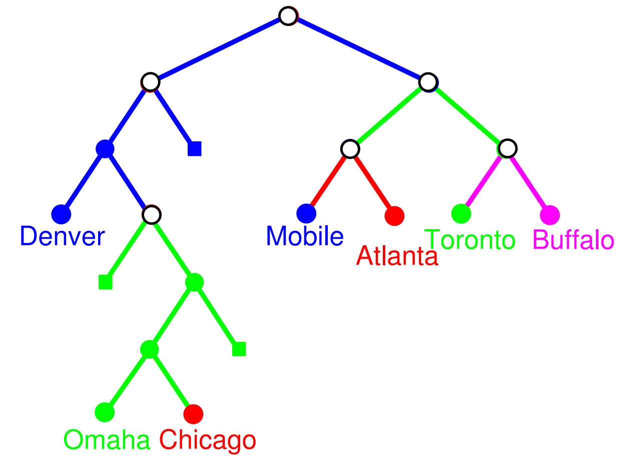




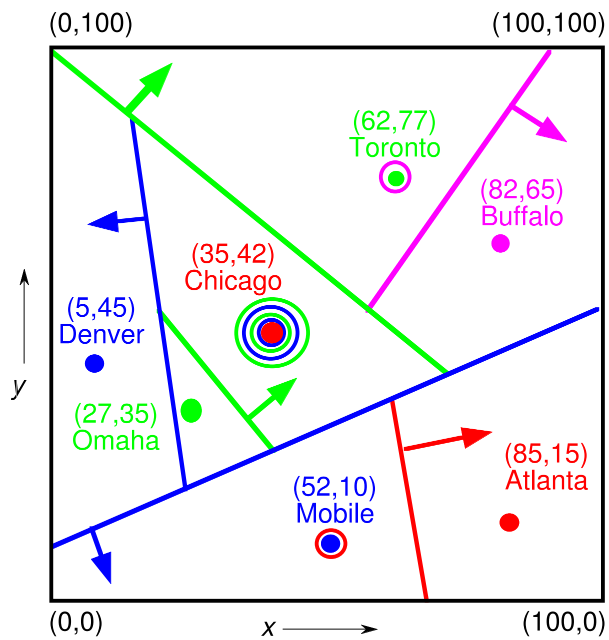
Comparison of mb-tree (BSP tree) and PR k-d tree

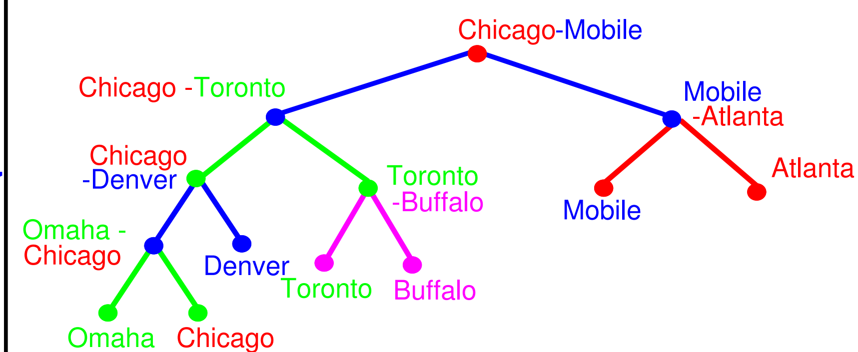
PR k-d tree





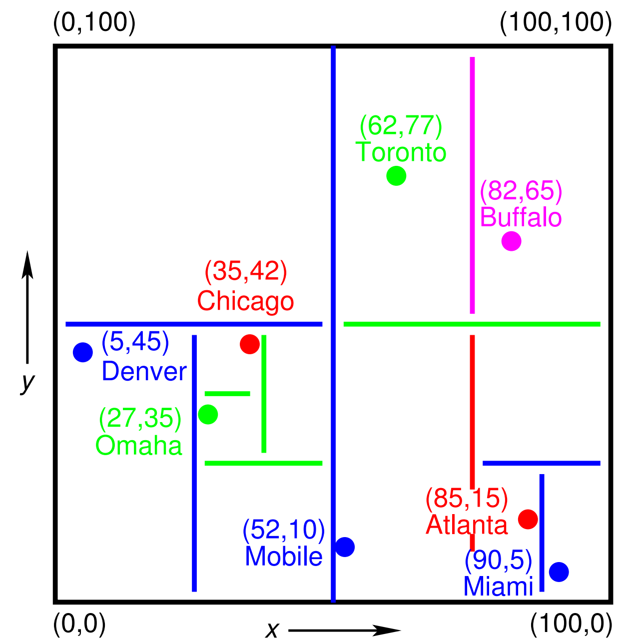
mb-tree

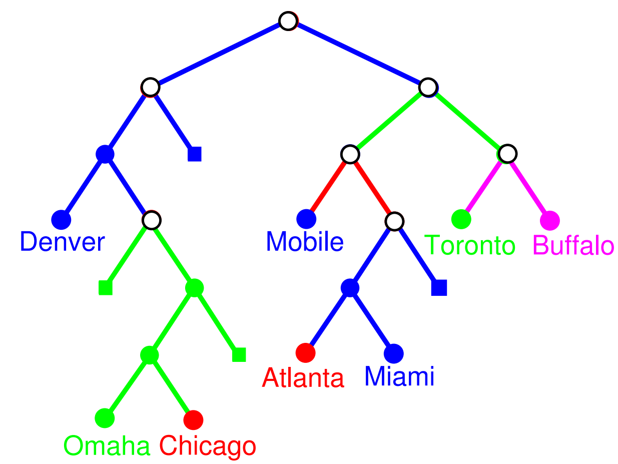




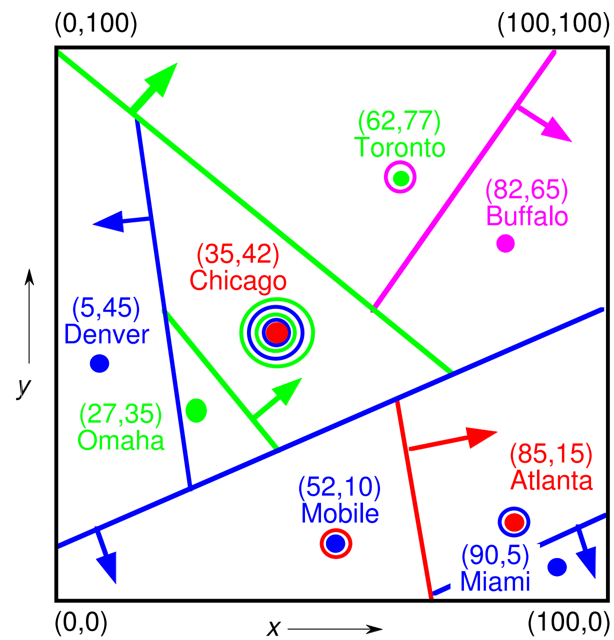
Comparison of mb-tree (BSP tree) and PR k-d tree

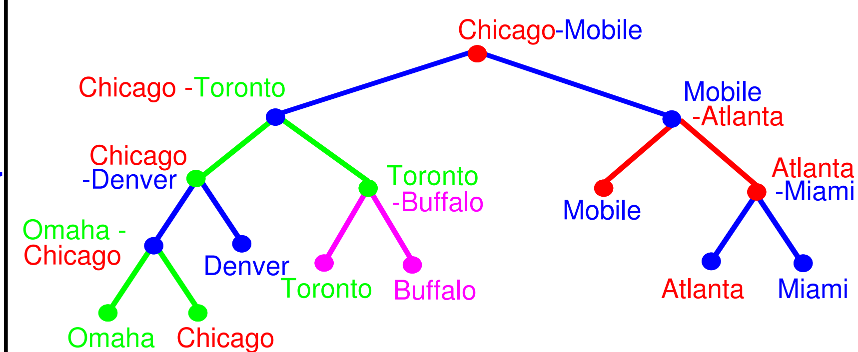
PR k-d tree





mb-tree

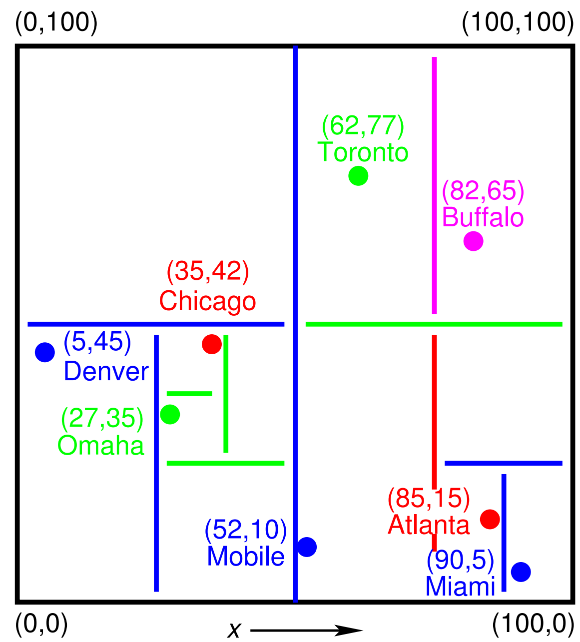


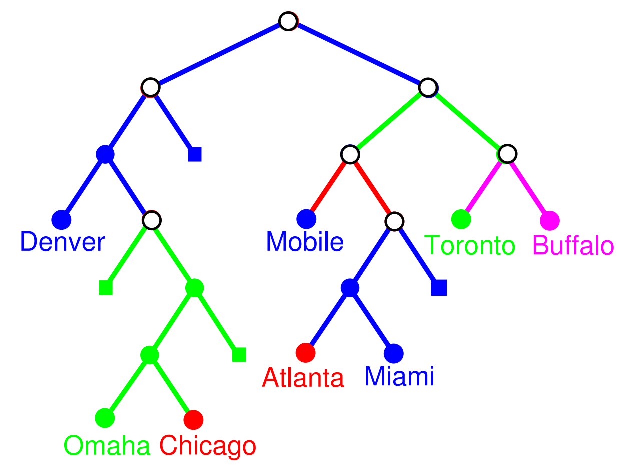


Comparison of mb-tree (BSP tree) and PR k-d tree

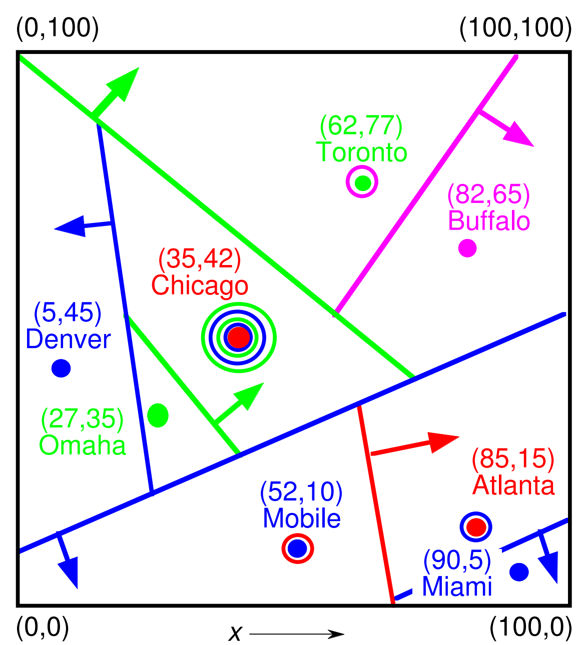
PR k-d tree

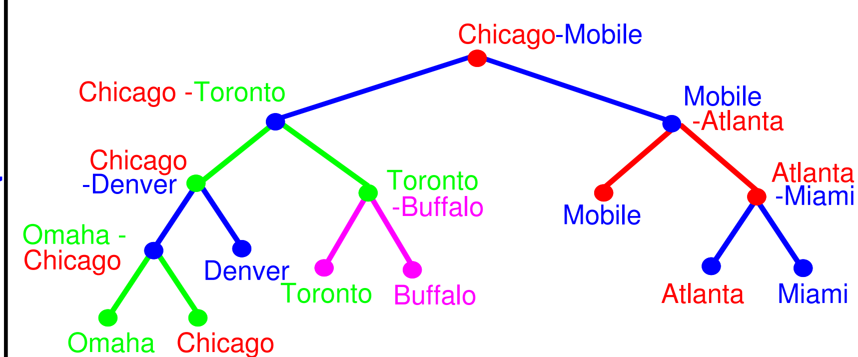
- Partition of underlying space analogous to that of BSP tree for points





mb-tree





Outline

1. Introduction
2. Points
3. Lines
4. Regions, Volumes, and Surfaces
5. Bounding Box Hierarchies
6. Rectangles
7. Surfaces and Volumes
8. Metric Data
9. Operations
10. Indexing Spatiotextual Data
11. Example system

Incremental Nearest Neighbors (Hjaltason/Samet)

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

Mechanics of Incremental Nearest Neighbor Algorithm

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

Incremental Nearest Neighbor Algorithm

Algorithm:

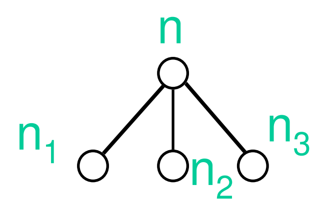
INCNEAREST(q, S, T)

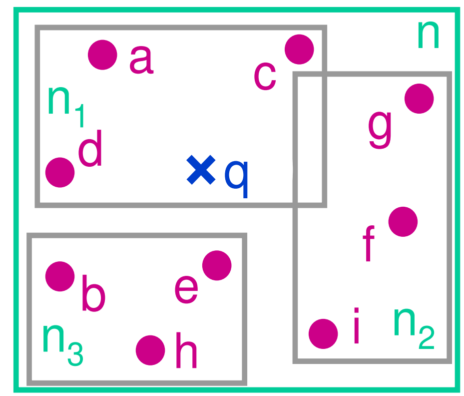
1 $Q \leftarrow$ NEWPRIORITYQUEUE()
2 $e_t \leftarrow$ root of the search hierarchy induced by q, S, and T
3 ENQUEUE(Q, e_t, 0)
4 while not ISEMPTY(Q) do
5   $e_t \leftarrow$ DEQUEUE(Q)
6   if $t = 0$ then /* e_t is an object */
7   Report e_t as the next nearest object
8   else
9   for each child element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

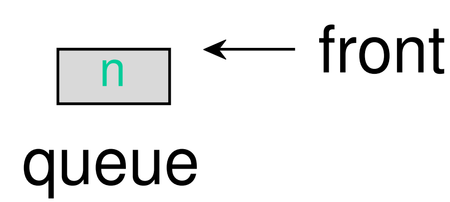
1. Lines 1-3 initialize priority queue with root
2. In main loop take element e_t closest to q off the queue
   - report e_t as next nearest object if e_t is an object
   - otherwise, insert child elements of e_t into priority queue

Example of INCNEAREST

- Initially, algorithm descends tree to leaf node containing q

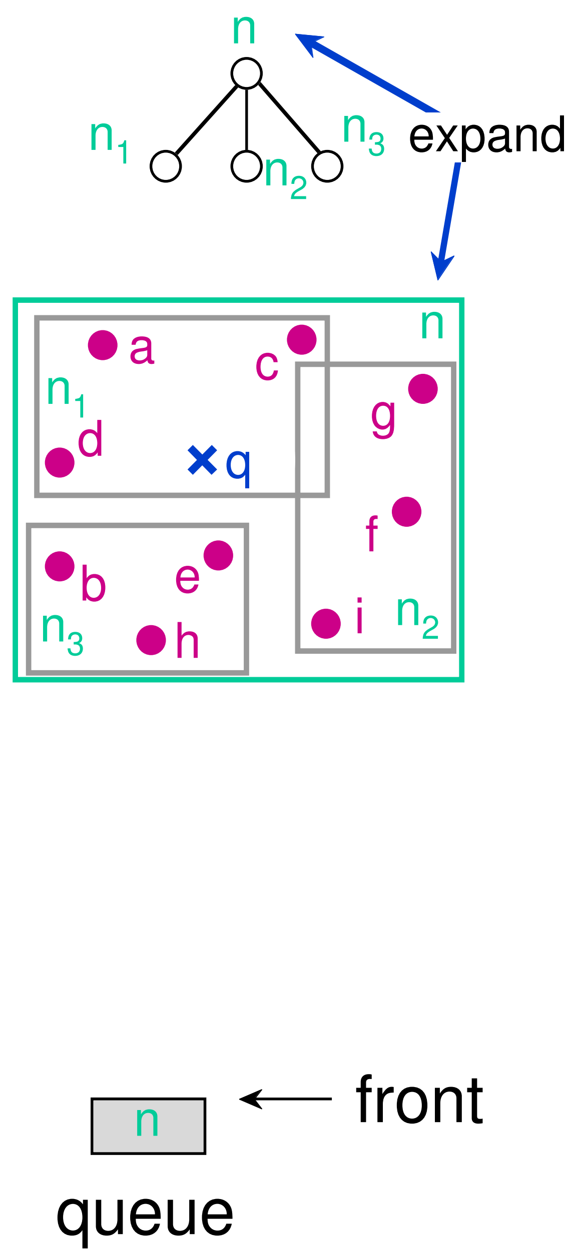






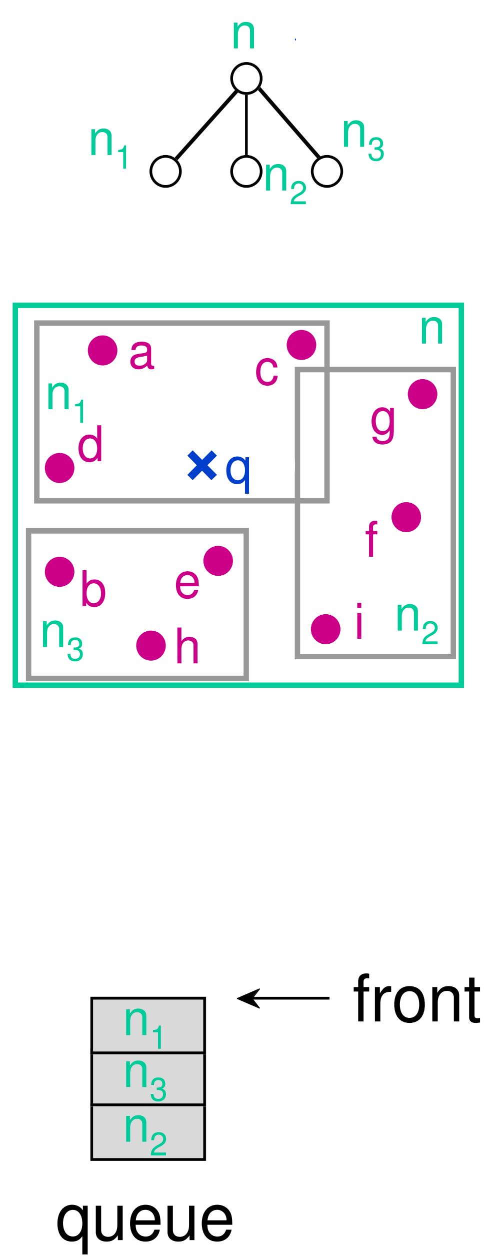
Example of INCNEAREST

- Initially, algorithm descends tree to leaf node containing q
  - expand n



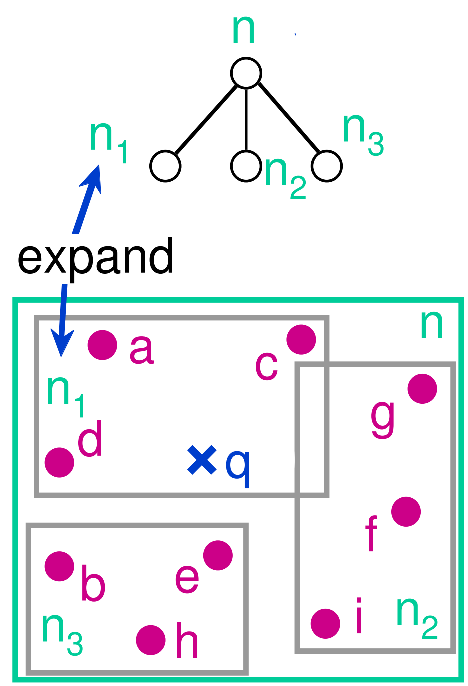
Example of INCNEAREST

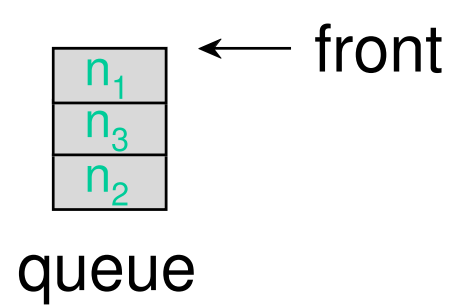
- Initially, algorithm descends tree to leaf node containing q
  - expand n



Example of INCNEAREST

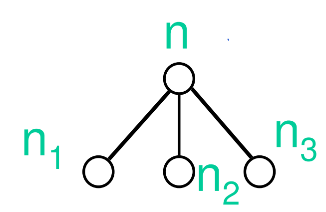
- Initially, algorithm descends tree to leaf node containing q
  - expand n
  - expand n_1

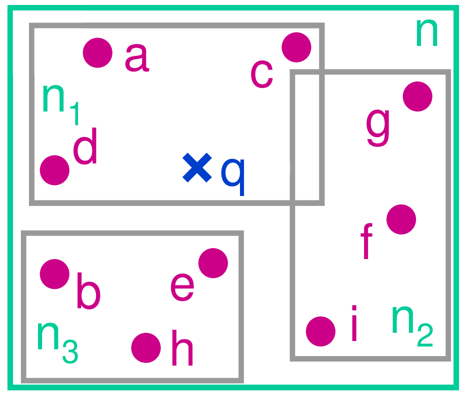


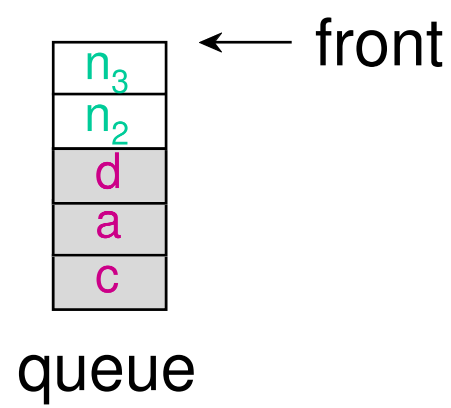


Example of INCNEAREST

- Initially, algorithm descends tree to leaf node containing q
  - expand n
  - expand n_1

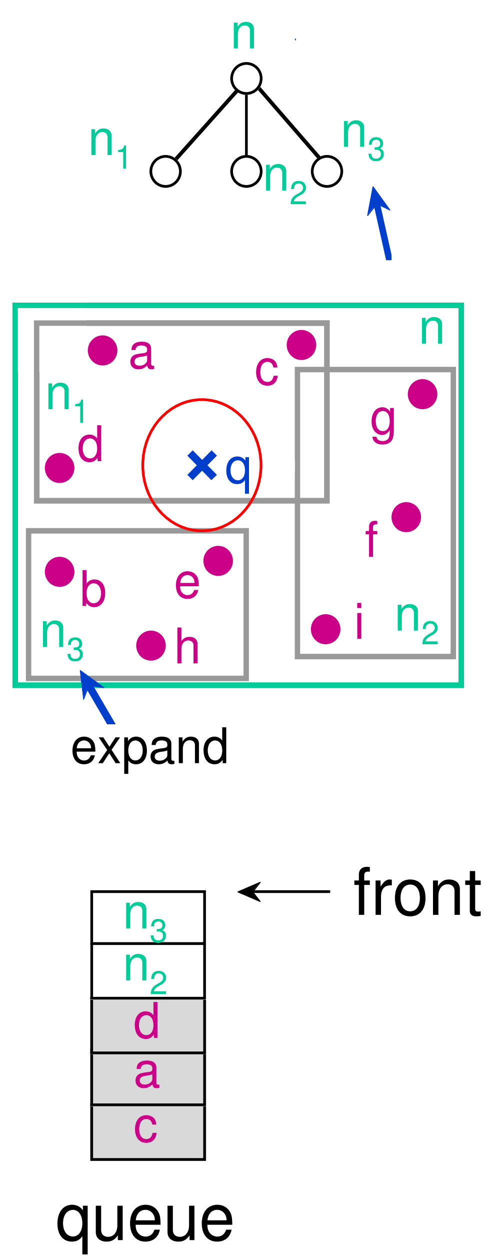






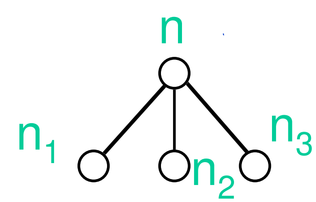
Example of IncNearest

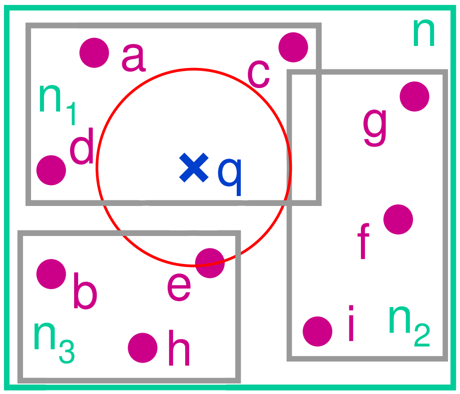
- Initially, algorithm descends tree to leaf node containing q
  - expand n
  - expand n_1
- Start growing search region
  - expand n_3

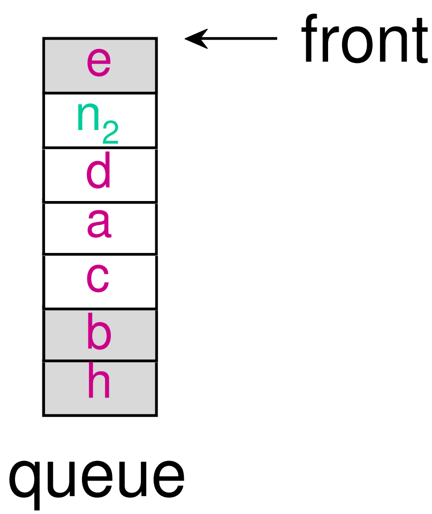


Example of INCNEAREST

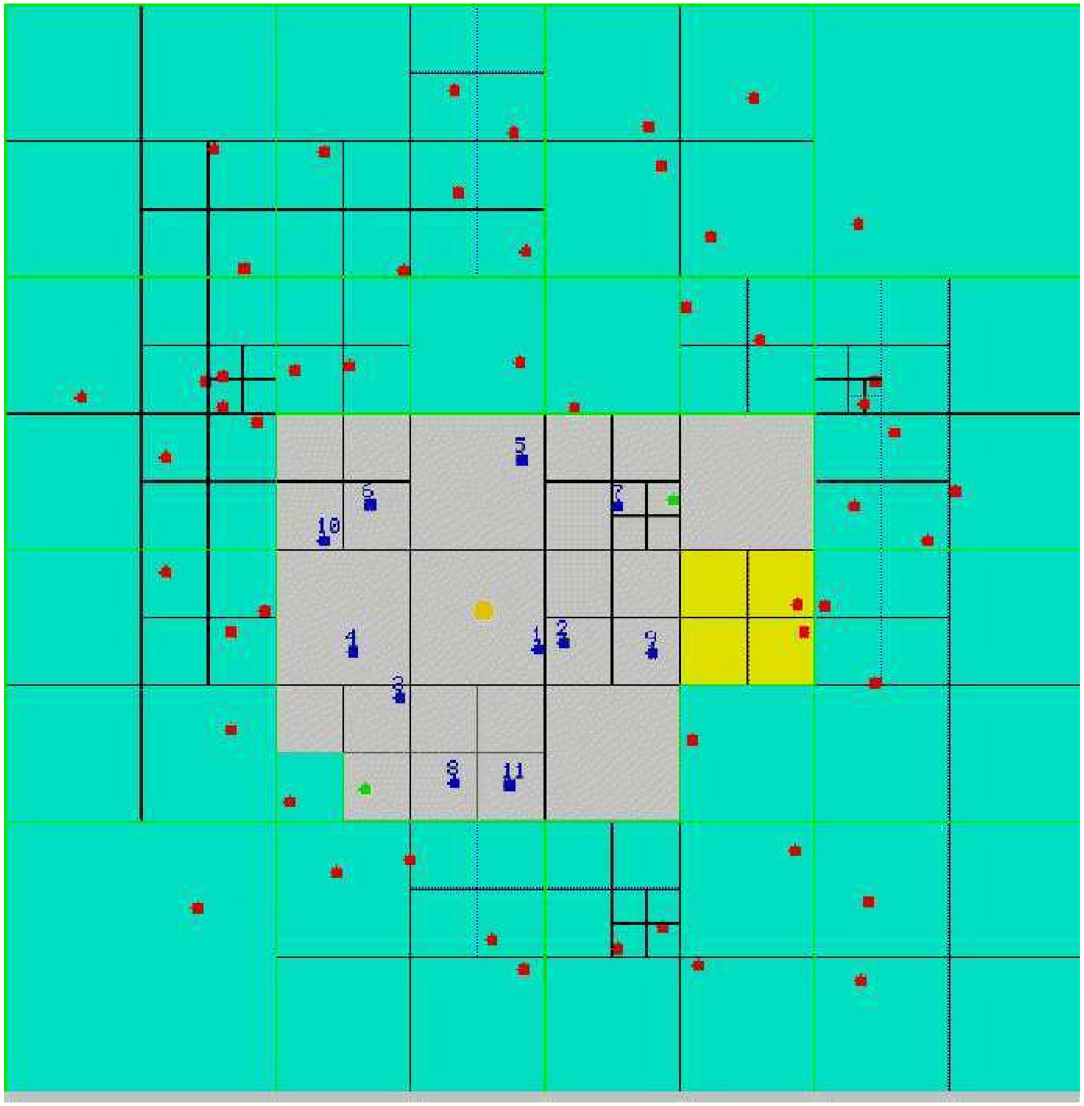
- Initially, algorithm descends tree to leaf node containing q
  - expand n
  - expand n_1
- Start growing search region
  - expand n_3
  - report e as nearest neighbor







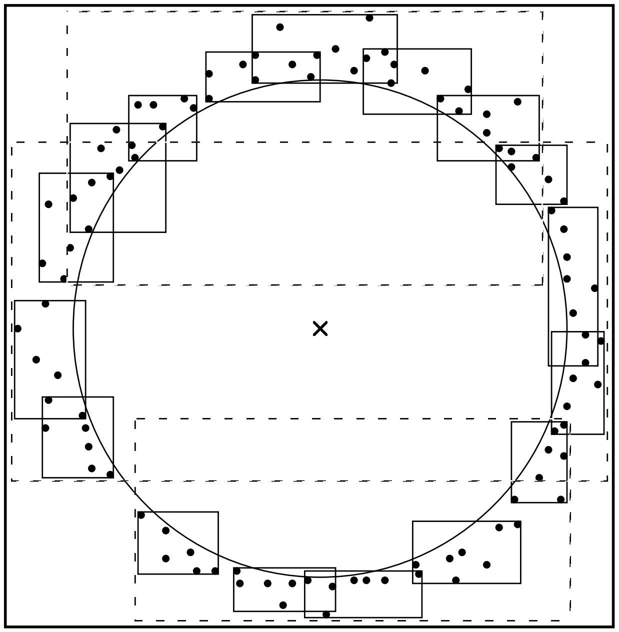
VASCO Spatial Applet



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

Complexity Analysis

- Algorithm is I/O optimal
  - no nodes outside search region are accessed
  - better pruning than branch and bound algorithm
- Observations for finding k nearest neighbors for uniformly-distributed two-dimensional points
  - expected # of points on priority queue: $O(\sqrt{k})$
  - expected # of leaf nodes intersecting search region: $O(k + \sqrt{k})$
- In worst case, priority queue will be as large as entire data set
  - e.g., when data objects are all nearly equidistant from query object
  - probability of worst case very low, as it depends on a particular configuration of both the data objects and the query object (but: curse of dimensionality!)



Key to Nearest Neighbor Finding in Spatial Networks

1. Use distance along a graph rather than “as the crow flies”
2. Precompute and store shortest paths between all vertices in network
   - Reduce cost of storing shortest paths between all pairs of N vertices from $O(N^3)$ to $O(N^{1.5})$ using path coherence of destination vertices

Key to Nearest Neighbor Finding in Spatial Networks

1. Use distance along a graph rather than “as the crow flies”
2. Precompute and store shortest paths between all vertices in network
   - Reduce cost of storing shortest paths between all pairs of N vertices from $O(N^3)$ to $O(N^{1.5})$ using path coherence of destination vertices
   - Can reduce to $O(N)$ by also using path coherence of source vertices

Key to Nearest Neighbor Finding in Spatial Networks

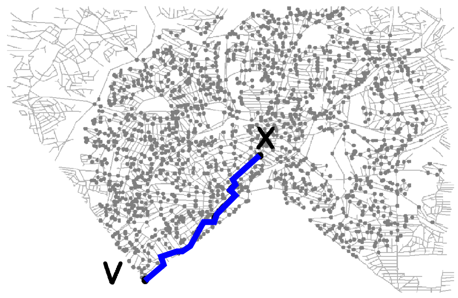
1. Use distance along a graph rather than "as the crow flies"
2. Precompute and store shortest paths between all vertices in network
   - Reduce cost of storing shortest paths between all pairs of N vertices from $O(N^3)$ to $O(N^{1.5})$ using path coherence of destination vertices
   - Can reduce to $O(N)$ by also using path coherence of source vertices

3. Decouple domain S of query objects (q) and objects from which neighbors are drawn from domain V of vertices of network
   - Implies no need to recompute shortest paths each time q or S change

Key to Nearest Neighbor Finding in Spatial Networks

1. Use distance along a graph rather than "as the crow flies"
2. Precompute and store shortest paths between all vertices in network
   - Reduce cost of storing shortest paths between all pairs of N vertices from $O(N^3)$ to $O(N^{1.5})$ using path coherence of destination vertices
   - Can reduce to $O(N)$ by also using path coherence of source vertices

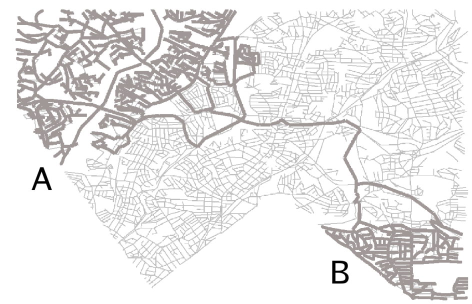
3. Decouple domain S of query objects (q) and objects from which neighbors are drawn from domain V of vertices of network
   - Implies no need to recompute shortest paths each time q or S change



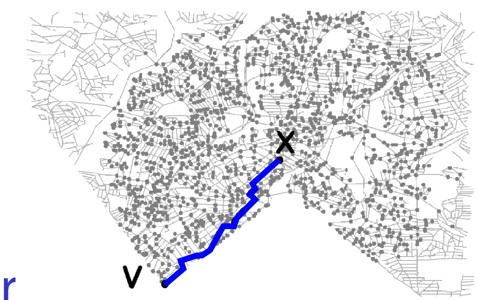
4. Avoids Dijkstra's algorithm which visits too many vertices
   - Ex: Dijkstra's algorithm visits 3191 out of the 4233 ver tices in network to identify a 76 edge path from X to V

Key to Nearest Neighbor Finding in Spatial Networks

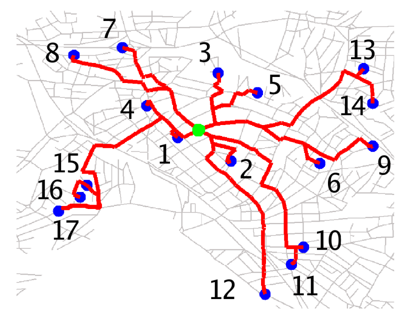
1. Use distance along a graph rather than "as the crow flies"
2. Precompute and store shortest paths between all vertices in network
   - Reduce cost of storing shortest paths between all pairs of N vertices from $O(N^3)$ to $O(N^{1.5})$ using path coherence of destination vertices
   - Can reduce to $O(N)$ by also using path coherence of source vertices



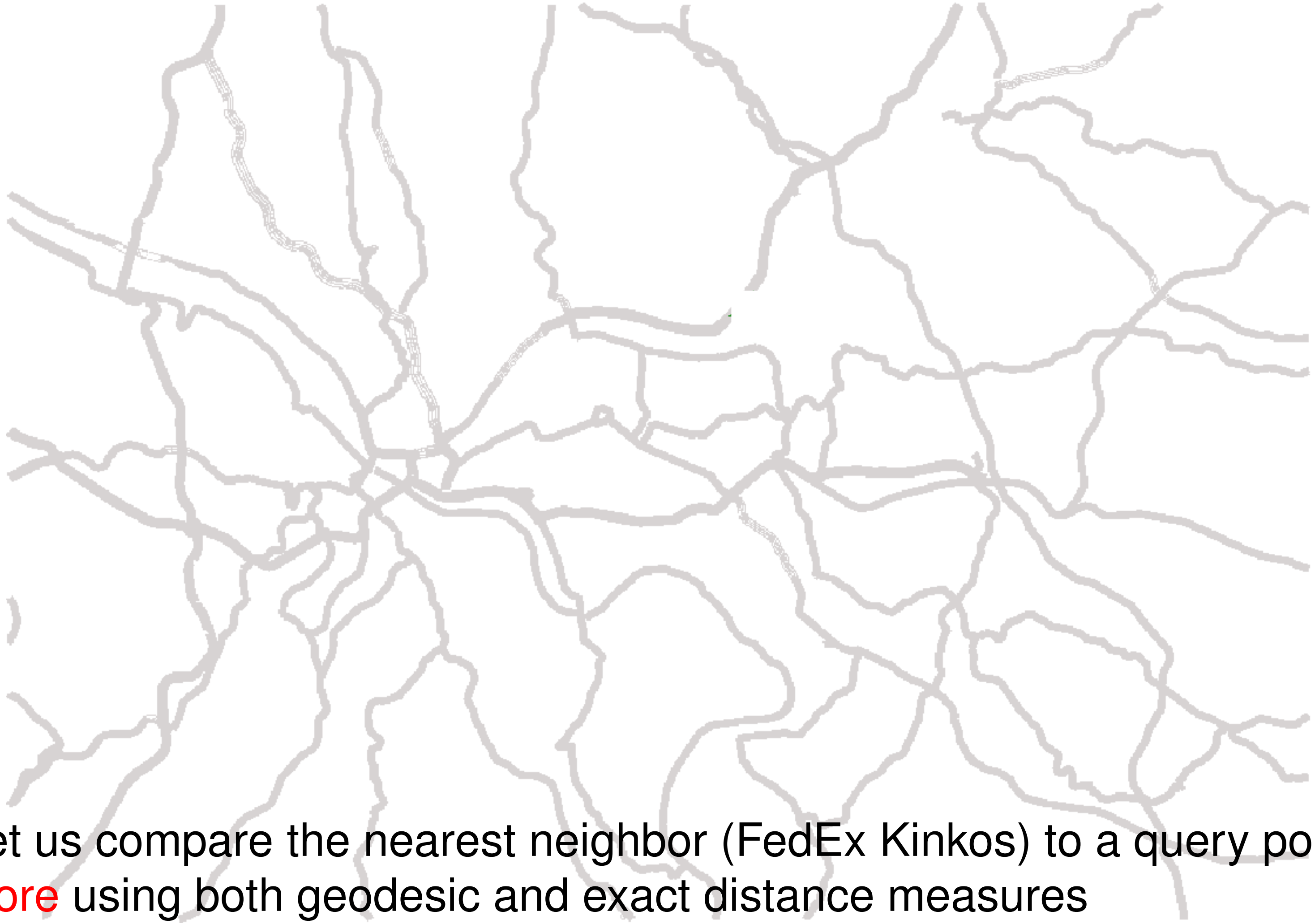
3. Decouple domain S of query objects (q) and objects from which neighbors are drawn from domain V of vertices of network
   - Implies no need to recompute shortest paths each time q or S change



4. Avoids Dijkstra's algorithm which visits too many vertices
   - Ex: Dijkstra's algorithm visits 3191 out of the 4233 ver tices in network to identify a 76 edge path from X to V
5. Instead, only visit vertices on shortest paths to nearest neighbors

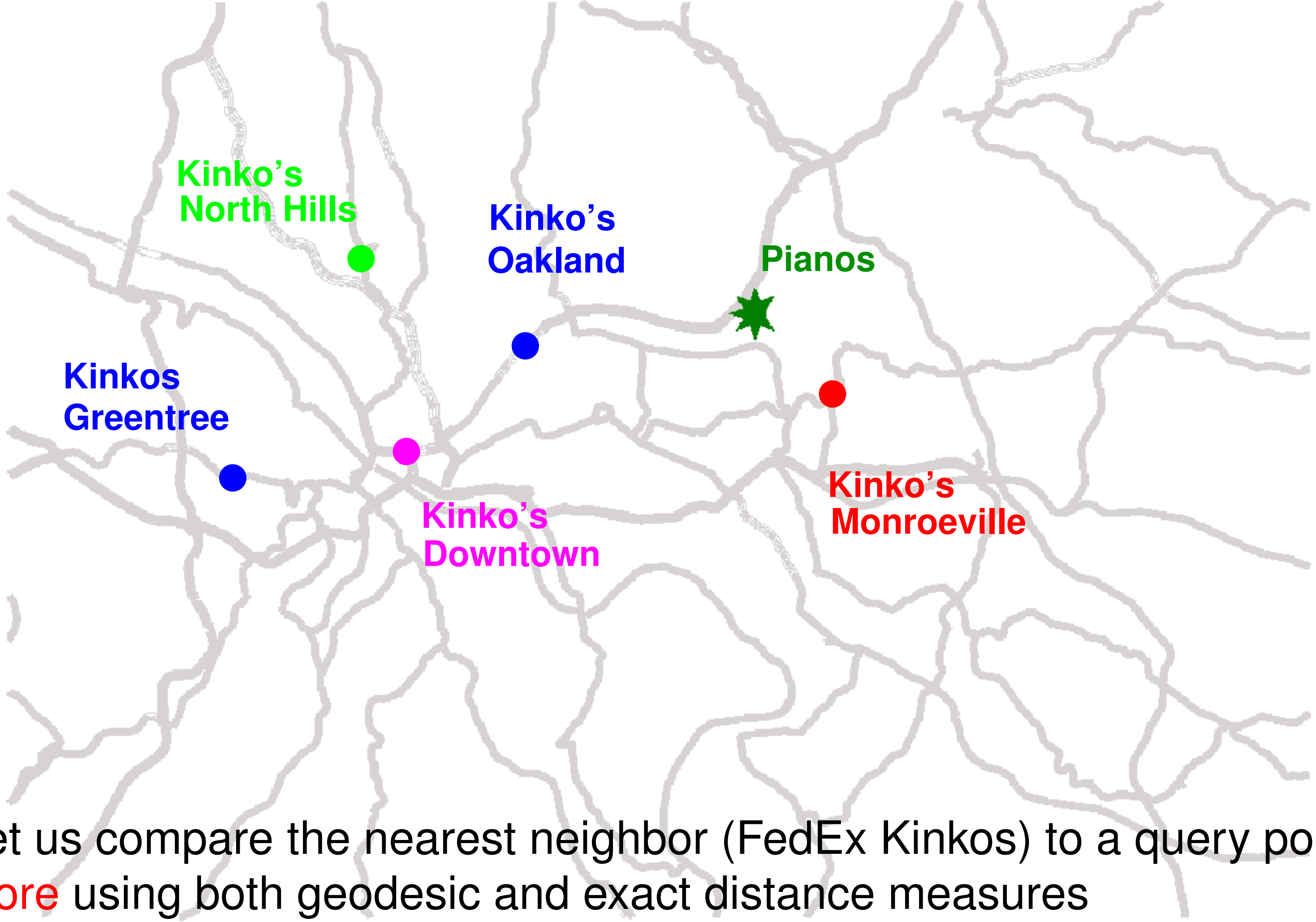


Application – Find the closest Kinko's



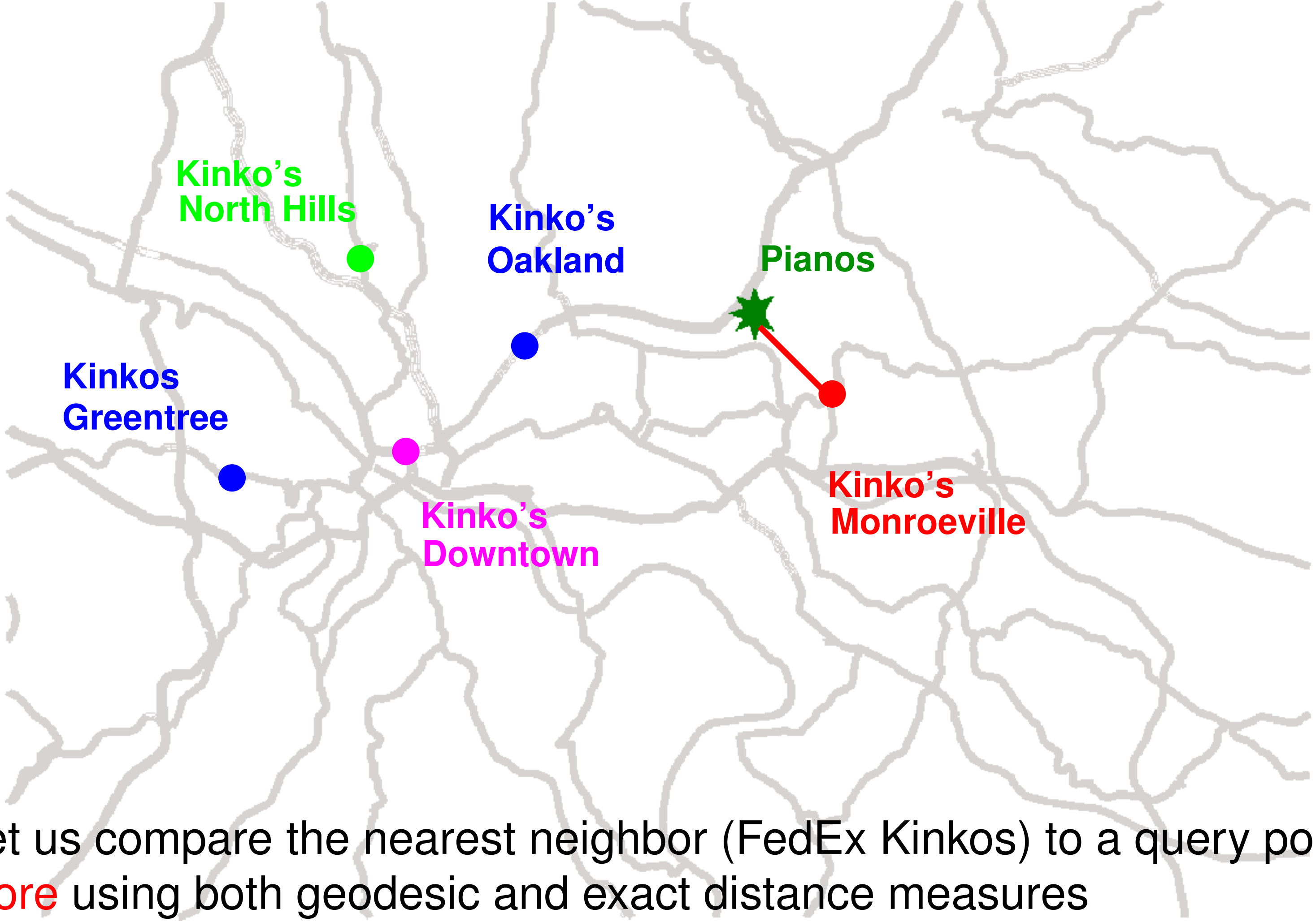
- Let us compare the nearest neighbor (FedEx Kinkos) to a query point Piano store using both geodesic and exact distance measures

Application – Find the closest Kinko's



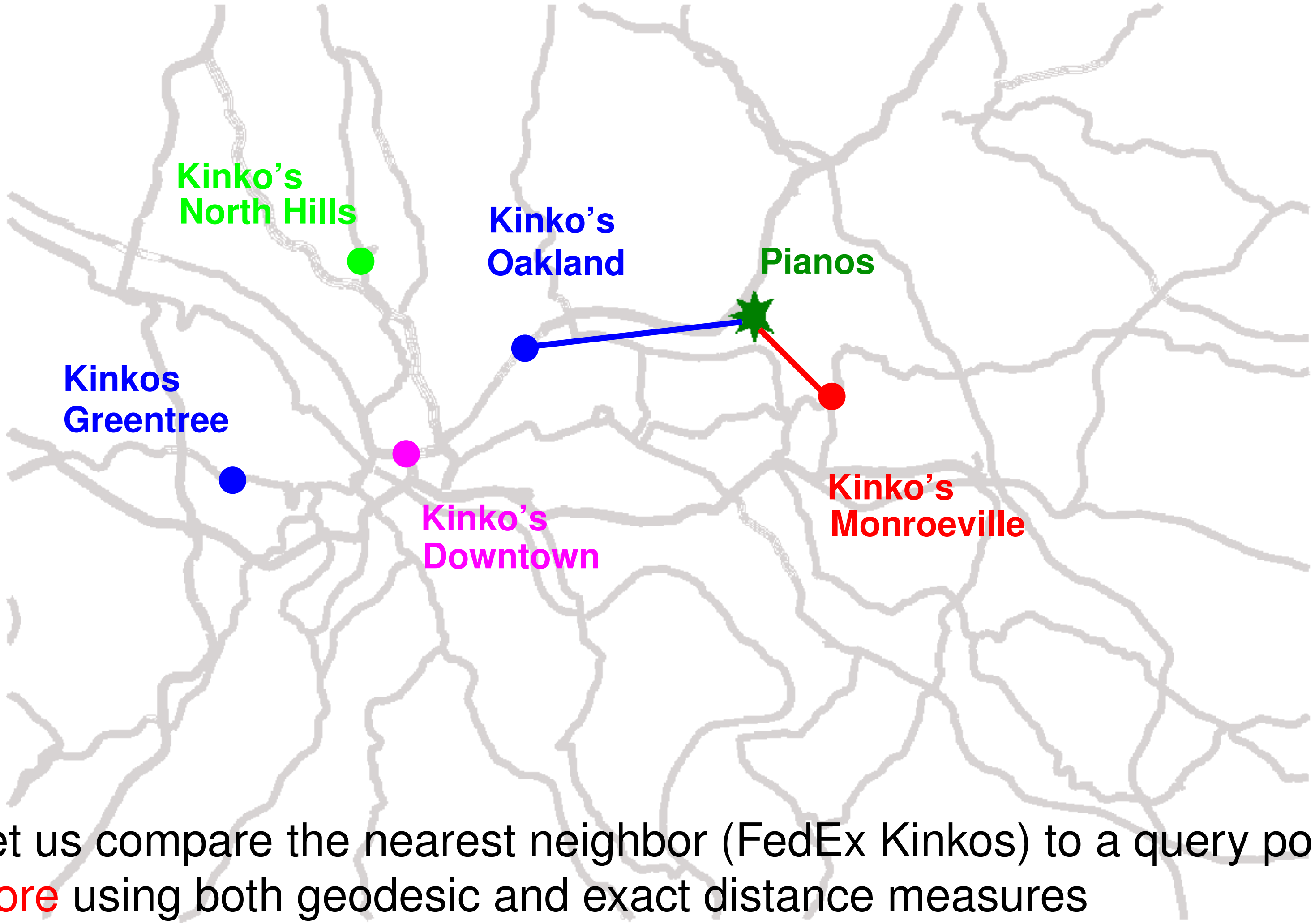
- Let us compare the nearest neighbor (FedEx Kinkos) to a query point Piano store using both geodesic and exact distance measures

Application – Find the closest Kinko’s



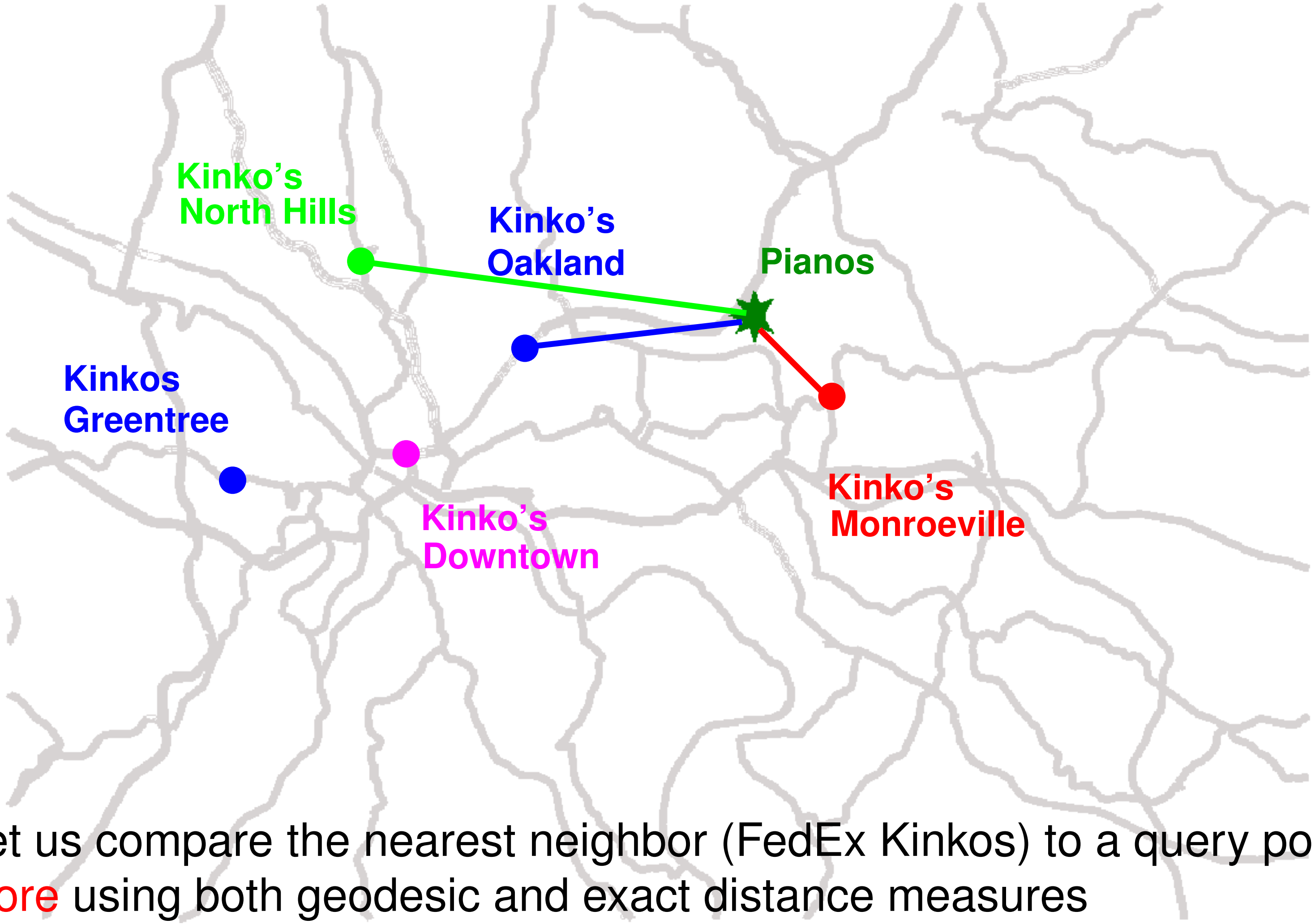
- Let us compare the nearest neighbor (FedEx Kinkos) to a query point Piano store using both geodesic and exact distance measures
  - geodesic ordering M

Application – Find the closest Kinko's



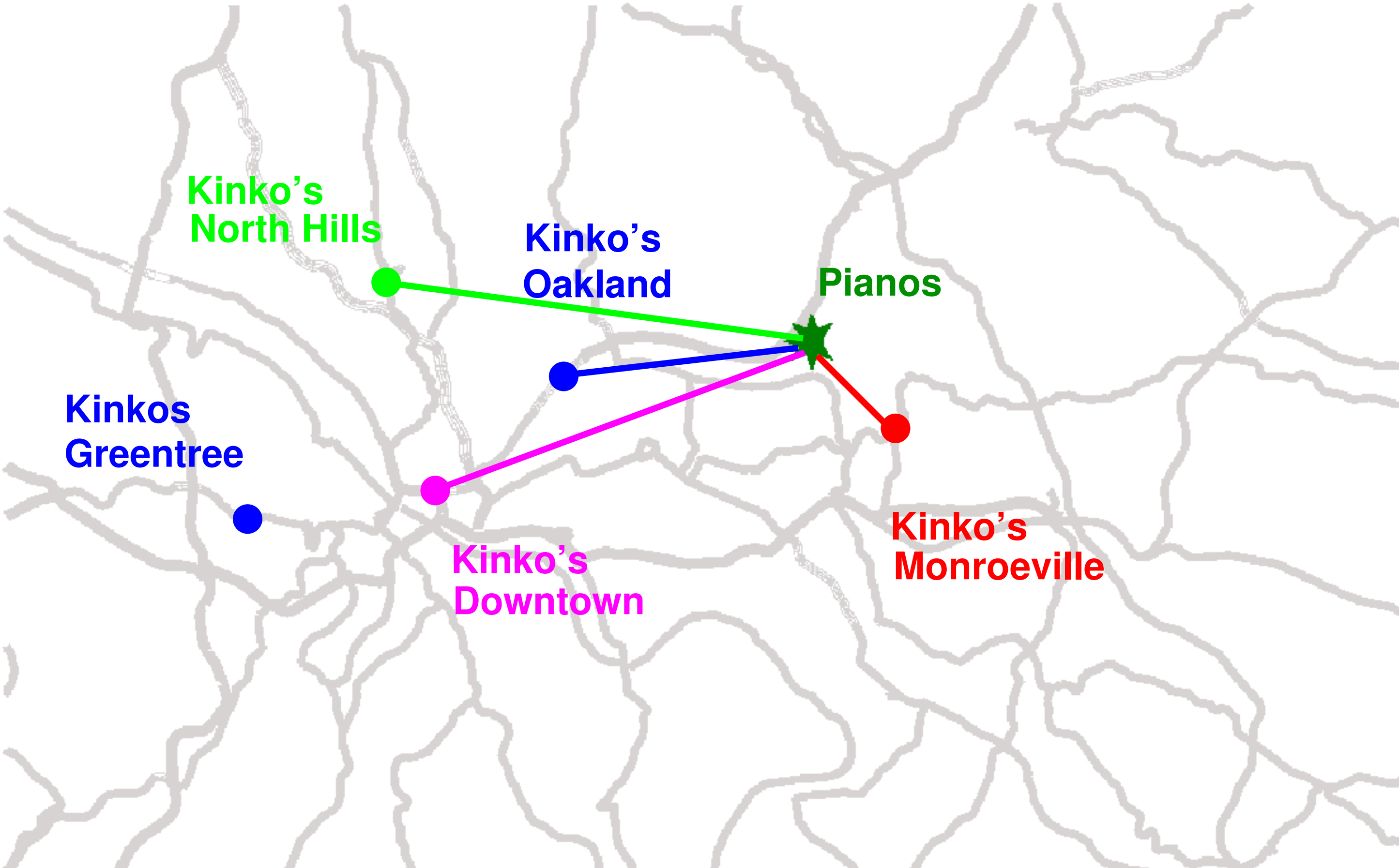
- Let us compare the nearest neighbor (FedEx Kinkos) to a query point Piano store using both geodesic and exact distance measures
  - geodesic ordering M O

Application – Find the closest Kinko's



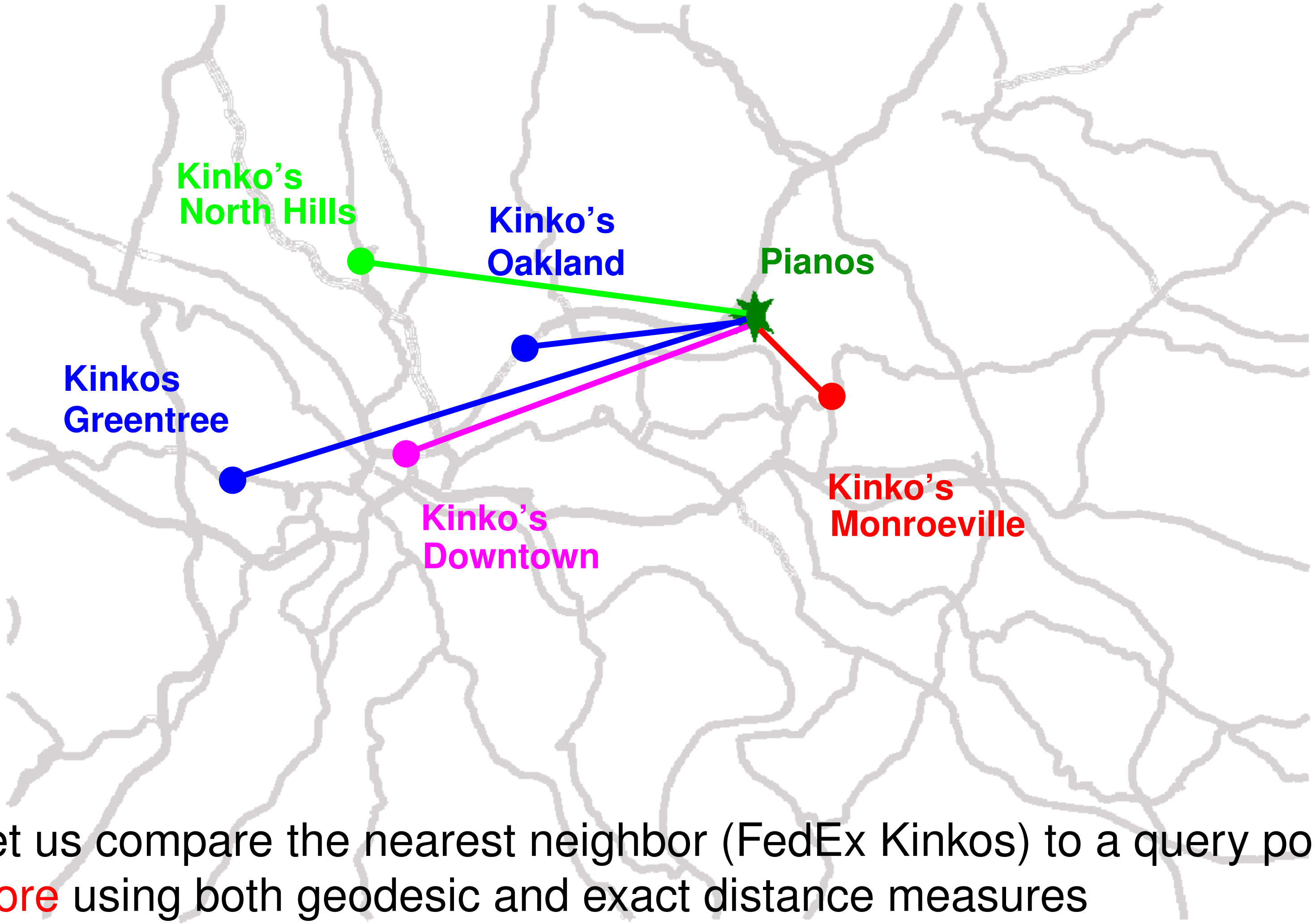
- Let us compare the nearest neighbor (FedEx Kinkos) to a query point Piano store using both geodesic and exact distance measures
  - geodesic ordering M O N

Application – Find the closest Kinko's



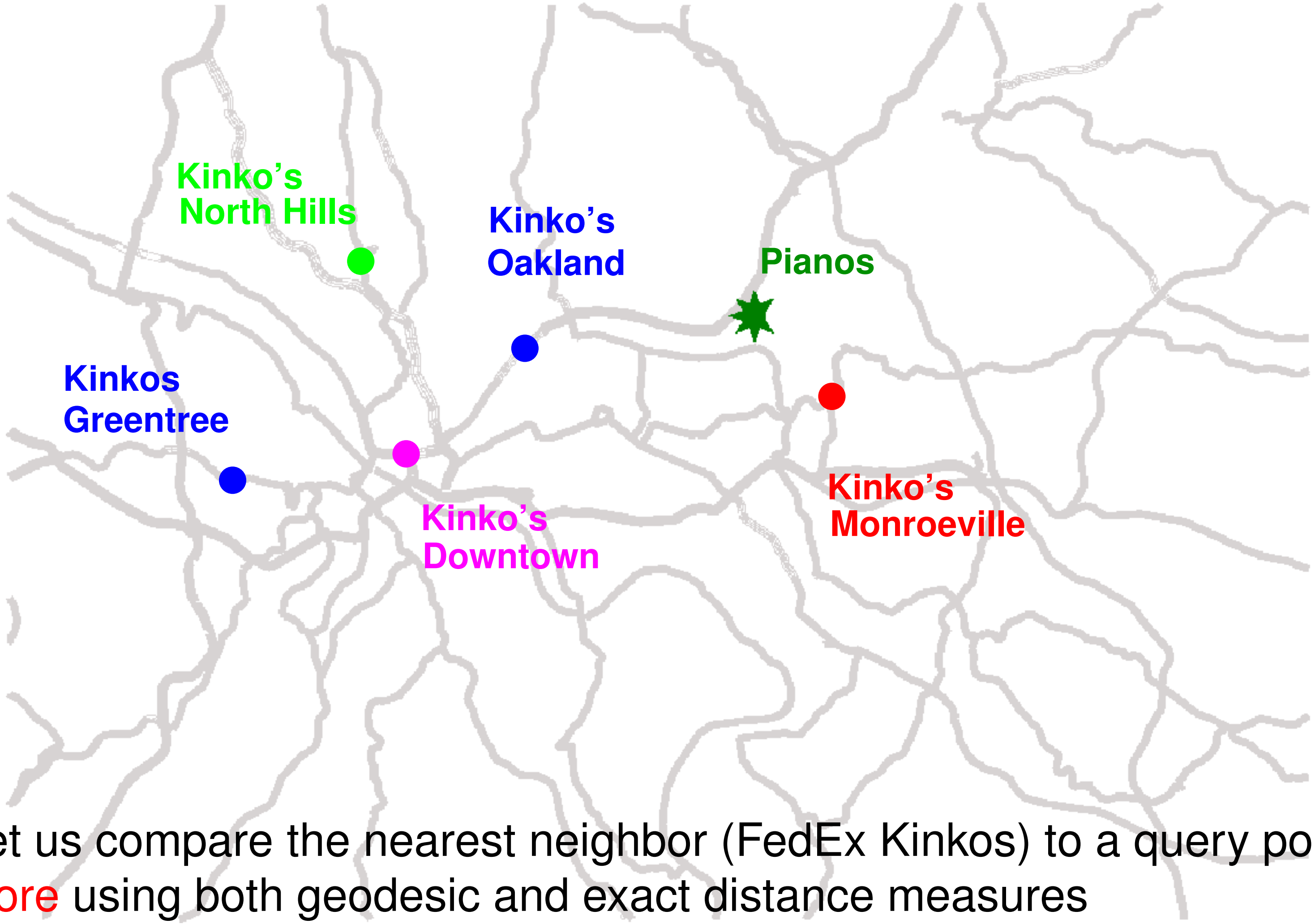
- Let us compare the nearest neighbor (FedEx Kinkos) to a query point Piano store using both geodesic and exact distance measures
  - geodesic ordering M O N D

Application – Find the closest Kinko's



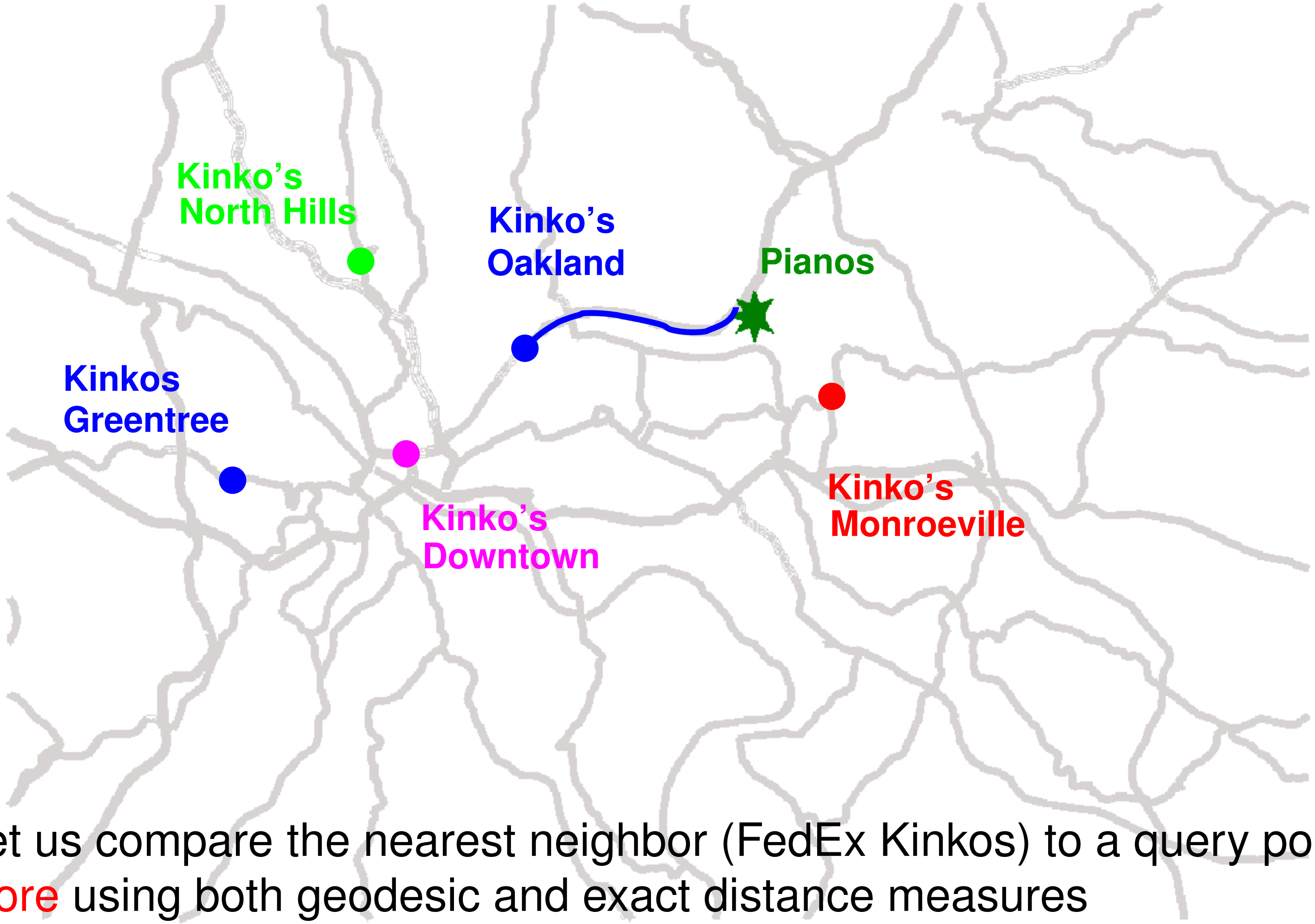
- Let us compare the nearest neighbor (FedEx Kinkos) to a query point Piano store using both geodesic and exact distance measures
  - geodesic ordering M O N D G

Application – Find the closest Kinko's



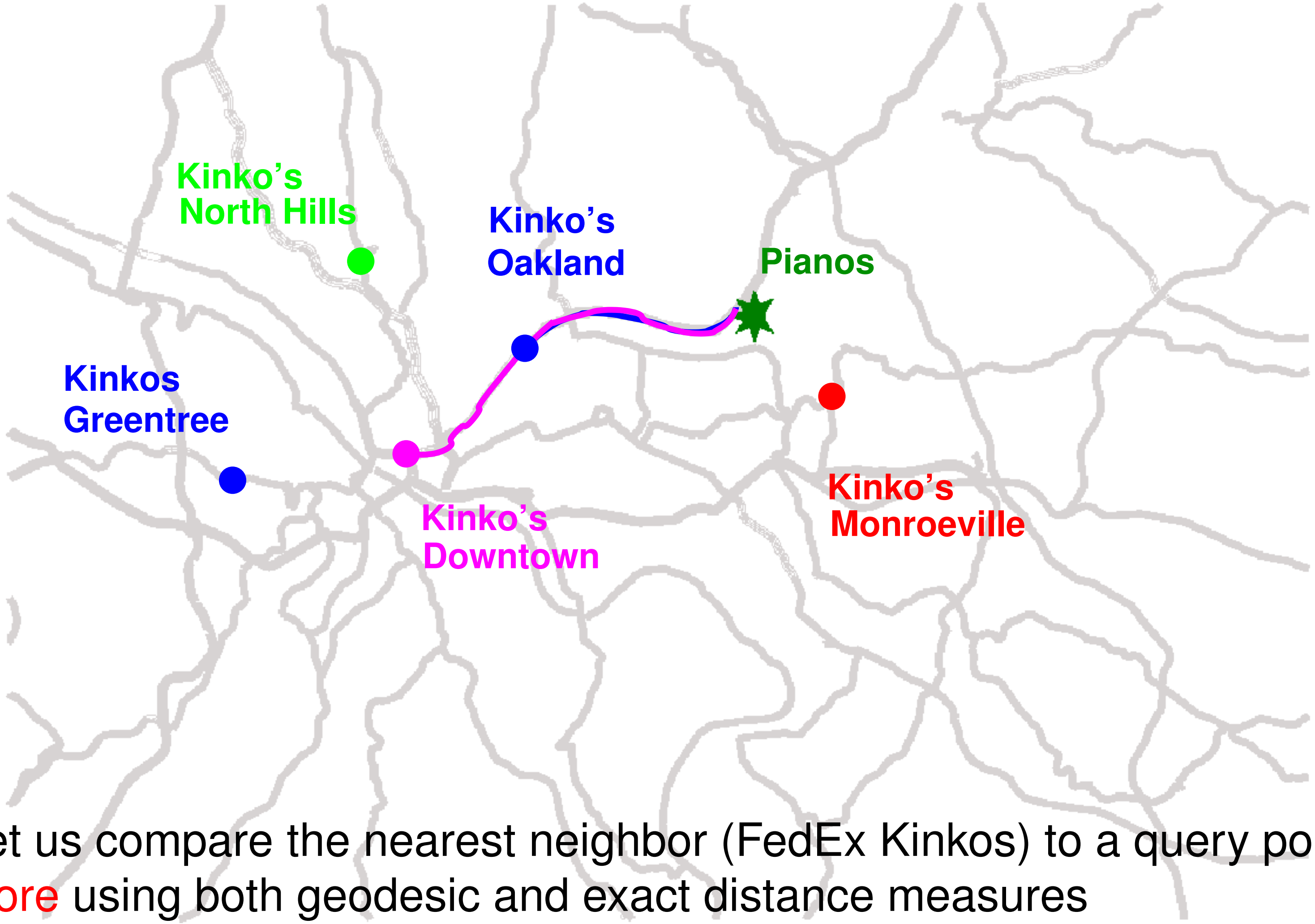
- Let us compare the nearest neighbor (FedEx Kinkos) to a query point Piano store using both geodesic and exact distance measures
  - geodesic ordering M O N D G
  - network distance ordering

Application – Find the closest Kinko's



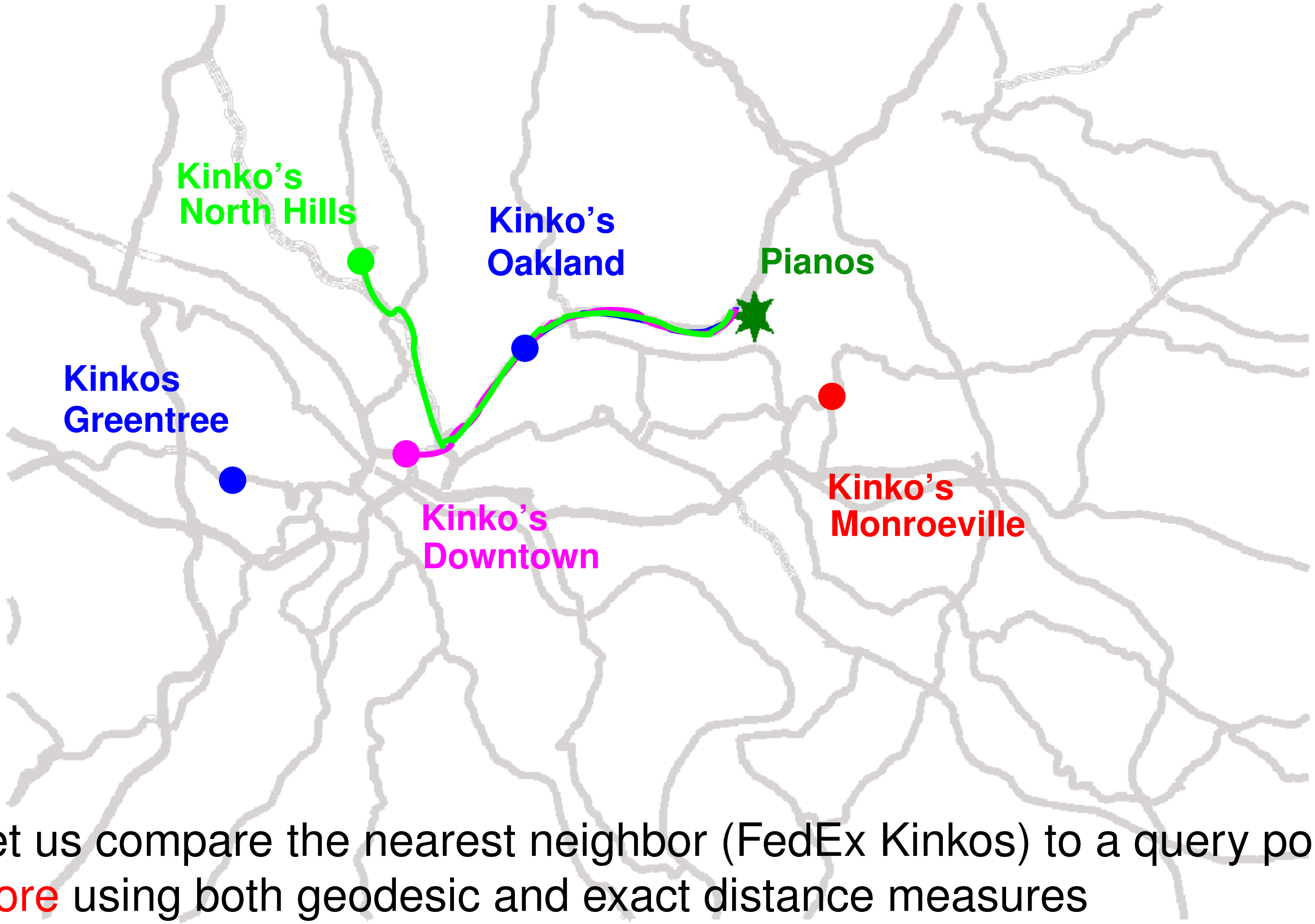
- Let us compare the nearest neighbor (FedEx Kinkos) to a query point Piano store using both geodesic and exact distance measures
  - geodesic ordering M O N D G
  - network distance ordering O

Application – Find the closest Kinko’s



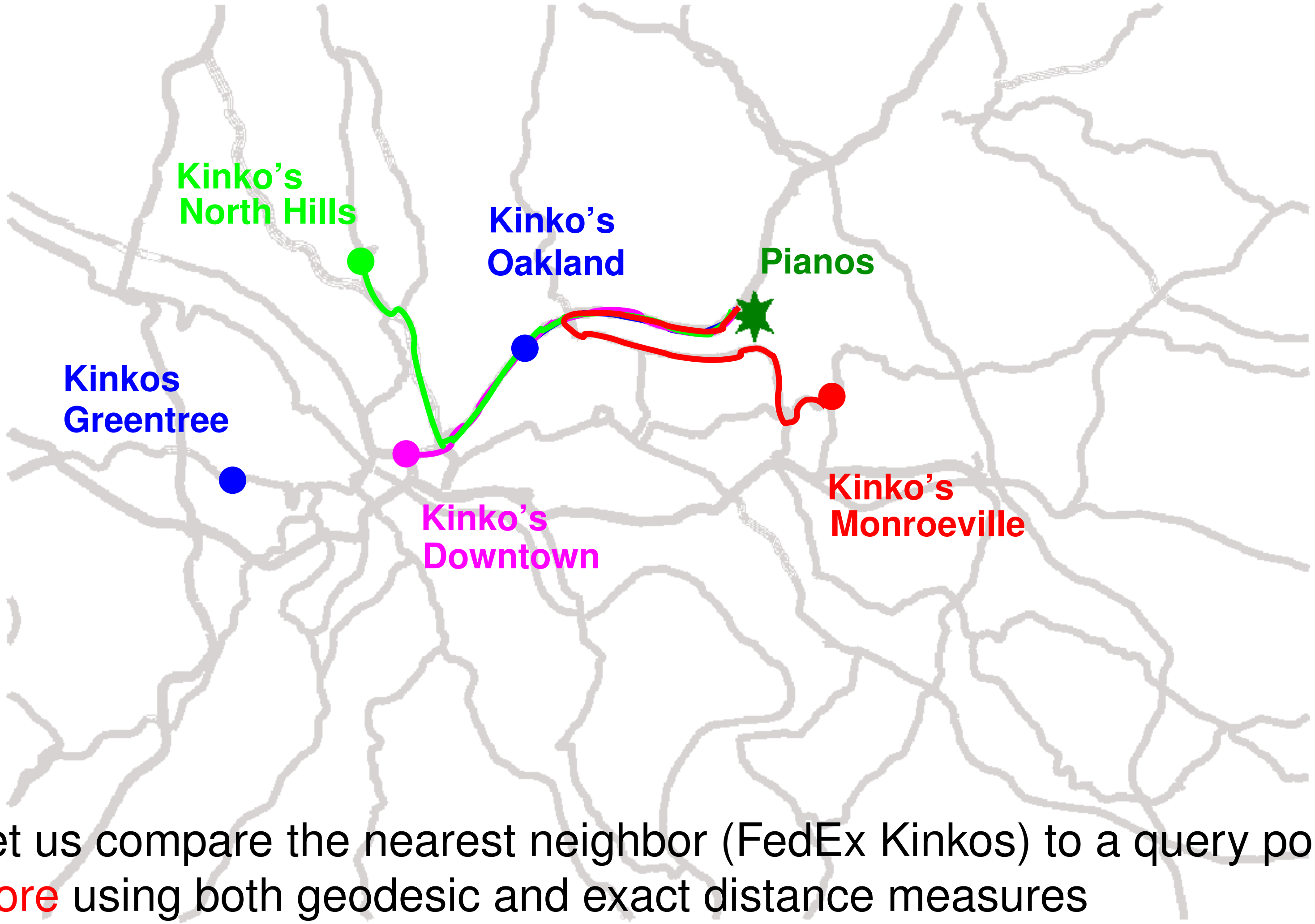
- Let us compare the nearest neighbor (FedEx Kinkos) to a query point Piano store using both geodesic and exact distance measures
  - geodesic ordering M O N D G
  - network distance ordering O D

Application – Find the closest Kinko's



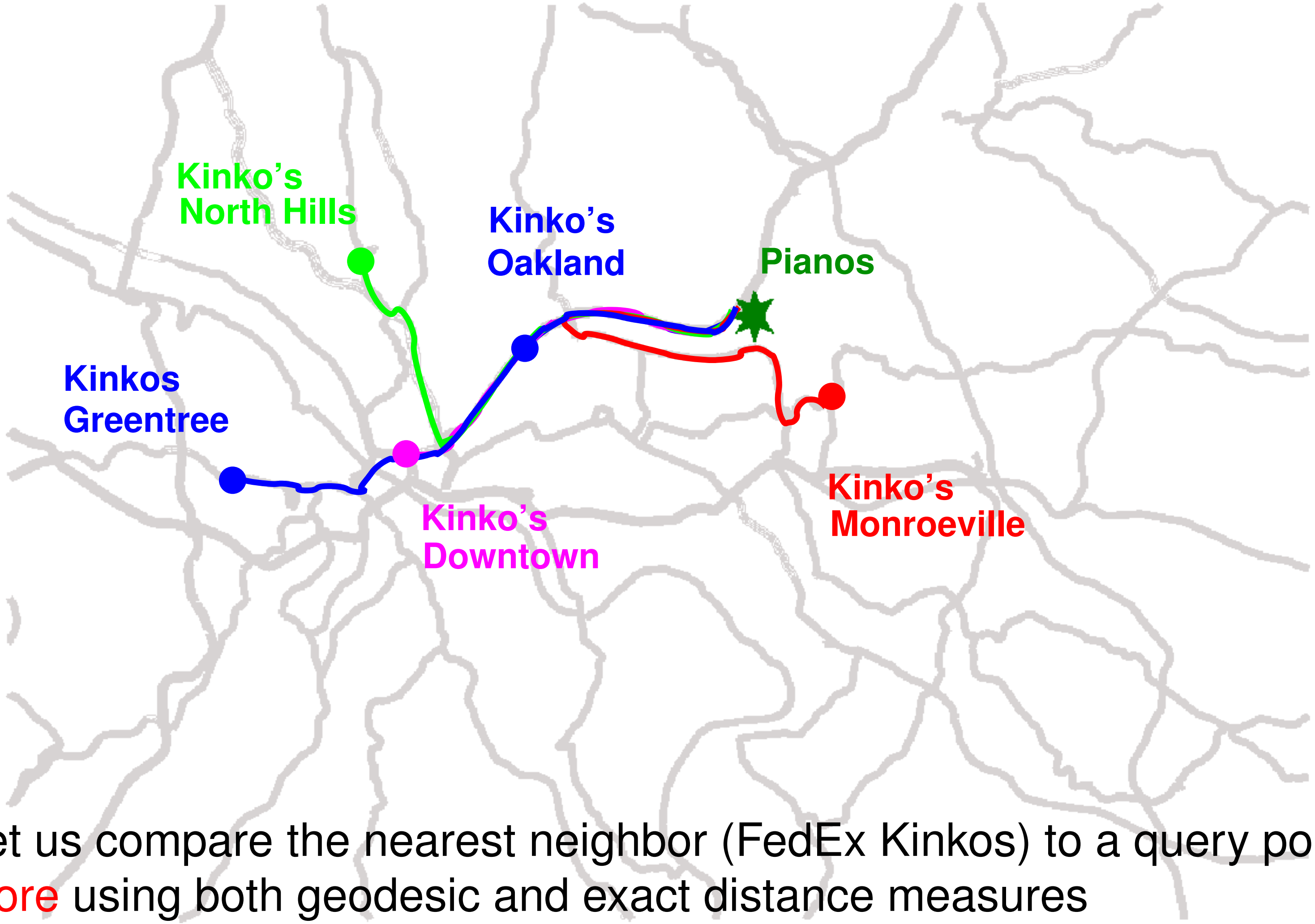
- Let us compare the nearest neighbor (FedEx Kinkos) to a query point Piano store using both geodesic and exact distance measures
  - geodesic ordering M O N D G
  - network distance ordering O D N

Application – Find the closest Kinko's



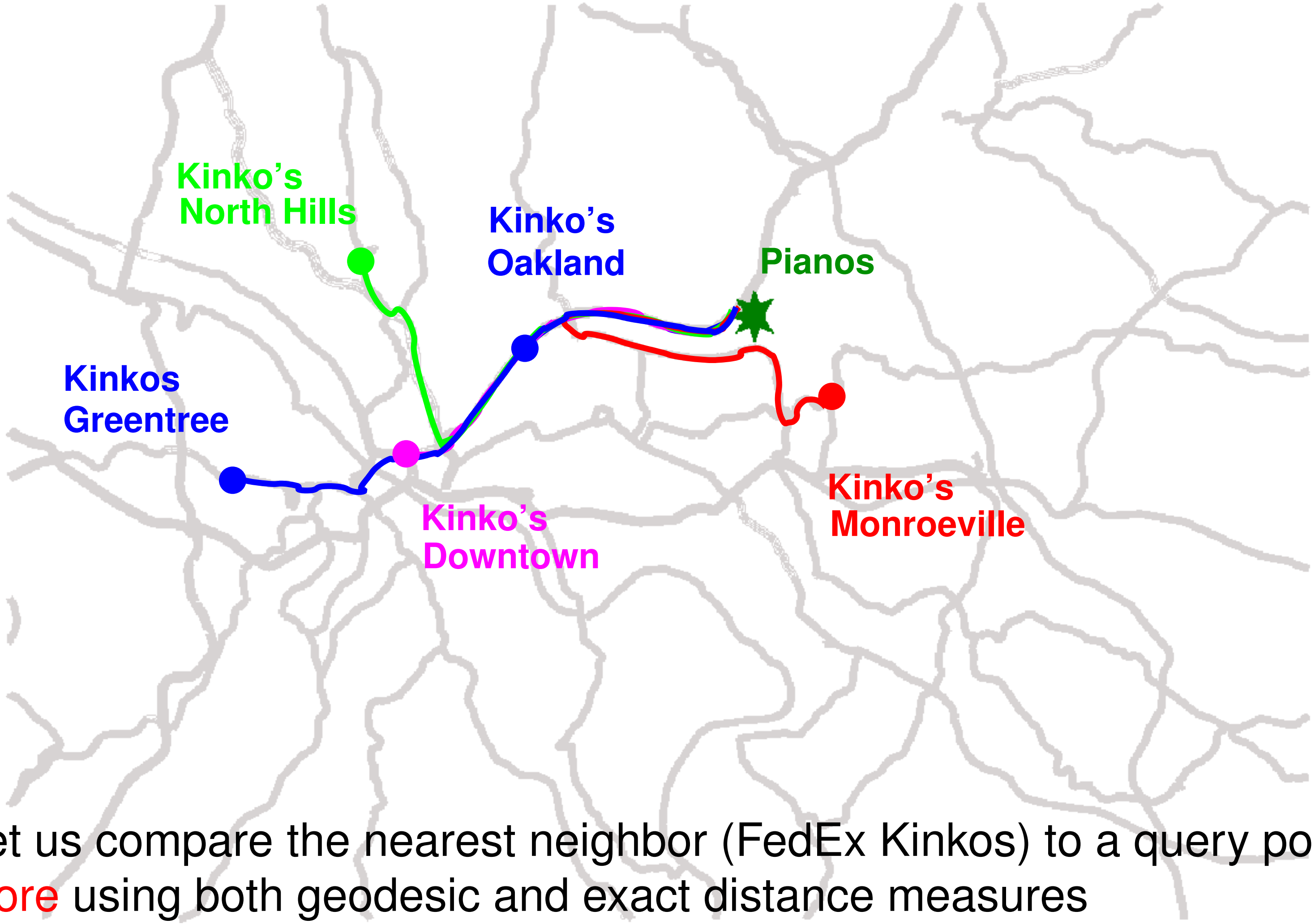
- Let us compare the nearest neighbor (FedEx Kinkos) to a query point Piano store using both geodesic and exact distance measures
  - geodesic ordering M O N D G
  - network distance ordering O D N M

Application – Find the closest Kinko's



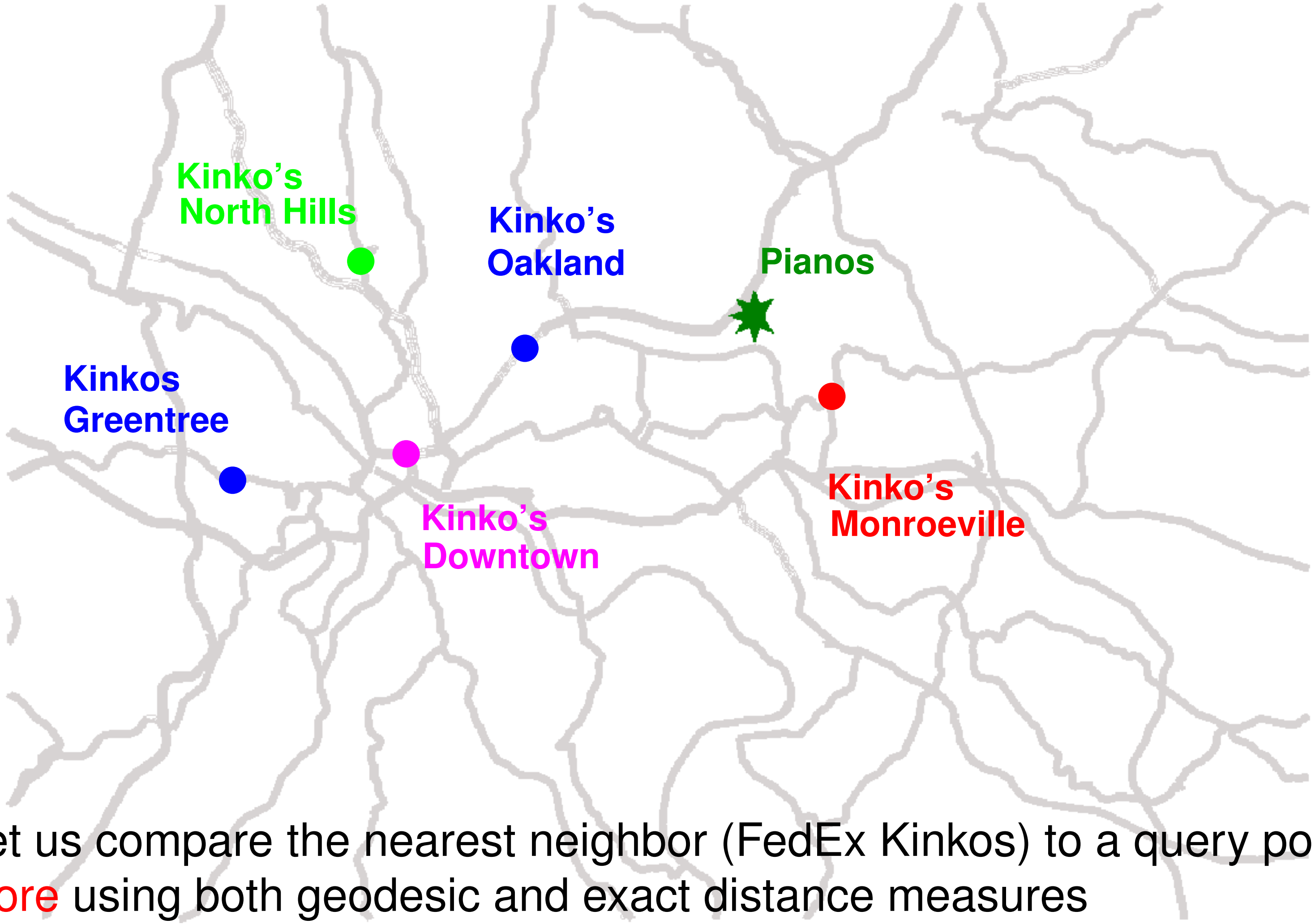
- Let us compare the nearest neighbor (FedEx Kinkos) to a query point Piano store using both geodesic and exact distance measures
  - geodesic ordering M O N D G
  - network distance ordering O D N M G

Application – Find the closest Kinko's



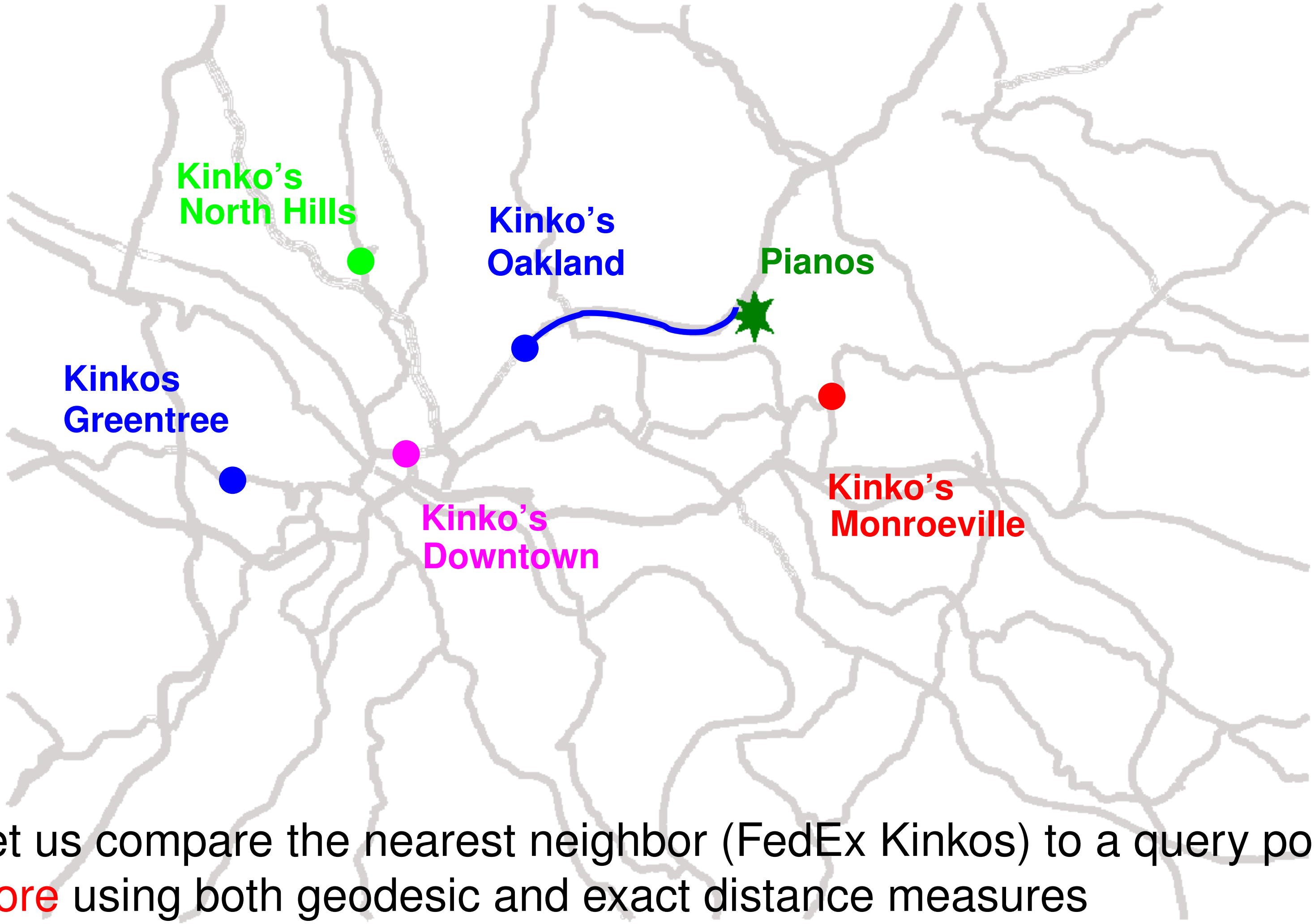
- Let us compare the nearest neighbor (FedEx Kinkos) to a query point Piano store using both geodesic and exact distance measures
  - geodesic ordering M O N D G
  - network distance ordering O D N M G (Error: +26 miles)

Application – Find the closest Kinko's



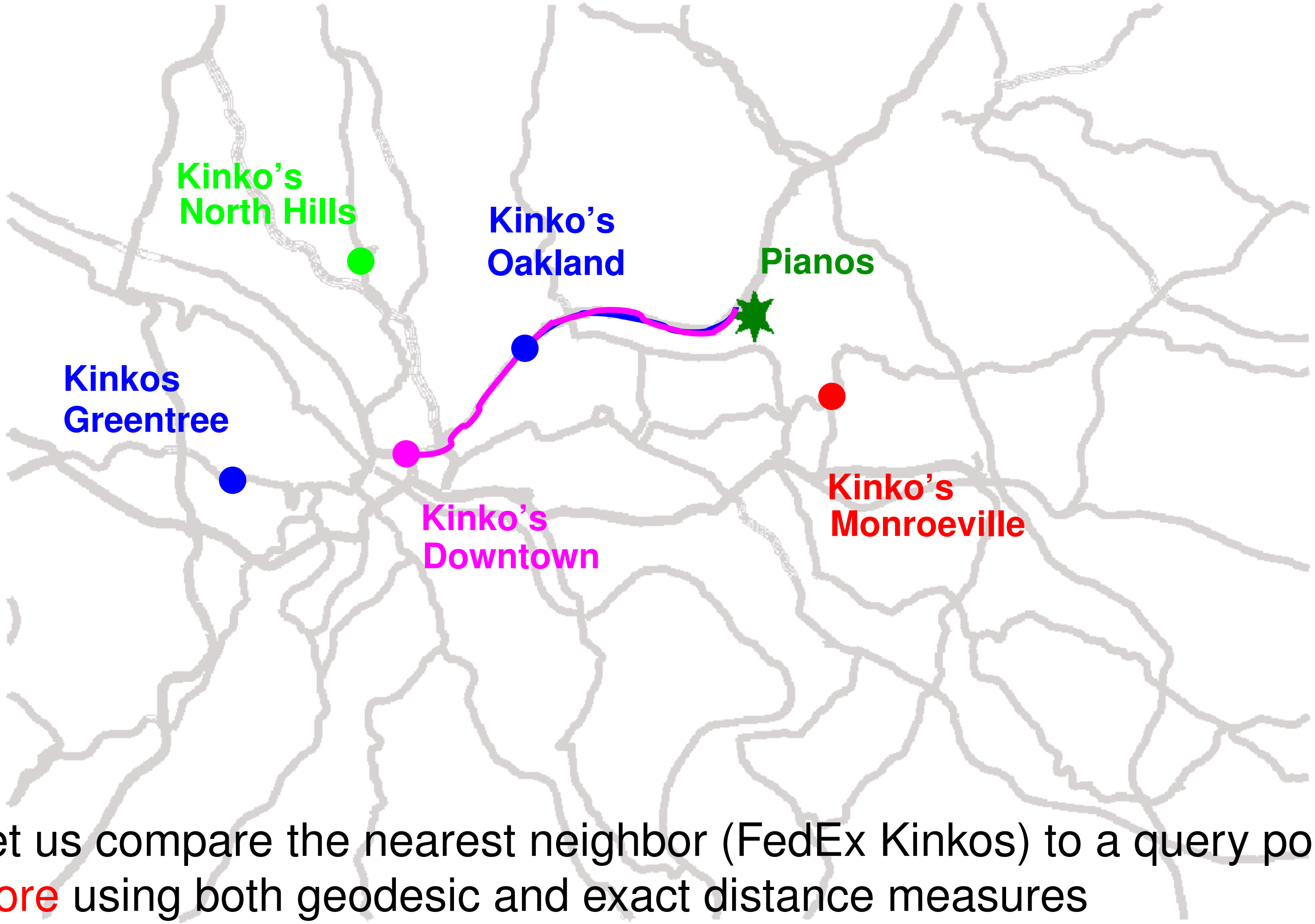
- Let us compare the nearest neighbor (FedEx Kinkos) to a query point Piano store using both geodesic and exact distance measures
  - geodesic ordering M O N D G
  - network distance ordering O D N M G (Error: +26 miles)
  - trafficability ordering

Application – Find the closest Kinko's



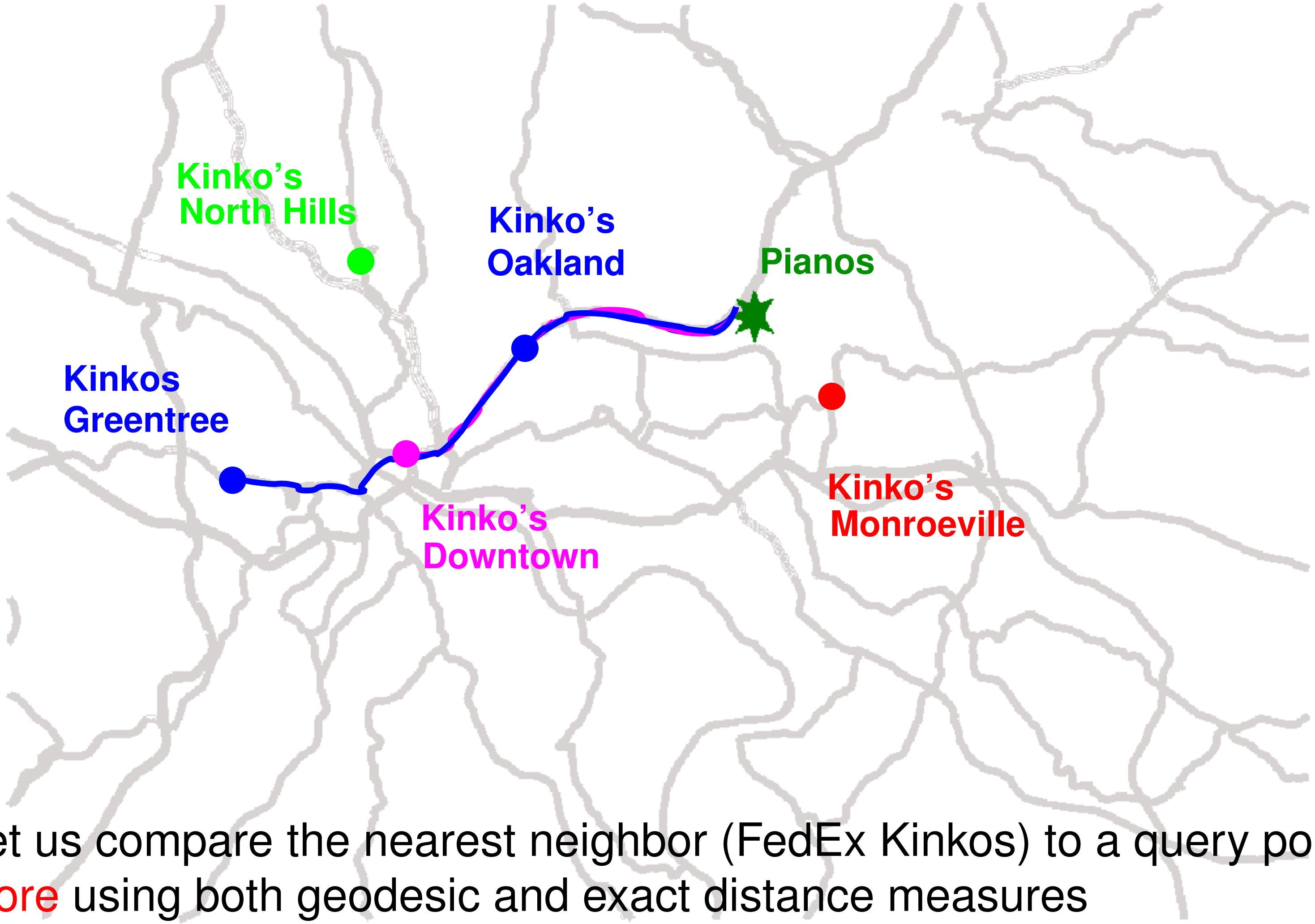
- Let us compare the nearest neighbor (FedEx Kinkos) to a query point Piano store using both geodesic and exact distance measures
  - geodesic ordering M O N D G
  - network distance ordering O D N M G (Error: +26 miles)
  - trafficability ordering O

Application – Find the closest Kinko's



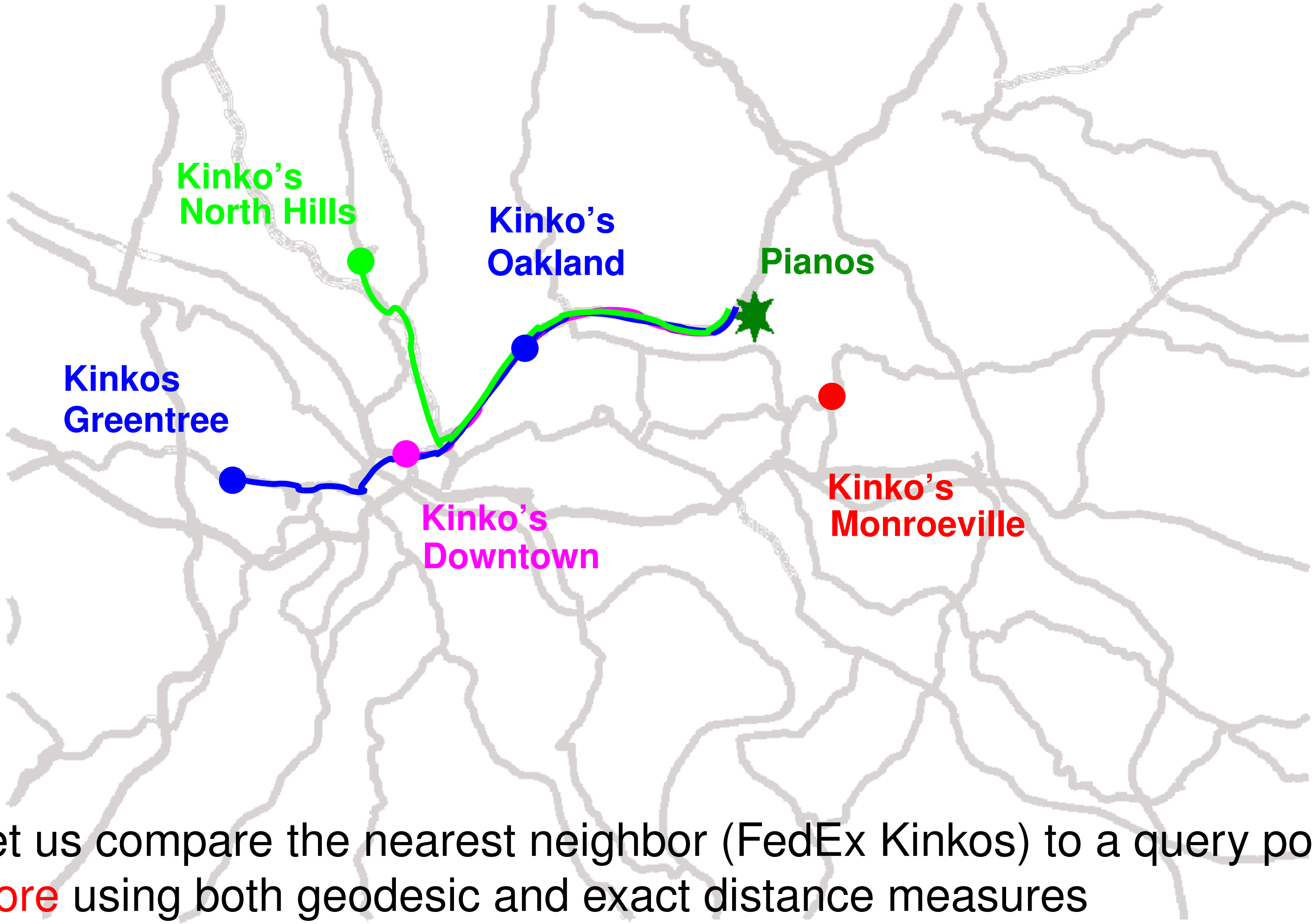
- Let us compare the nearest neighbor (FedEx Kinkos) to a query point Piano store using both geodesic and exact distance measures
  - geodesic ordering M O N D G
  - network distance ordering O D N M G (Error: +26 miles)
  - trafficability ordering O D

Application – Find the closest Kinko's



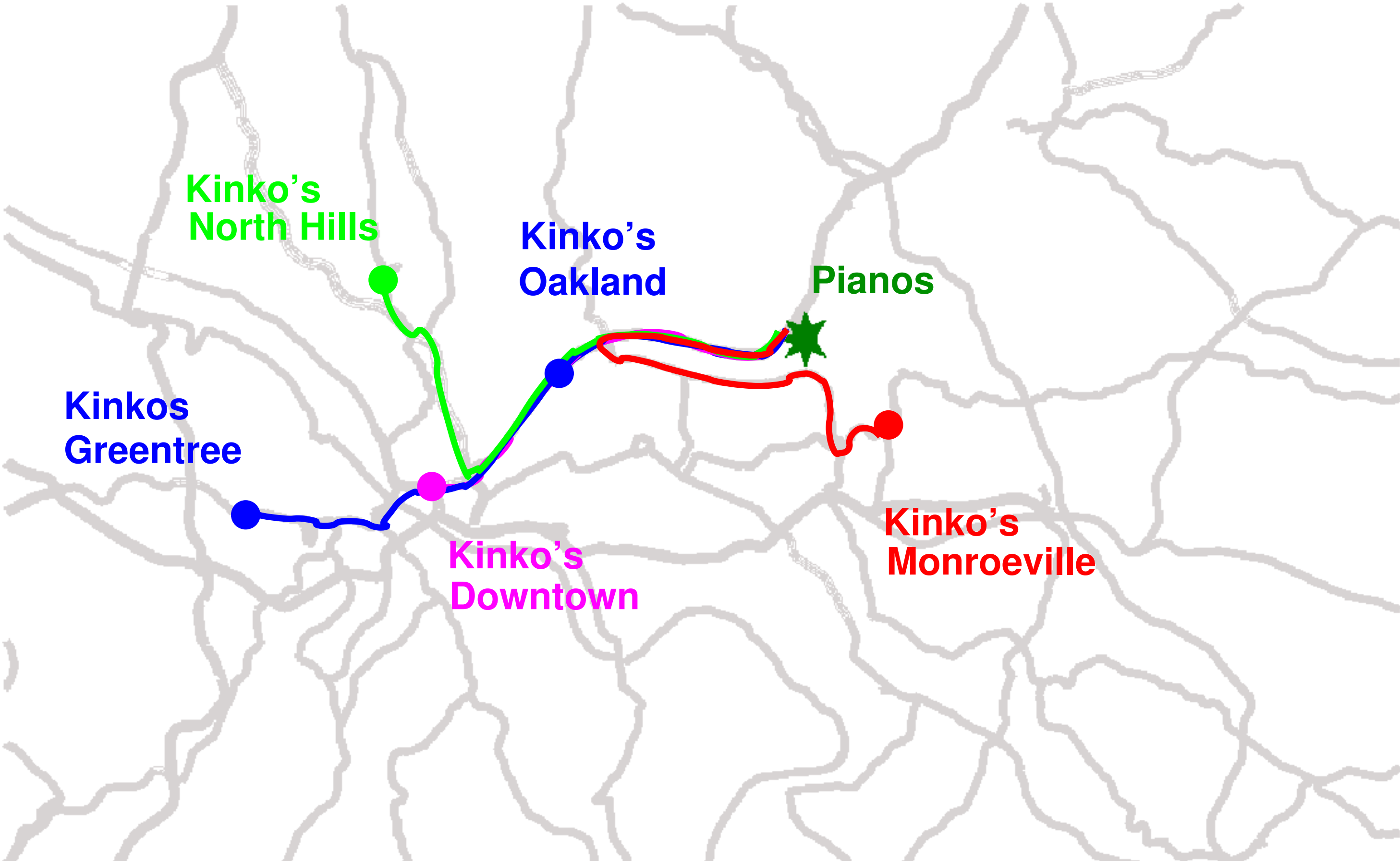
- Let us compare the nearest neighbor (FedEx Kinkos) to a query point Piano store using both geodesic and exact distance measures
  - geodesic ordering M O N D G
  - network distance ordering O D N M G (Error: +26 miles)
  - trafficability ordering O D G

Application – Find the closest Kinko's



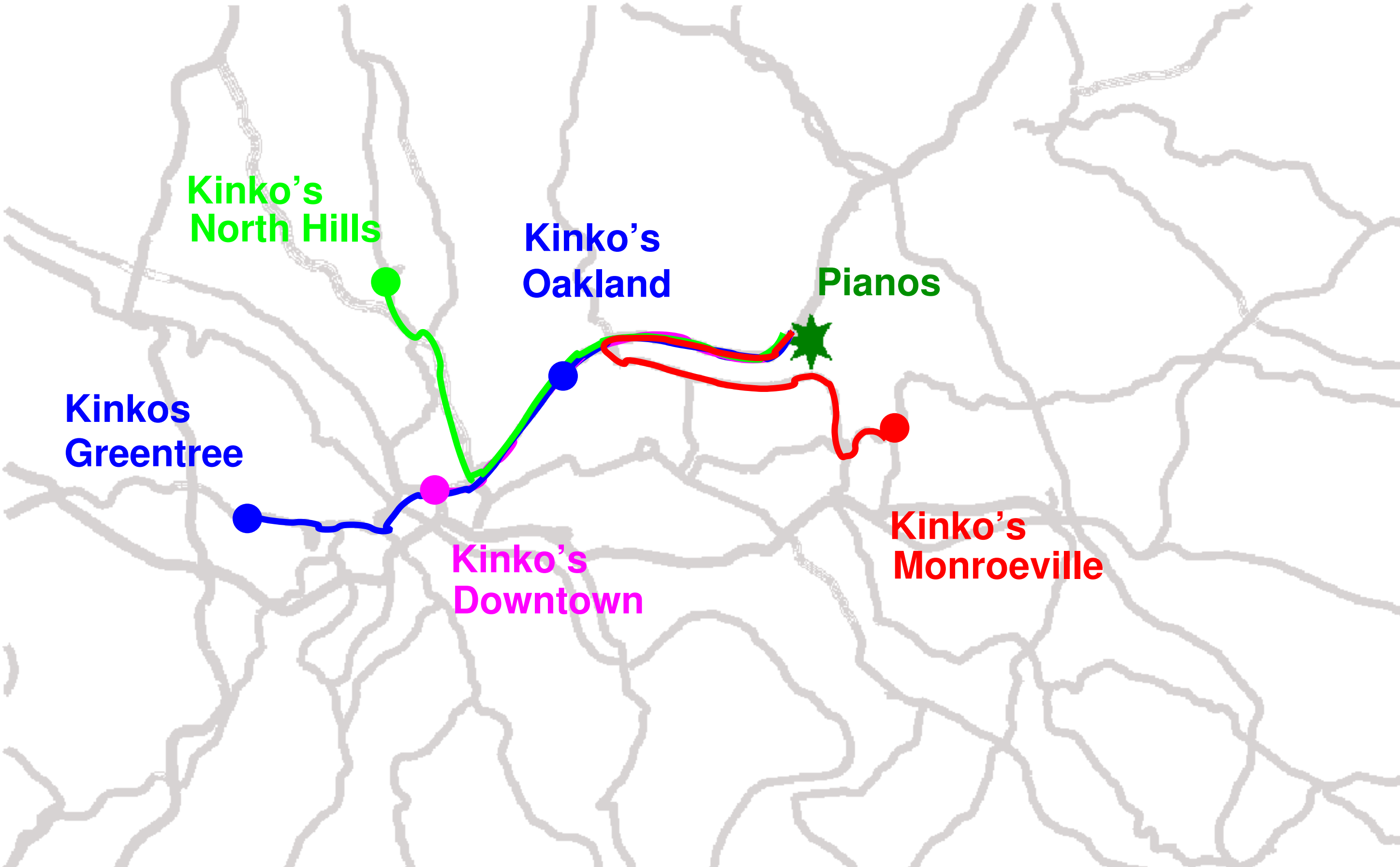
- Let us compare the nearest neighbor (FedEx Kinkos) to a query point Piano store using both geodesic and exact distance measures
  - geodesic ordering M O N D G
  - network distance ordering O D N M G (Error: +26 miles)
  - trafficability ordering O D G N

Application – Find the closest Kinko's



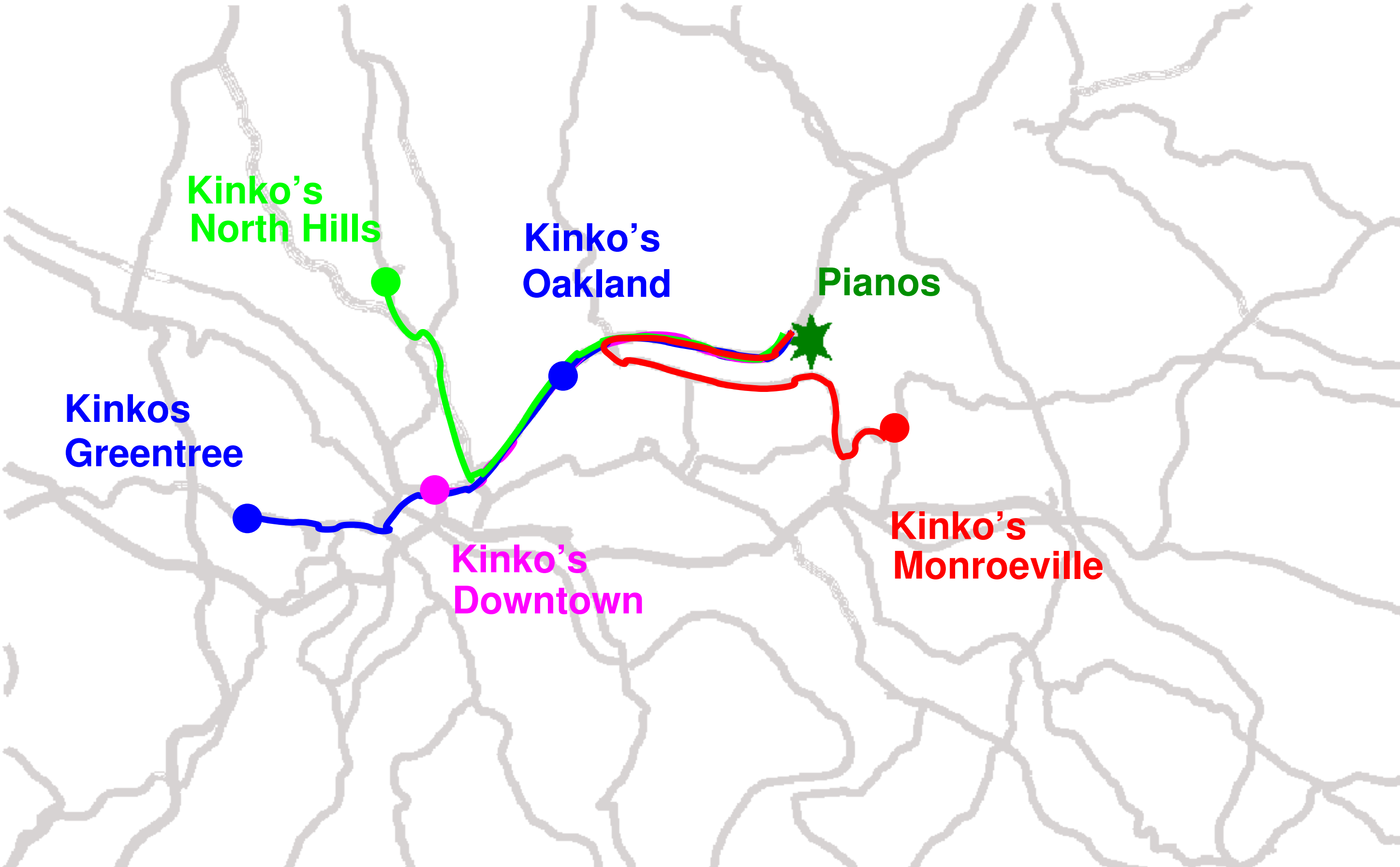
- Let us compare the nearest neighbor (FedEx Kinkos) to a query point Piano store using both geodesic and exact distance measures
  - geodesic ordering M O N D G
  - network distance ordering O D N M G (Error: +26 miles)
  - trafficability ordering O D G N M

Application – Find the closest Kinko's



- Let us compare the nearest neighbor (FedEx Kinkos) to a query point Piano store using both geodesic and exact distance measures
  - geodesic ordering M O N D G
  - network distance ordering O D N M G (Error: +26 miles)
  - trafficability ordering O D G N M (Error: +32 minutes)

Application – Find the closest Kinko's



- Let us compare the nearest neighbor (FedEx Kinkos) to a query point Piano store using both geodesic and exact distance measures
  - geodesic ordering M O N D G
  - network distance ordering O D N M G (Error: +26 miles)
  - trafficability ordering O D G N M (Error: +32 minutes)
- Challenge: Real time + exact queries

Proximity Search on “Google Local”

- Let us examine the errors between ordering by the spatial distance (“as the crow flies” used by Google) and by the network distance (used by us)

Proximity Search on “Google Local”

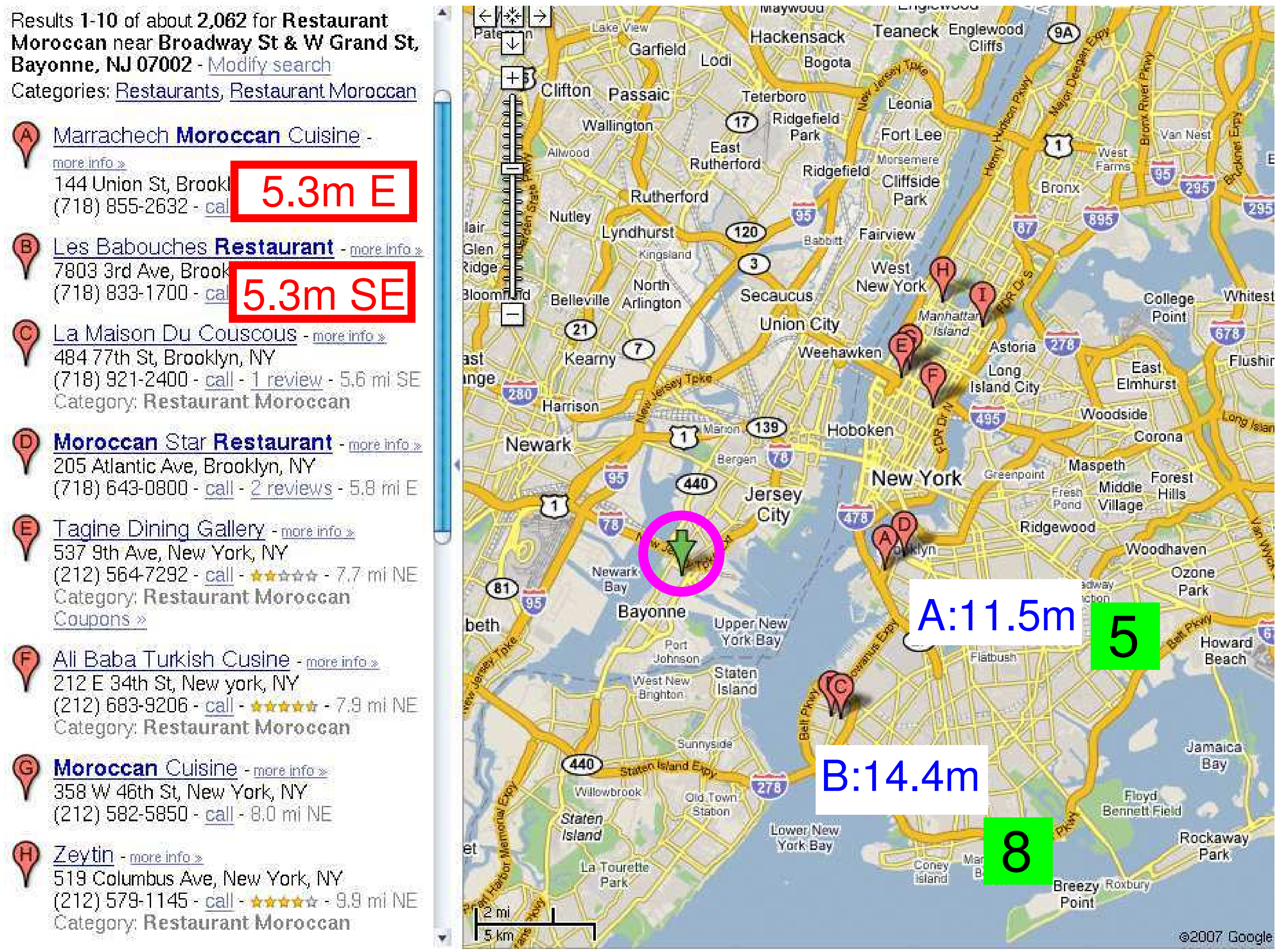
- Let us examine the errors between ordering by the spatial distance (“as the crow flies” used by Google) and by the network distance (used by us)

Proximity Search on “Google Local”

- Let us examine the errors between ordering by the spatial distance (“as the crow flies” used by Google) and by the network distance (used by us)

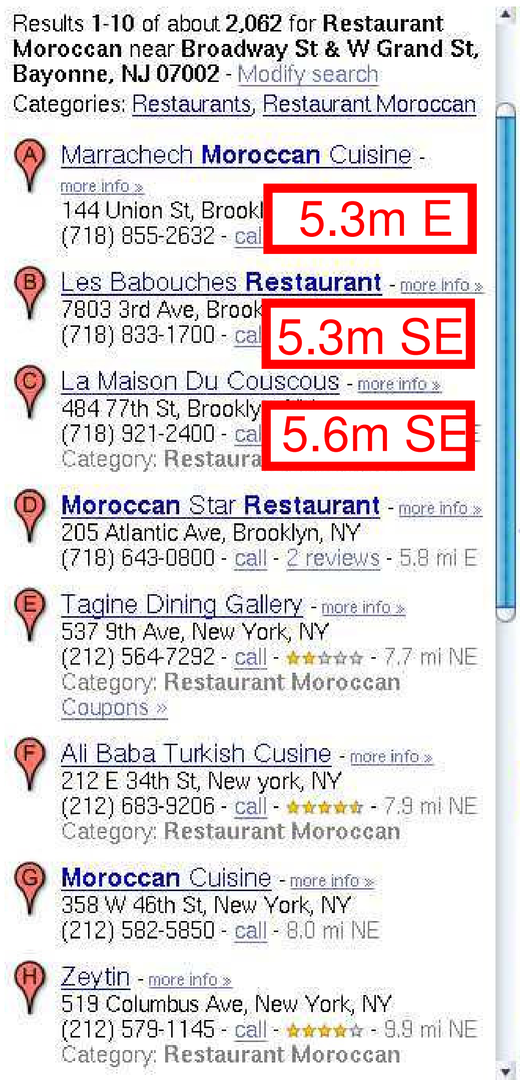
Proximity Search on “Google Local”

- Let us examine the errors between ordering by the spatial distance (“as the crow flies” used by Google) and by the network distance (used by us)



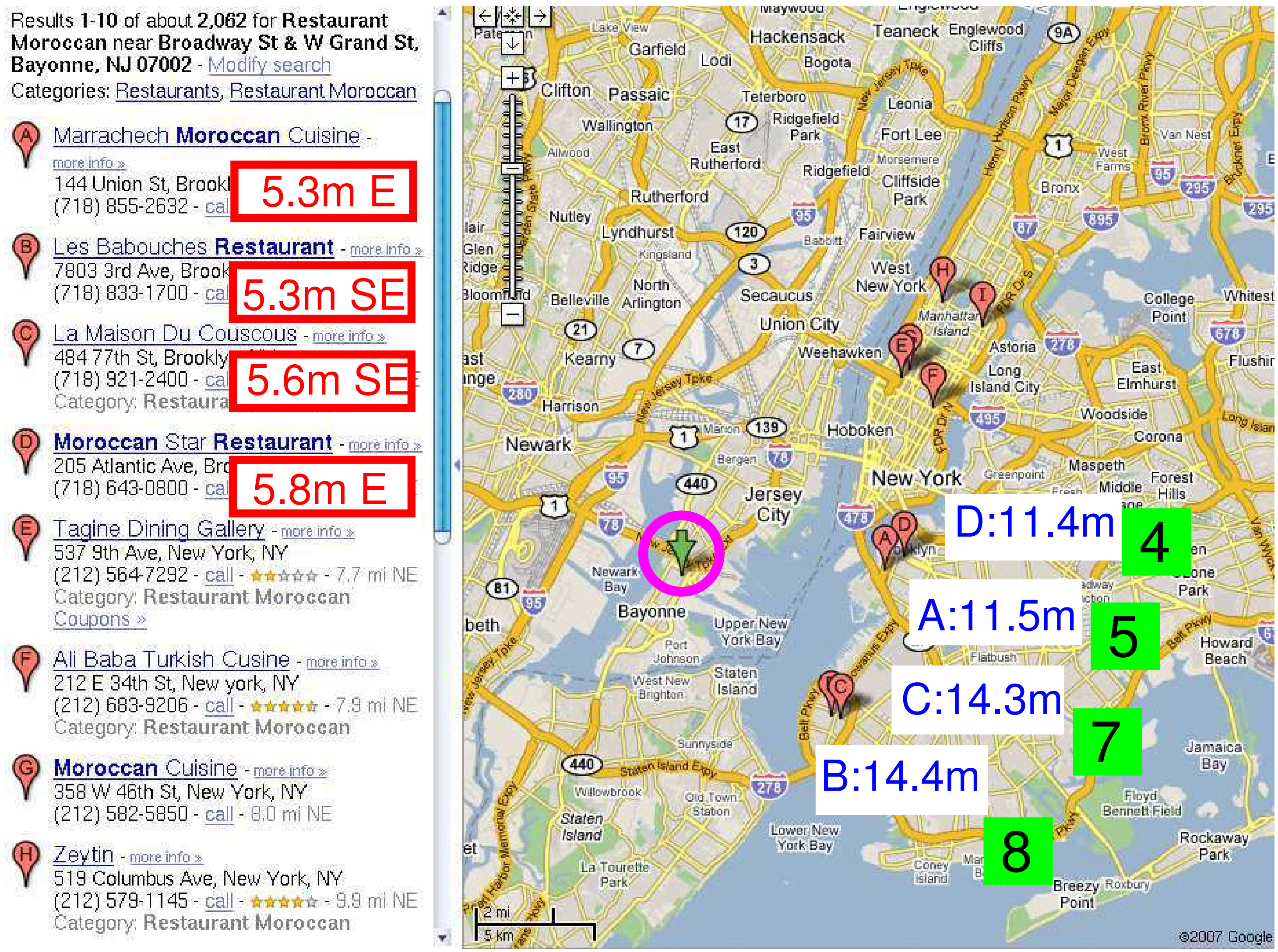
Proximity Search on “Google Local”

- Let us examine the errors between ordering by the spatial distance (“as the crow flies” used by Google) and by the network distance (used by us)



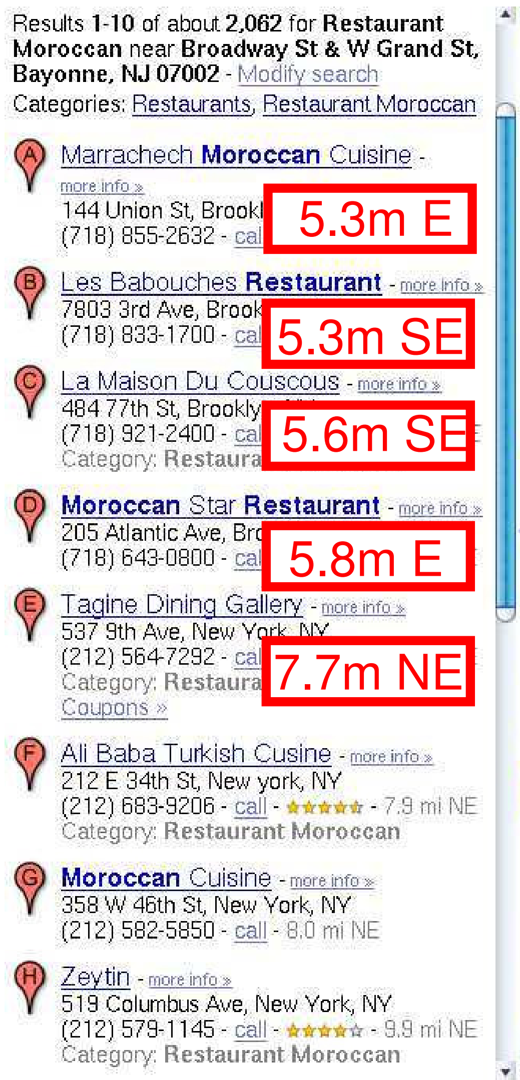
Proximity Search on “Google Local”

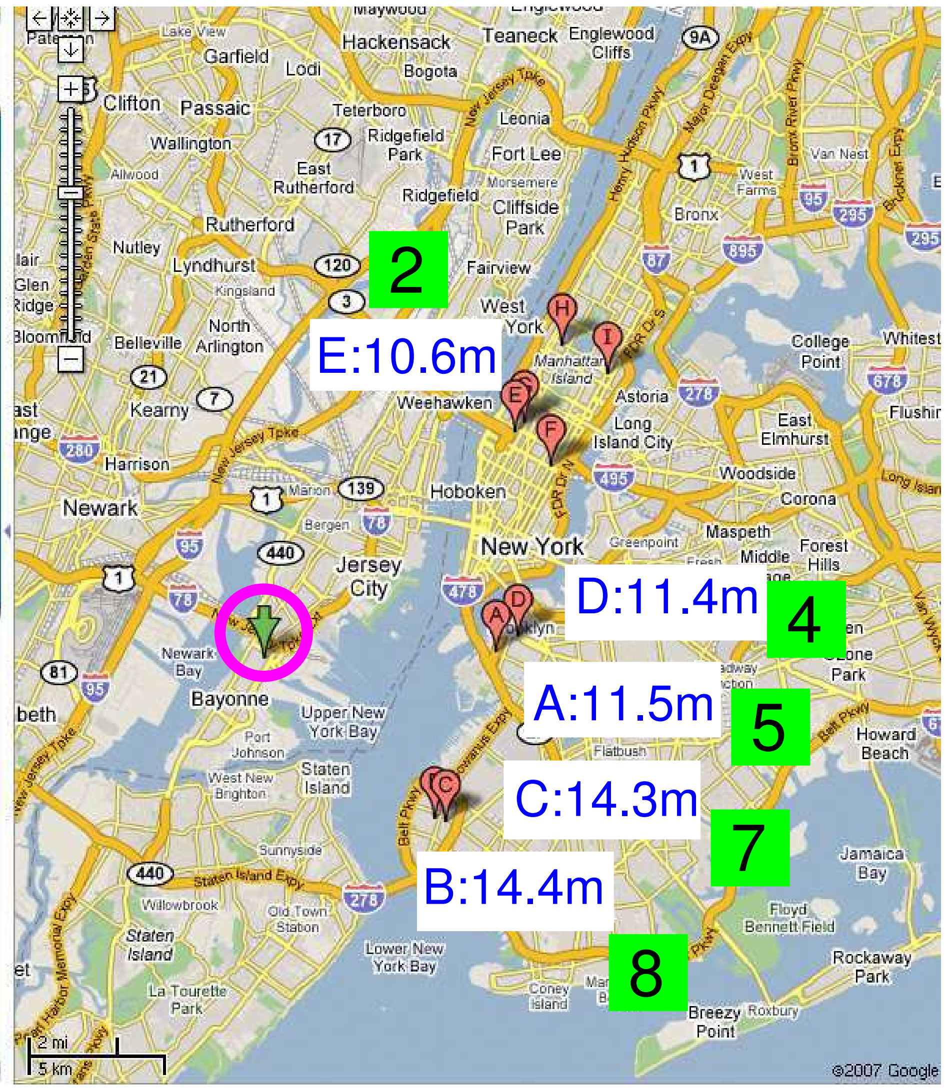
- Let us examine the errors between ordering by the spatial distance (“as the crow flies” used by Google) and by the network distance (used by us)



Proximity Search on “Google Local”

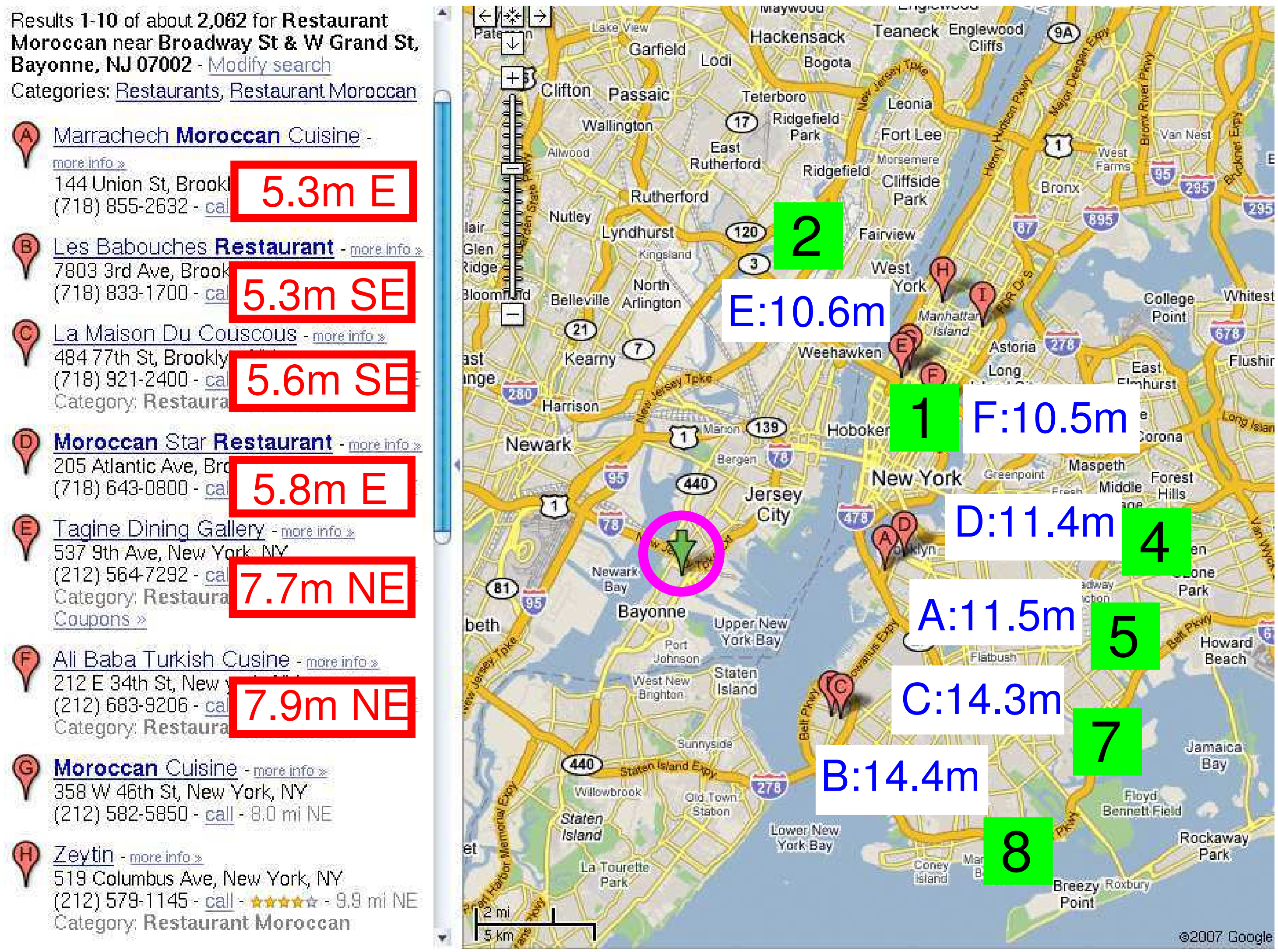
- Let us examine the errors between ordering by the spatial distance (“as the crow flies” used by Google) and by the network distance (used by us)





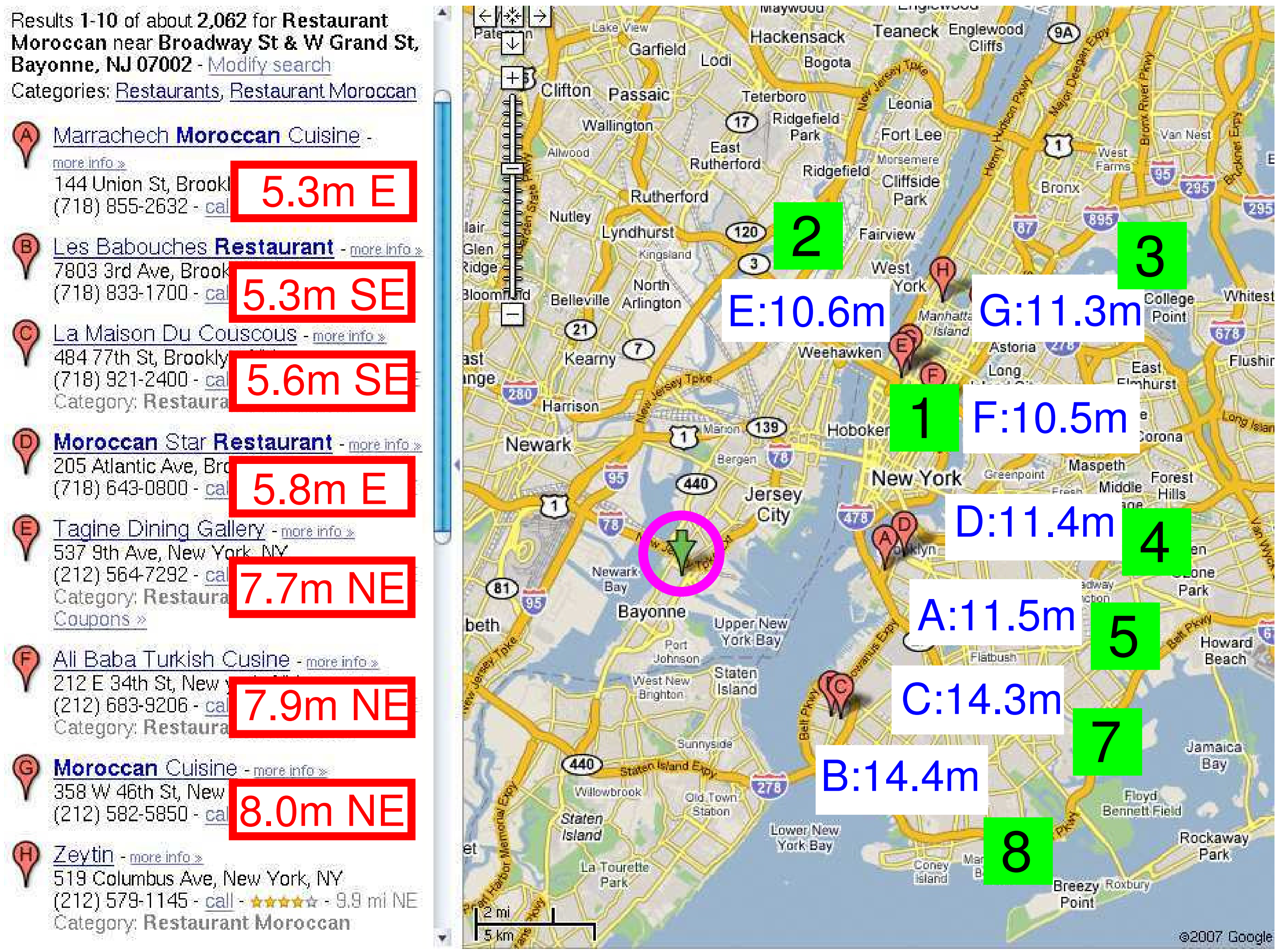
Proximity Search on “Google Local”

- Let us examine the errors between ordering by the spatial distance (“as the crow flies” used by Google) and by the network distance (used by us)



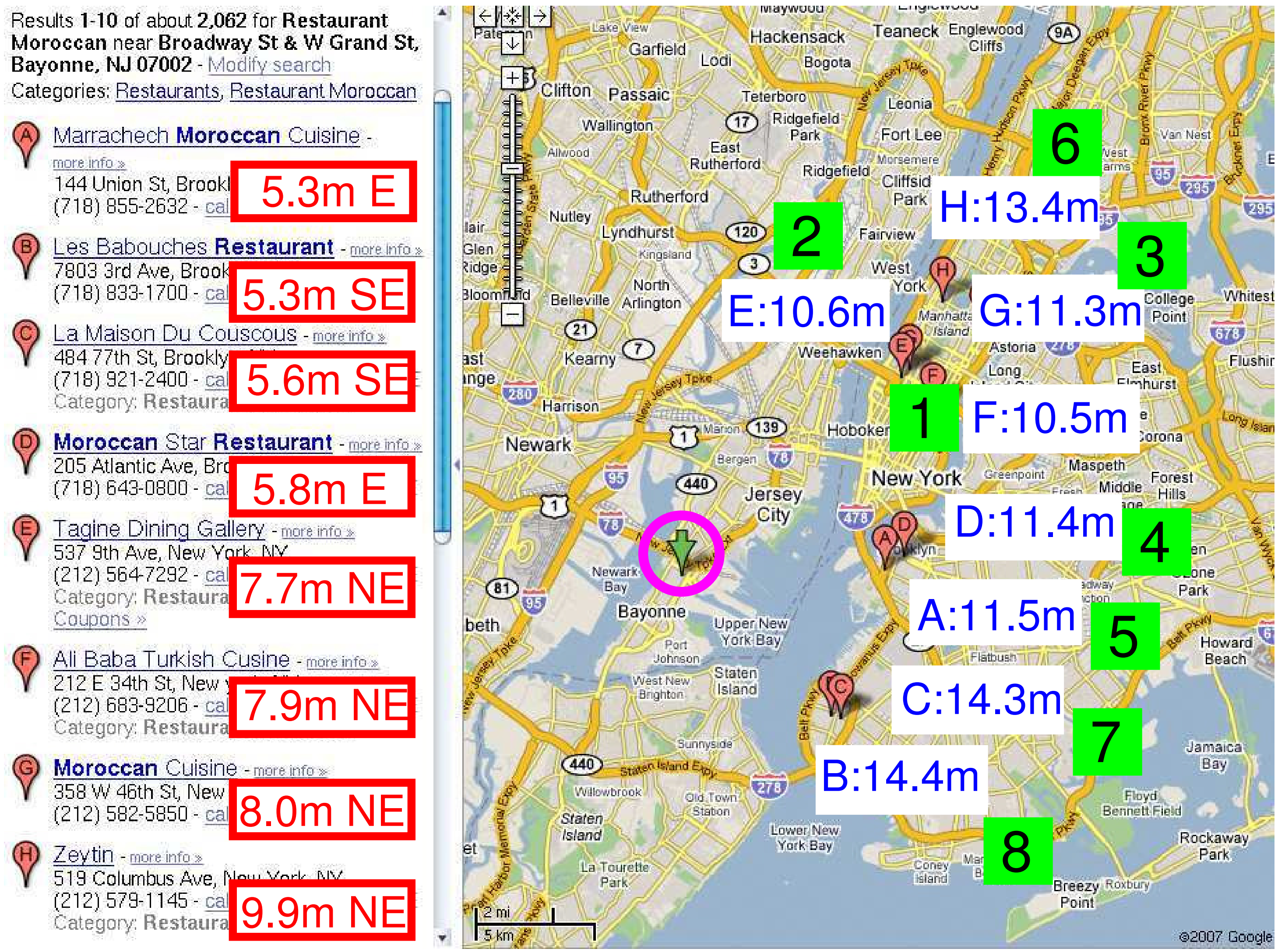
Proximity Search on “Google Local”

- Let us examine the errors between ordering by the spatial distance (“as the crow flies” used by Google) and by the network distance (used by us)



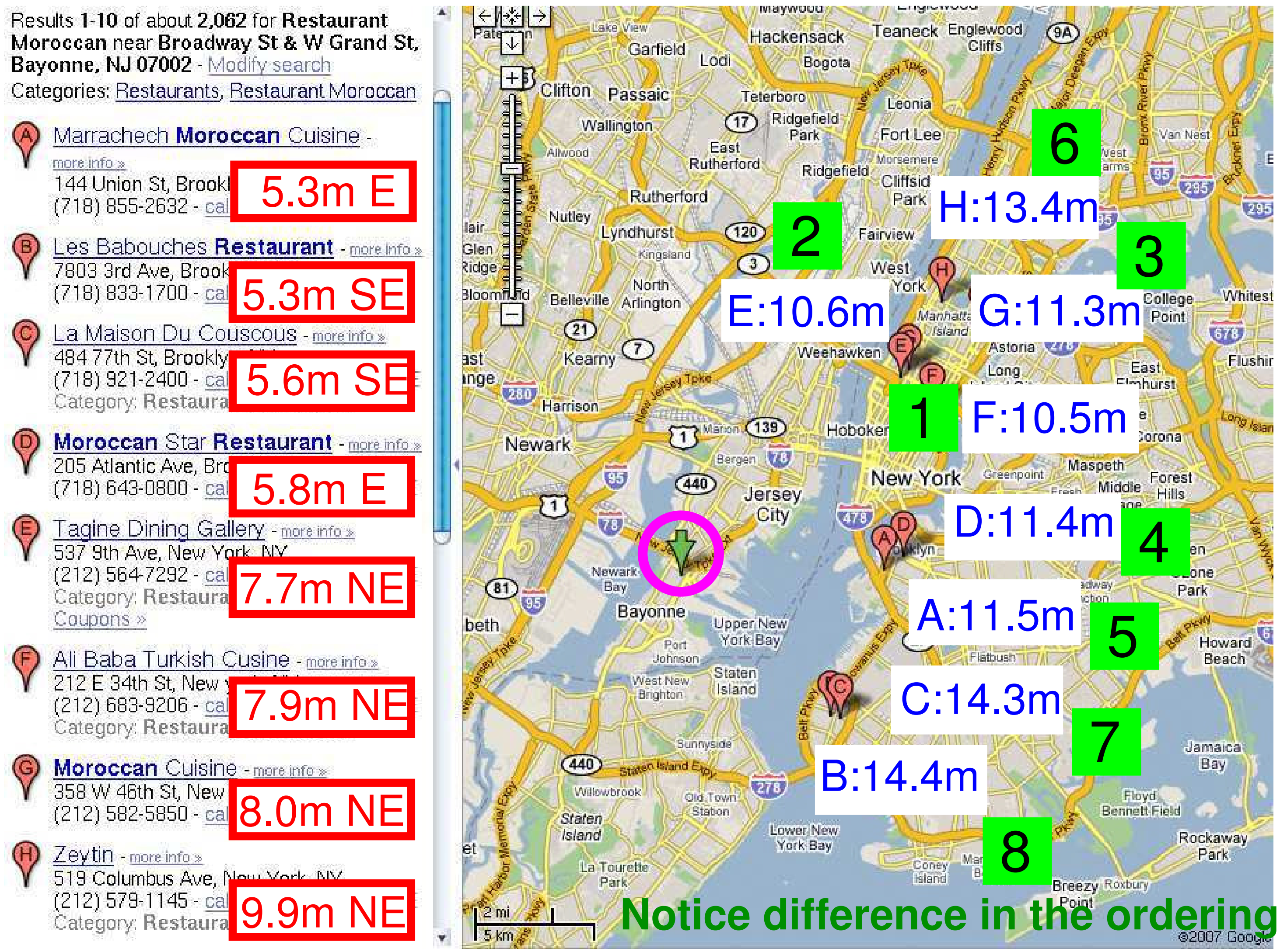
Proximity Search on “Google Local”

- Let us examine the errors between ordering by the spatial distance (“as the crow flies” used by Google) and by the network distance (used by us)



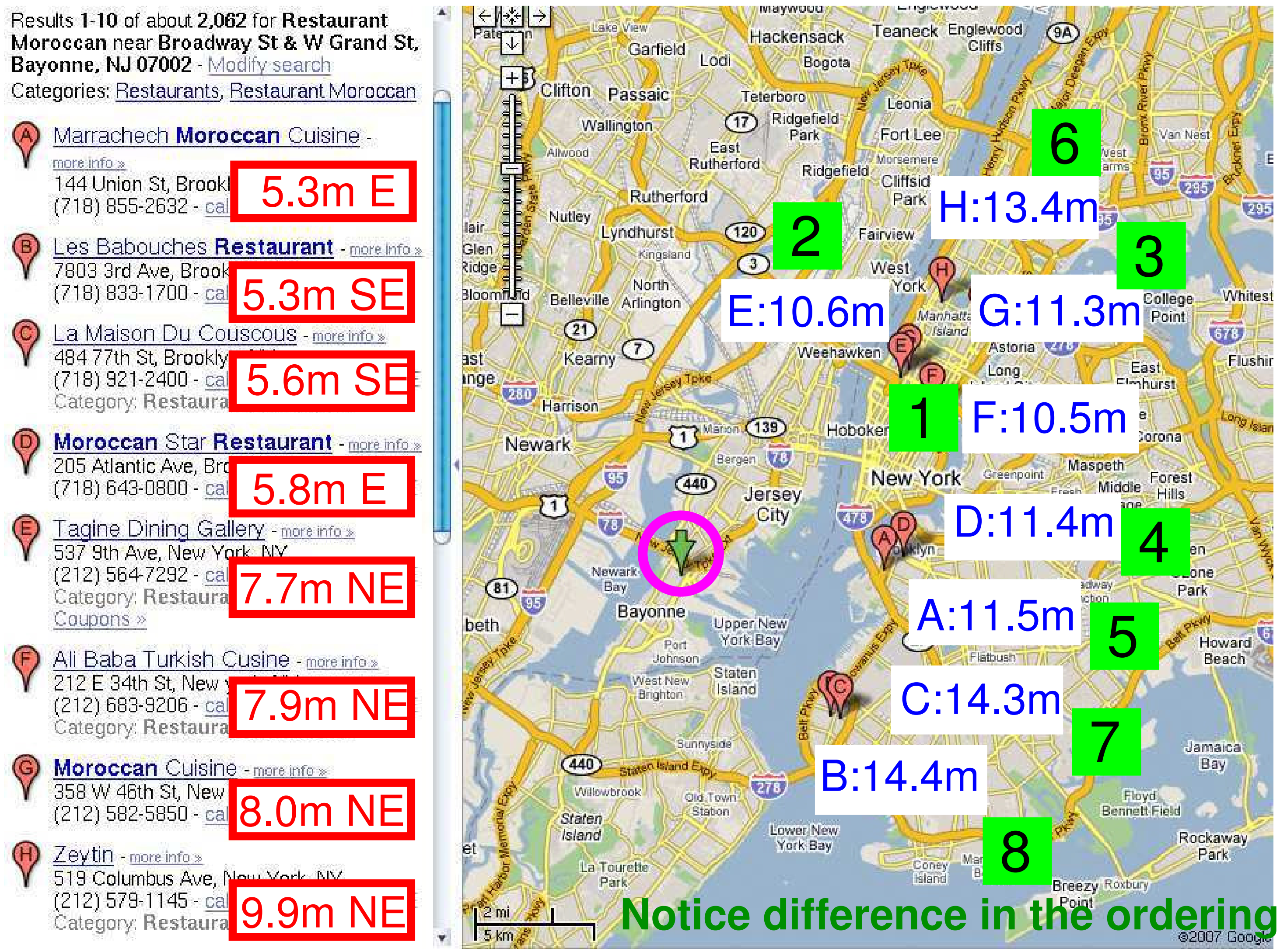
Proximity Search on “Google Local”

- Let us examine the errors between ordering by the spatial distance (“as the crow flies” used by Google) and by the network distance (used by us)



Proximity Search on "Google Local"

- Let us examine the errors between ordering by the spatial distance ("as the crow flies" used by Google) and by the network distance (used by us)



- Goal: Instant answers as well as accurate answers

SILC: Using Path Coherence to Encode Shortest Paths

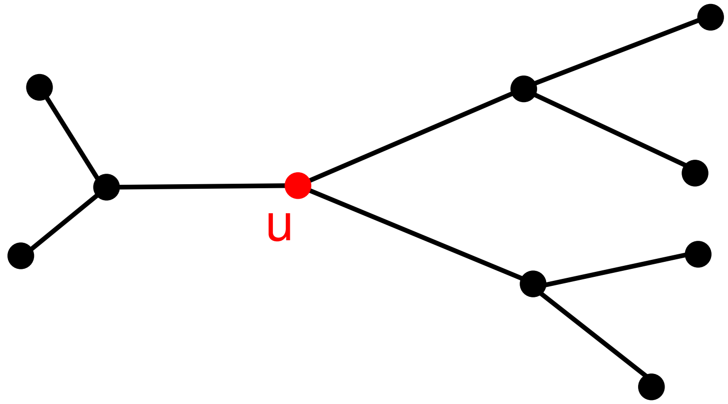
- The SILC path encoding takes advantage of the path coherence

SILC: Using Path Coherence to Encode Shortest Paths

- The SILC path encoding takes advantage of the path coherence
- How? Use a *coloring* algorithm

SILC: Using Path Coherence to Encode Shortest Paths

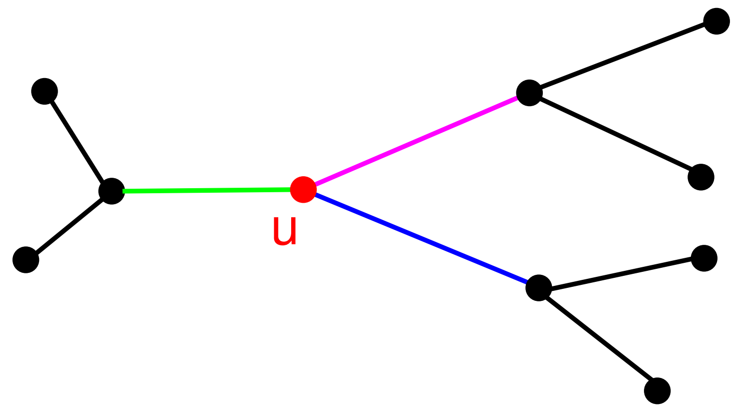
- The SILC path encoding takes advantage of the path coherence
- How? Use a *coloring* algorithm



- Source vertex u in a spatial network

SILC: Using Path Coherence to Encode Shortest Paths

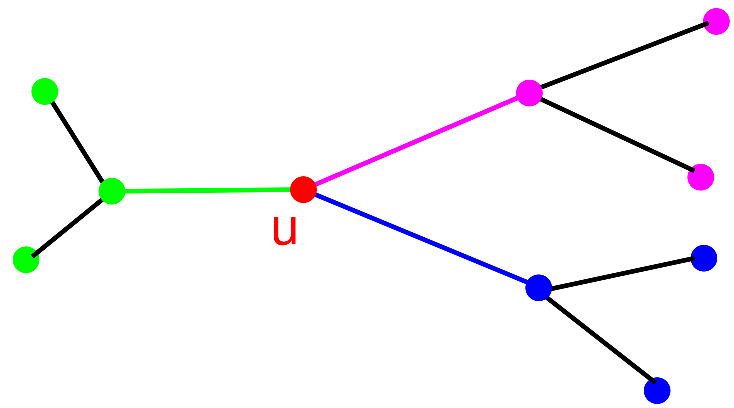
- The SILC path encoding takes advantage of the path coherence
- How? Use a *coloring* algorithm



  - Source vertex u in a spatial network
  - Assign colors to the outgoing edges of u

SILC: Using Path Coherence to Encode Shortest Paths

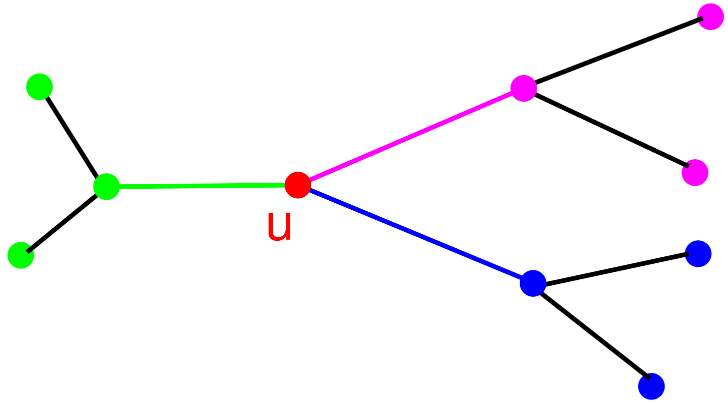
- The SILC path encoding takes advantage of the path coherence
- How? Use a *coloring* algorithm

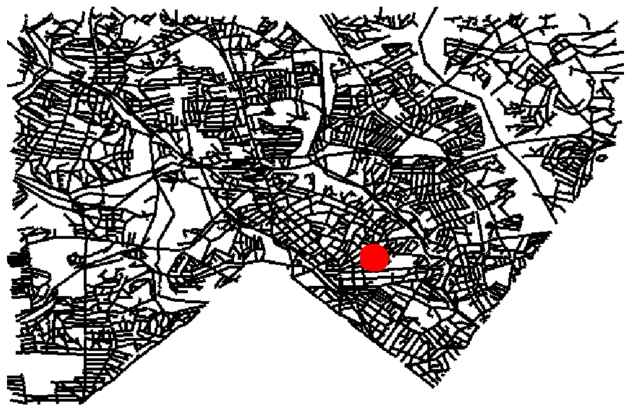


- Source vertex u in a spatial network
- Assign colors to the outgoing edges of u
- Color vertex based on the first edge on the shortest path from u

SILC: Using Path Coherence to Encode Shortest Paths

- The SILC path encoding takes advantage of the path coherence
- How? Use a *coloring* algorithm



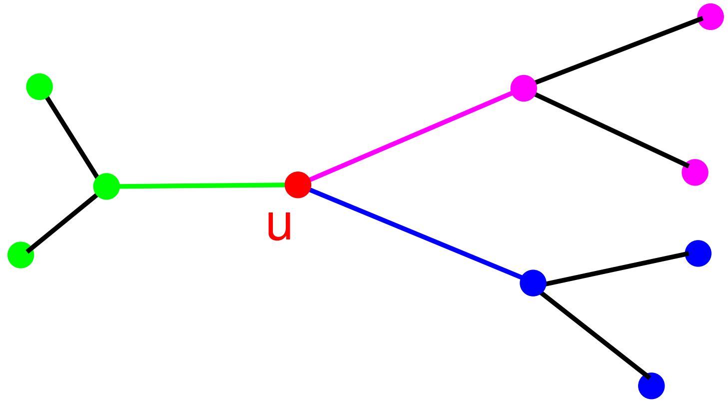


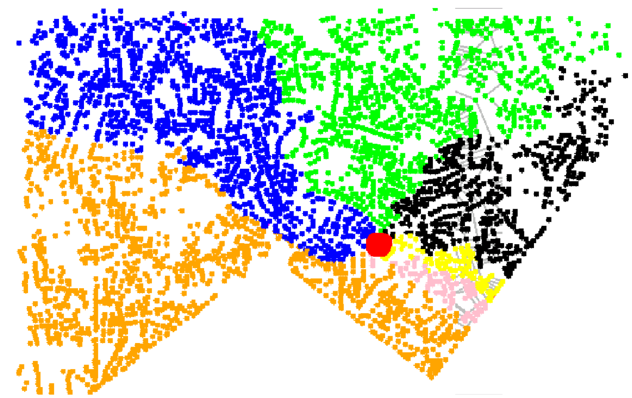
- Source vertex u in a spatial network
- Assign colors to the outgoing edges of u
- Color vertex based on the first edge on the shortest path from u

- Source vertex u in the spatial network of Silver Spring, MD

SILC: Using Path Coherence to Encode Shortest Paths

- The SILC path encoding takes advantage of the path coherence
- How? Use a *coloring* algorithm



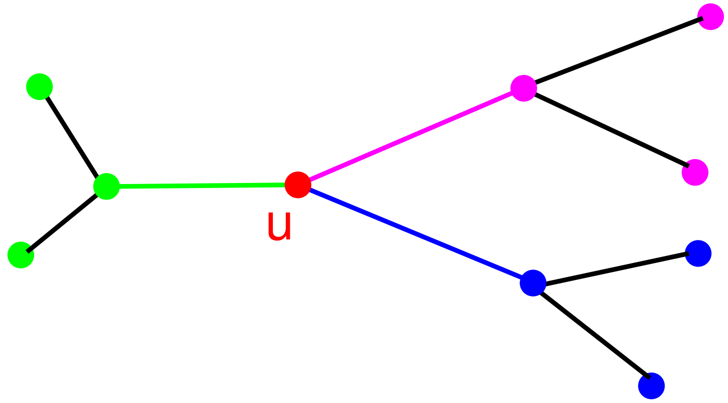


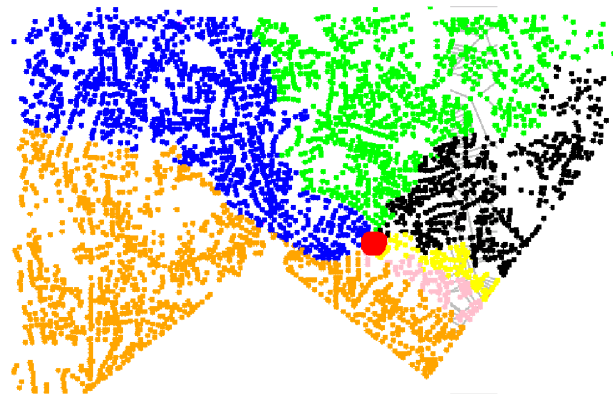
- Source vertex u in a spatial network
- Assign colors to the outgoing edges of u
- Color vertex based on the first edge on the shortest path from u

- Source vertex u in the spatial network of Silver Spring, MD
- Color remaining vertices based on which of the six adjacent vertices of u is the first link in the shortest path from u

SILC: Using Path Coherence to Encode Shortest Paths

- The SILC path encoding takes advantage of the path coherence
- How? Use a *coloring* algorithm





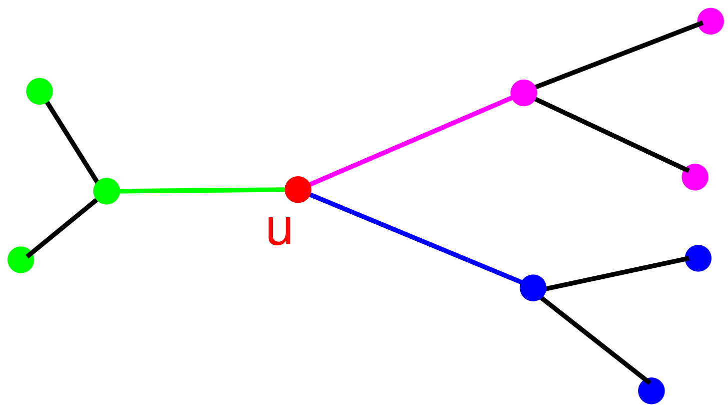
- Source vertex u in a spatial network
- Assign colors to the outgoing edges of u
- Color vertex based on the first edge on the shortest path from u

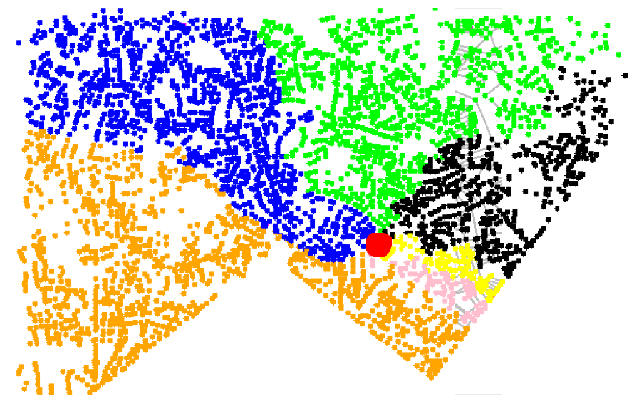
- Source vertex u in the spatial network of Silver Spring, MD
- Color remaining vertices based on which of the six adjacent vertices of u is the first link in the shortest path from u

- Resulting representation is termed the *shortest-path map* of u

SILC: Using Path Coherence to Encode Shortest Paths

- The SILC path encoding takes advantage of the path coherence
- How? Use a *coloring* algorithm



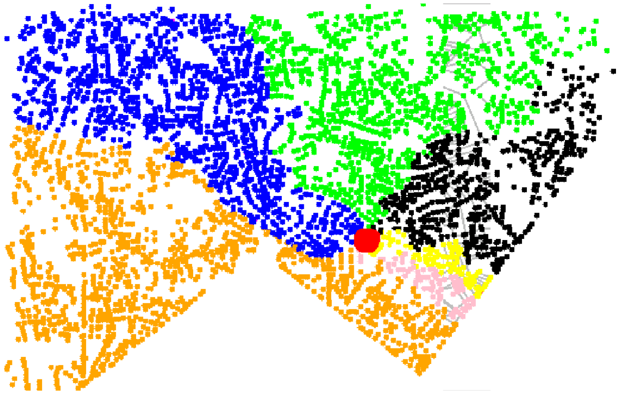


- Source vertex u in a spatial network
- Assign colors to the outgoing edges of u
- Color vertex based on the first edge on the shortest path from u

- Source vertex u in the spatial network of Silver Spring, MD
- Color remaining vertices based on which of the six adjacent vertices of u is the first link in the shortest path from u

- Resulting representation is termed the *shortest-path map* of u
- Assuming planar spatial network graphs means that the coloring results in spatially contiguous colored regions due to path coherence

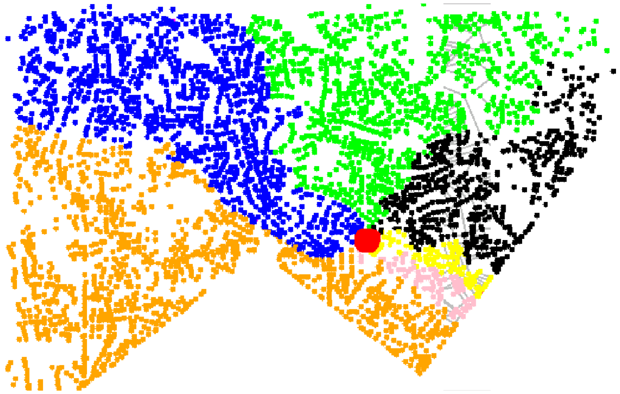
How to Store Colored Regions?



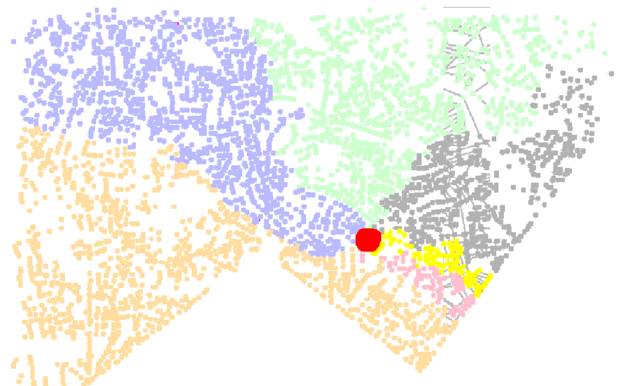
Shortest-path Map

How to Store Colored Regions?

- Minimum bounding boxes (e.g., R-tree) [Wagn03]



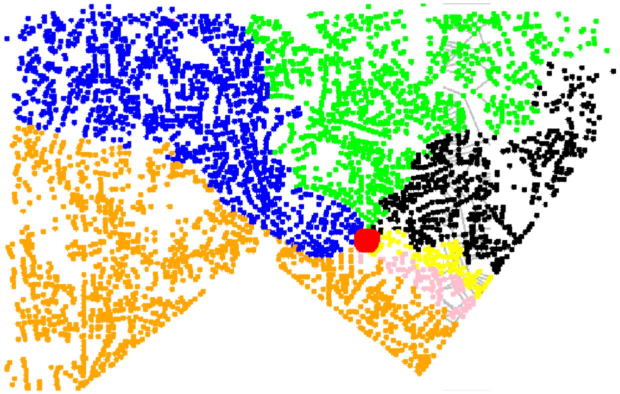
Shortest-path Map



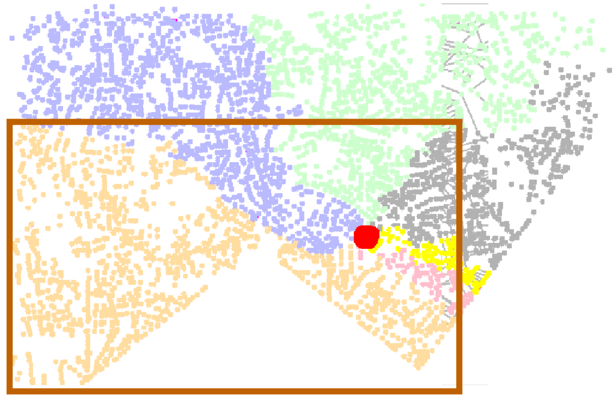
R-tree

How to Store Colored Regions?

- Minimum bounding boxes (e.g., R-tree) [Wagn03]



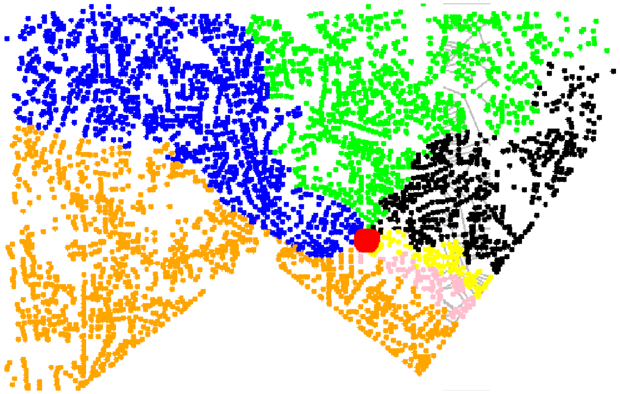
Shortest-path Map



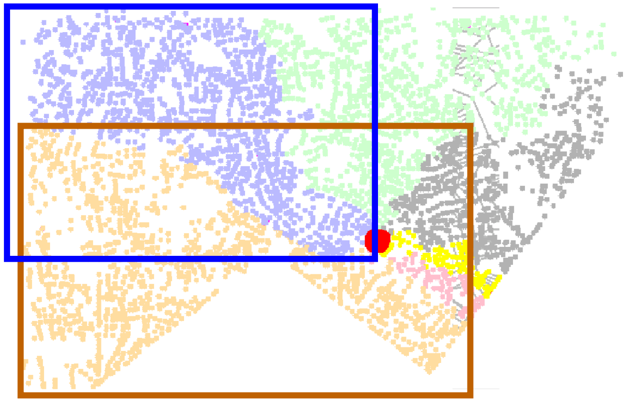
R-tree

How to Store Colored Regions?

- Minimum bounding boxes (e.g., R-tree) [Wagn03]



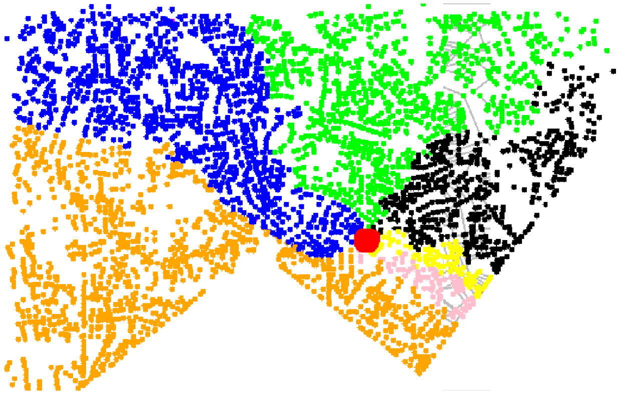
Shortest-path Map



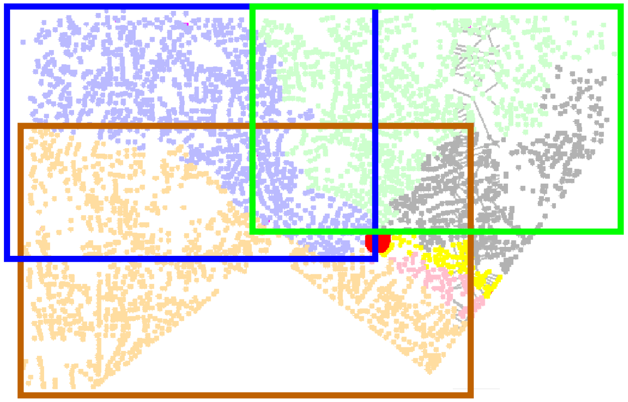
R-tree

How to Store Colored Regions?

- Minimum bounding boxes (e.g., R-tree) [Wagn03]



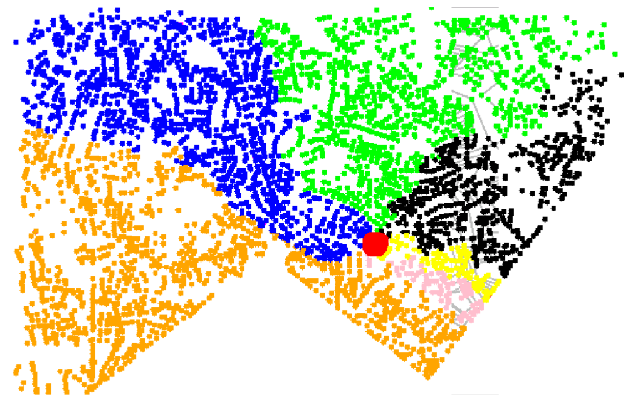
Shortest-path Map



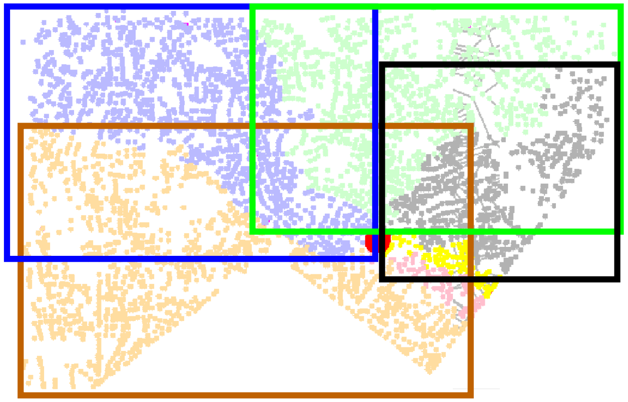
R-tree

How to Store Colored Regions?

- Minimum bounding boxes (e.g., R-tree) [Wagn03]



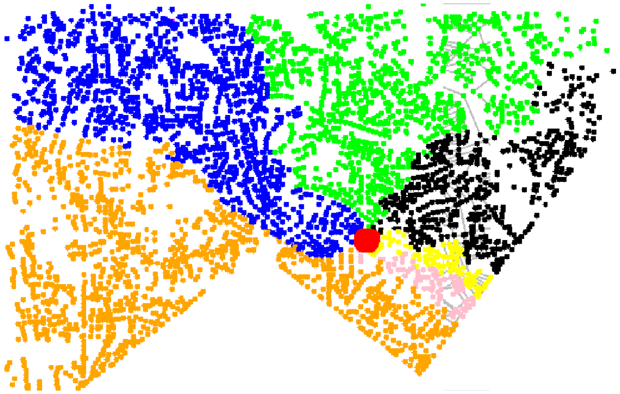
Shortest-path Map



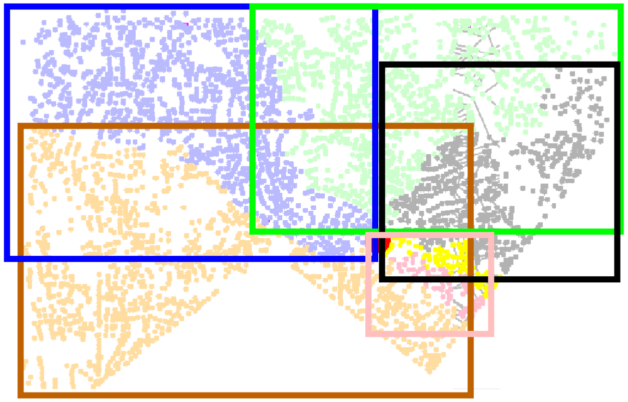
R-tree

How to Store Colored Regions?

- Minimum bounding boxes (e.g., R-tree) [Wagn03]



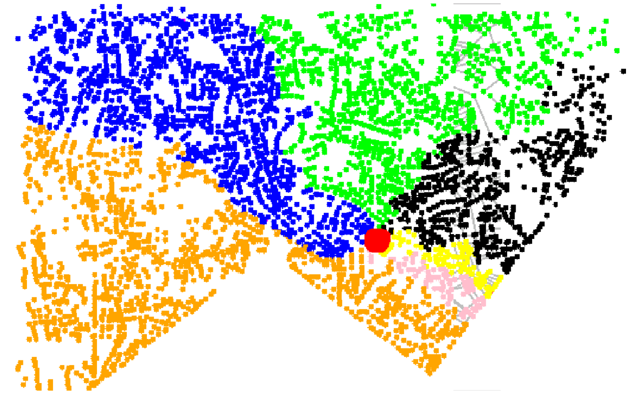
Shortest-path Map



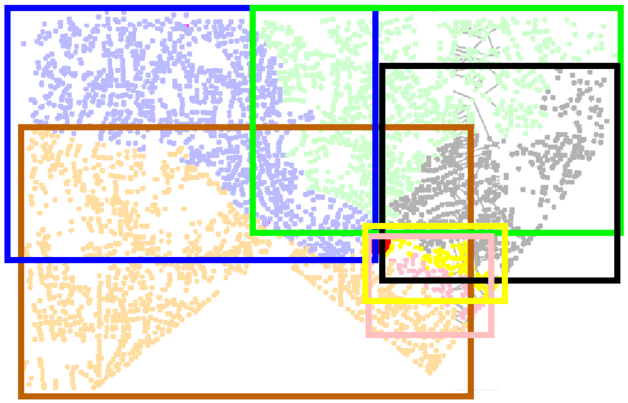
R-tree

How to Store Colored Regions?

- Minimum bounding boxes (e.g., R-tree) [Wagn03]



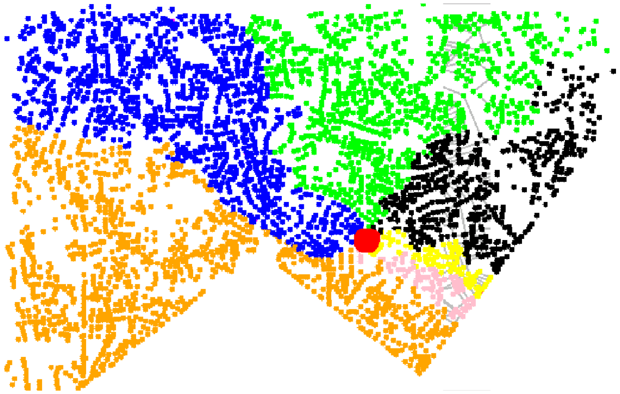
Shortest-path Map



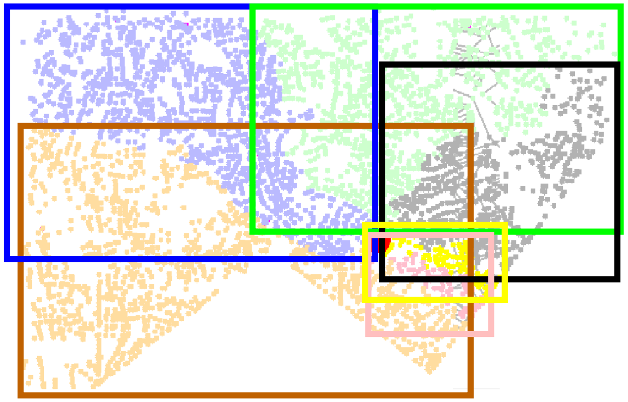
R-tree

How to Store Colored Regions?

- Minimum bounding boxes (e.g., R-tree) [Wagn03]
  - overlapping boxes imply identity of next vertex cannot be uniquely determined causing the shortest path algorithm to possibly degenerate to Dijkstra's algorithm



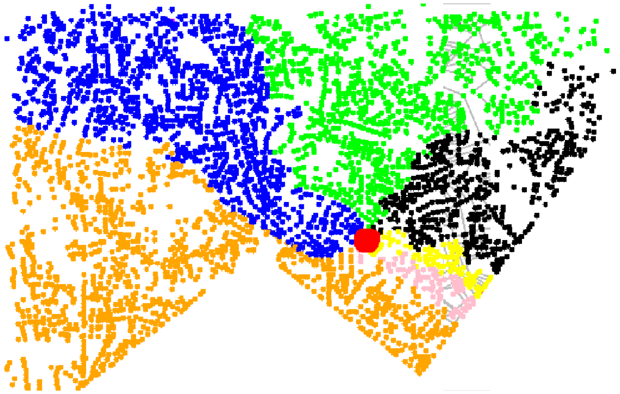
Shortest-path Map



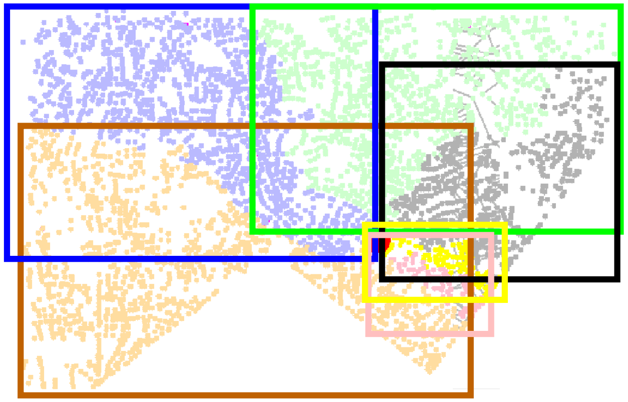
R-tree

How to Store Colored Regions?

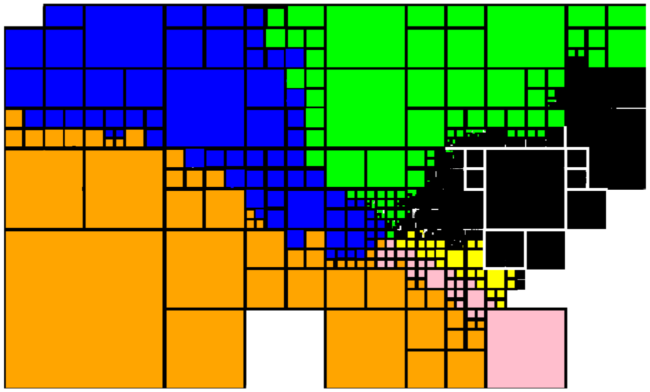
- Minimum bounding boxes (e.g., R-tree) [Wagn03]
  - overlapping boxes imply identity of next vertex cannot be uniquely determined causing the shortest path algorithm to possibly degenerate to Dijkstra's algorithm
- Disjoint decomposition: shortest-path quadtree



Shortest-path Map



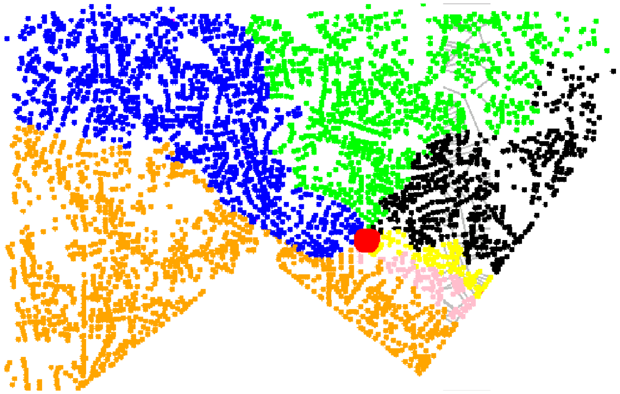
R-tree



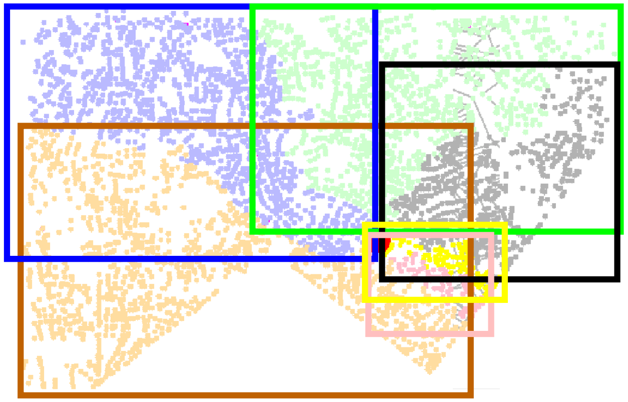
Shortest-Path Quadtree

How to Store Colored Regions?

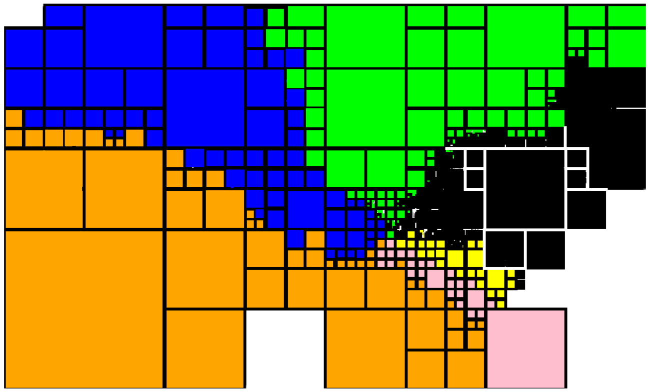
- Minimum bounding boxes (e.g., R-tree) [Wagn03]
  - overlapping boxes imply identity of next vertex cannot be uniquely determined causing the shortest path algorithm to possibly degenerate to Dijkstra's algorithm
- Disjoint decomposition: shortest-path quadtree
  - Decompose until all vertices in block have the same color



Shortest-path Map



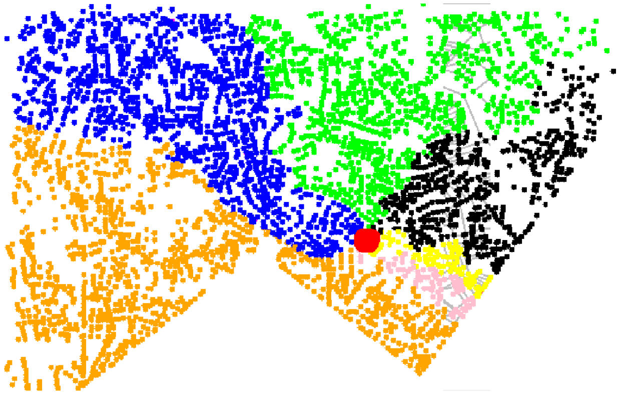
R-tree



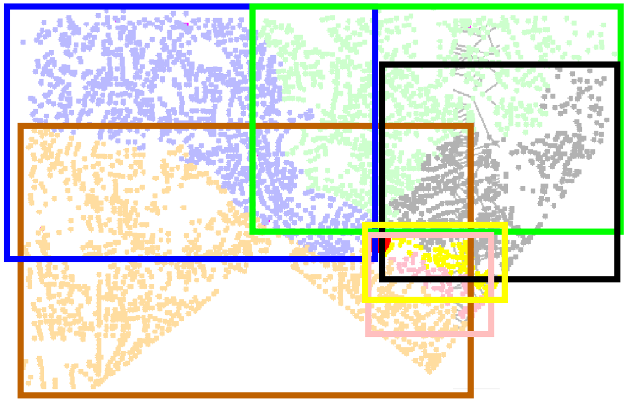
Shortest-Path Quadtree

How to Store Colored Regions?

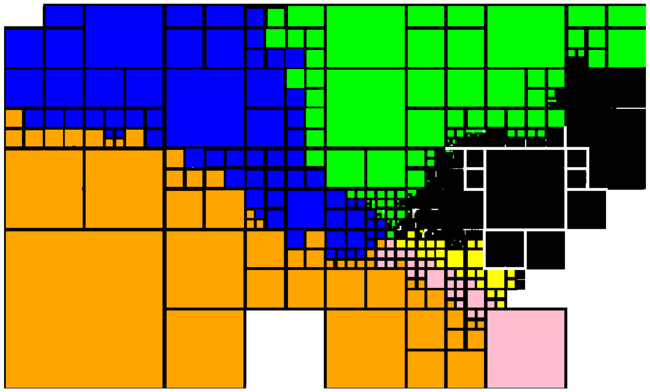
- Minimum bounding boxes (e.g., R-tree) [Wagn03]
  - overlapping boxes imply identity of next vertex cannot be uniquely determined causing the shortest path algorithm to possibly degenerate to Dijkstra's algorithm
- Disjoint decomposition: shortest-path quadtree
  - Decompose until all vertices in block have the same color
- Shortest-path quadtree stored as a collection of Morton blocks



Shortest-path Map



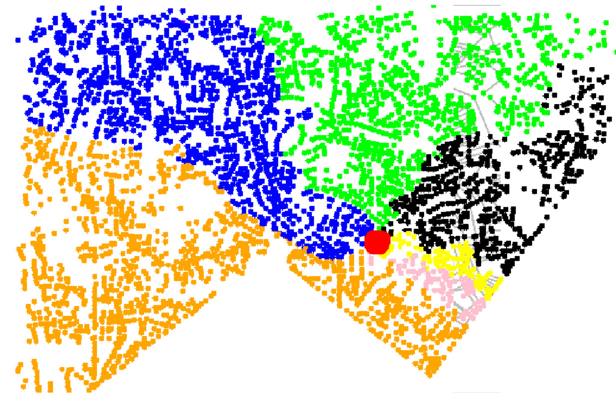
R-tree



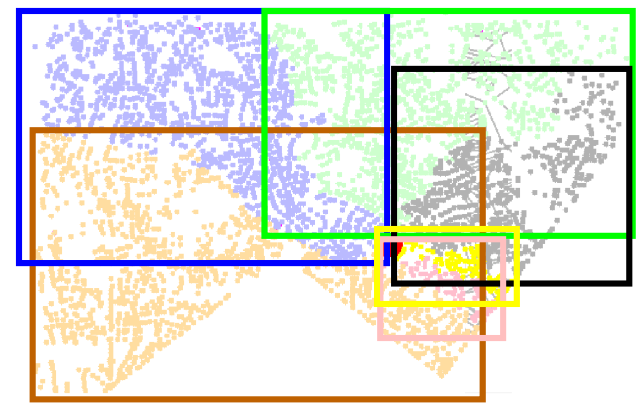
Shortest-Path Quadtree

How to Store Colored Regions?

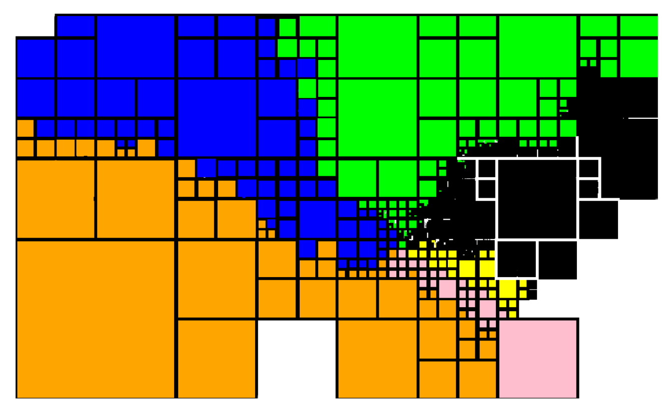
- Minimum bounding boxes (e.g., R-tree) [Wagn03]
  - overlapping boxes imply identity of next vertex cannot be uniquely determined causing the shortest path algorithm to possibly degenerate to Dijkstra's algorithm
- Disjoint decomposition: shortest-path quadtree
  - Decompose until all vertices in block have the same color
- Shortest-path quadtree stored as a collection of Morton blocks
  - Note: no need to store identity of vertices in the blocks



Shortest-path Map



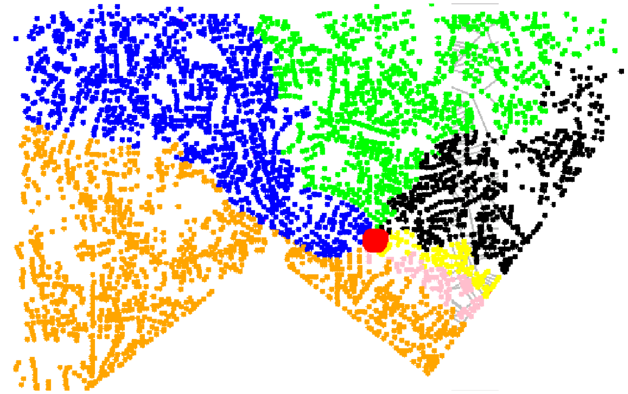
R-tree



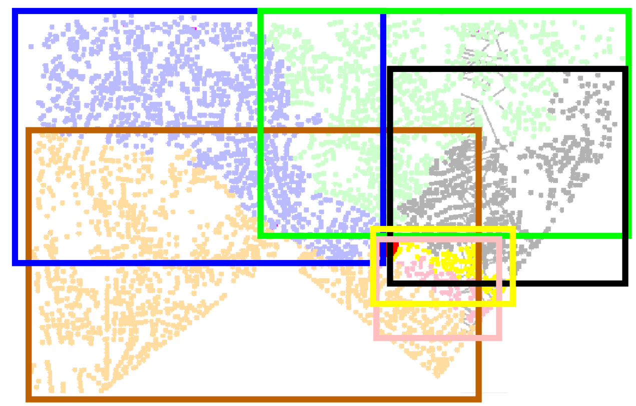
Shortest-Path Quadtree

How to Store Colored Regions?

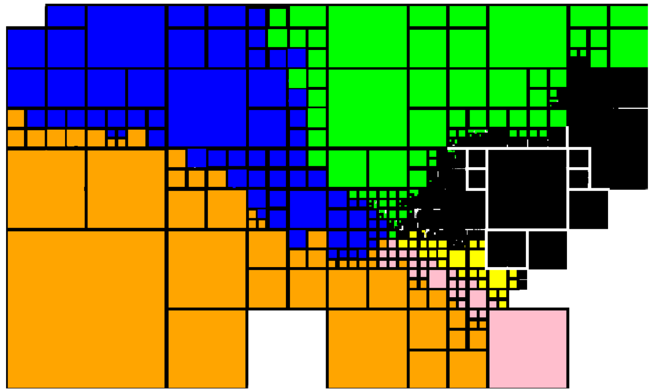
- Minimum bounding boxes (e.g., R-tree) [Wagn03]
  - overlapping boxes imply identity of next vertex cannot be uniquely determined causing the shortest path algorithm to possibly degenerate to Dijkstra's algorithm
- Disjoint decomposition: shortest-path quadtree
  - Decompose until all vertices in block have the same color
- Shortest-path quadtree stored as a collection of Morton blocks
  - Note: no need to store identity of vertices in the blocks
- Proposed encoding leverages the dimensionality reduction property of MX and region quadtrees
  - Required storage cost to represent a region R in a region and MX quadtree is $O(p)$, where p is the perimeter of R



Shortest-path Map



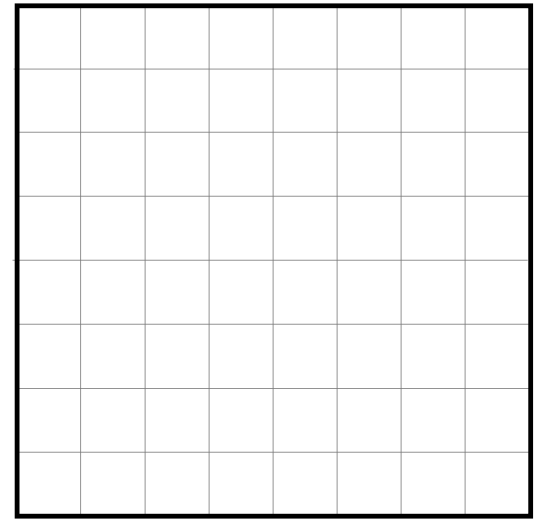
R-tree



Shortest-Path Quadtree

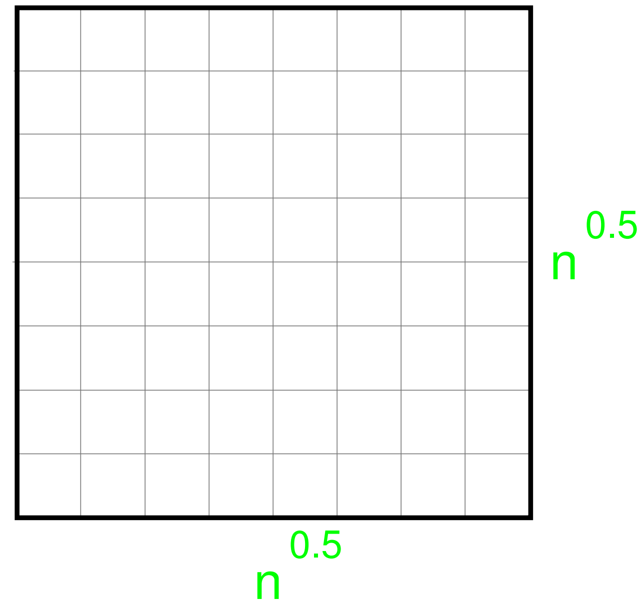
Space Complexity Analysis of Shortest-Path Quadtrees

- Consider a spatial network containing N vertices



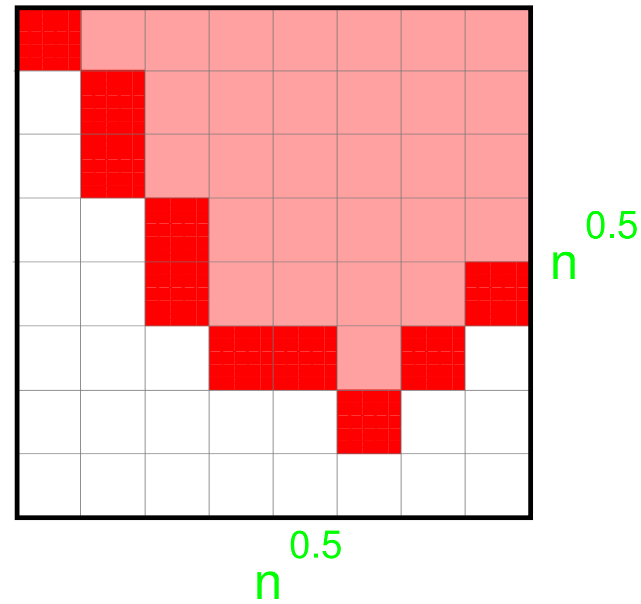
Space Complexity Analysis of Shortest-Path Quadtrees

- Consider a spatial network containing N vertices in a square grid of size $N^{0.5} \times N^{0.5}$ and embed it



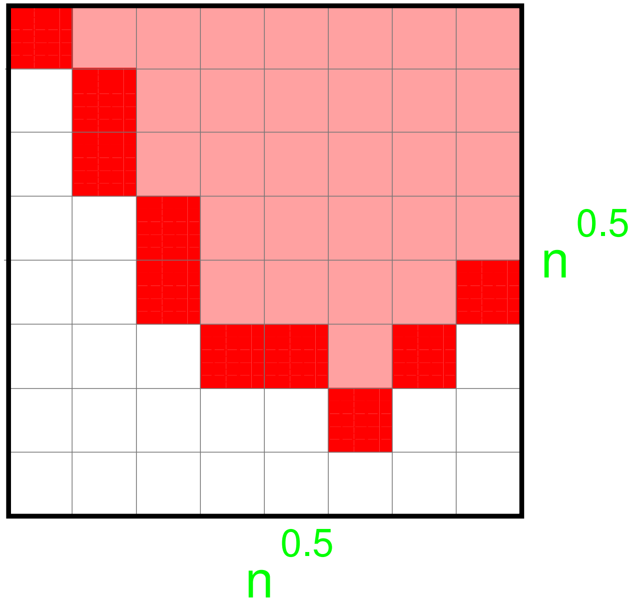
Space Complexity Analysis of Shortest-Path Quadtrees

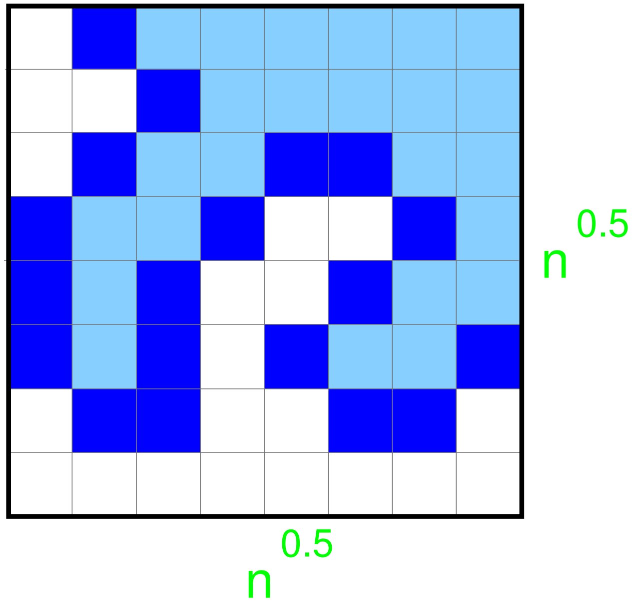
- Consider a spatial network containing N vertices in a square grid of size $N^{0.5} \times N^{0.5}$ and embed it
- Perimeter of a region with monotonic boundary on one of its coordinates is of size $O(N^{0.5})$



Space Complexity Analysis of Shortest-Path Quadtrees

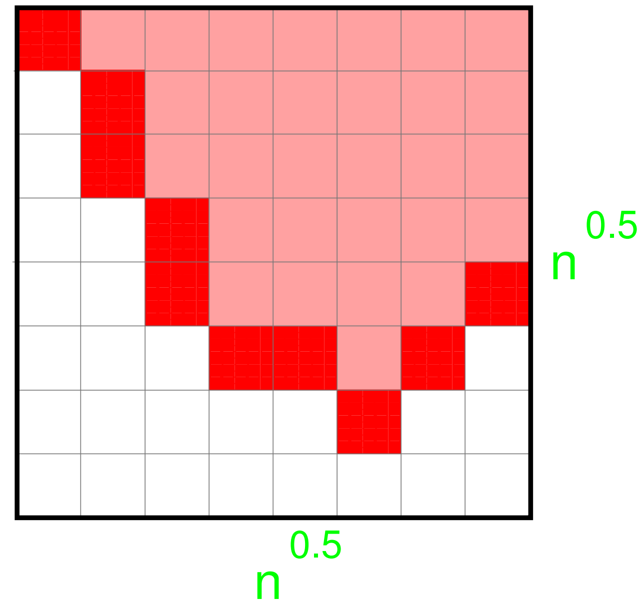
- Consider a spatial network containing N vertices in a square grid of size $N^{0.5} \times N^{0.5}$ and embed it
- Perimeter of a region with monotonic boundary on one of its coordinates is of size $O(N^{0.5})$
- Perimeter of a region with a non-monotonic boundary can be of size $O(N)$

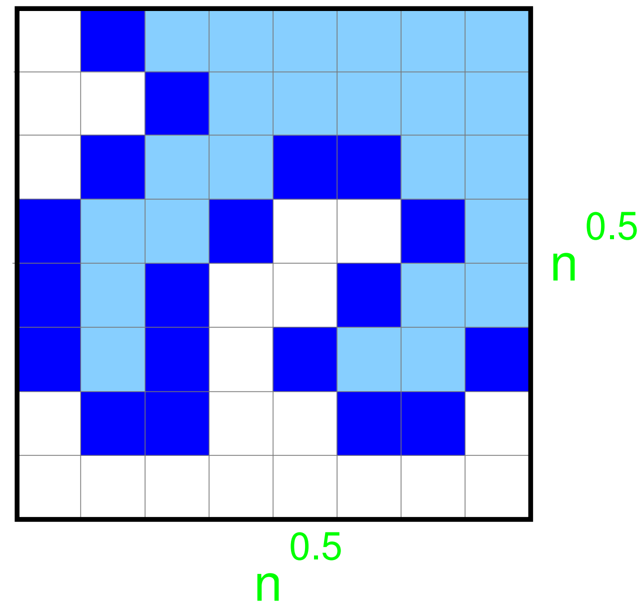




Space Complexity Analysis of Shortest-Path Quadtrees

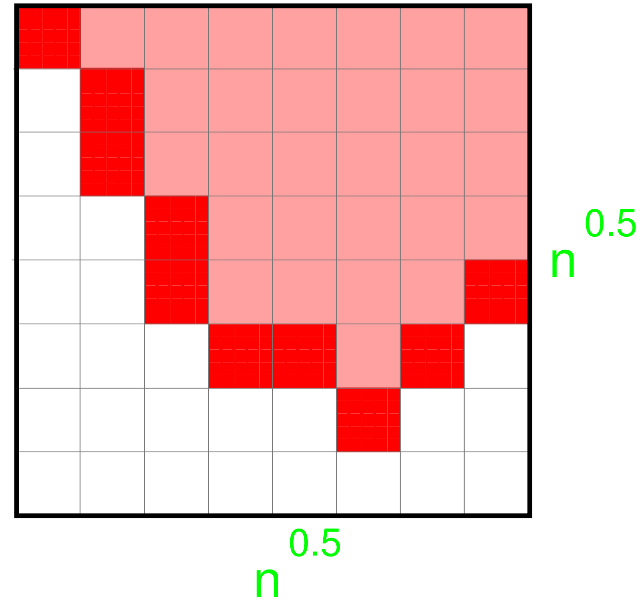
- Consider a spatial network containing N vertices in a square grid of size $N^{0.5} \times N^{0.5}$ and embed it
- Perimeter of a region with monotonic boundary on one of its coordinates is of size $O(N^{0.5})$
- Perimeter of a region with a non-monotonic boundary can be of size $O(N)$
- Assumption: Regions of the shortest-path quadtree have *monotonic* boundaries

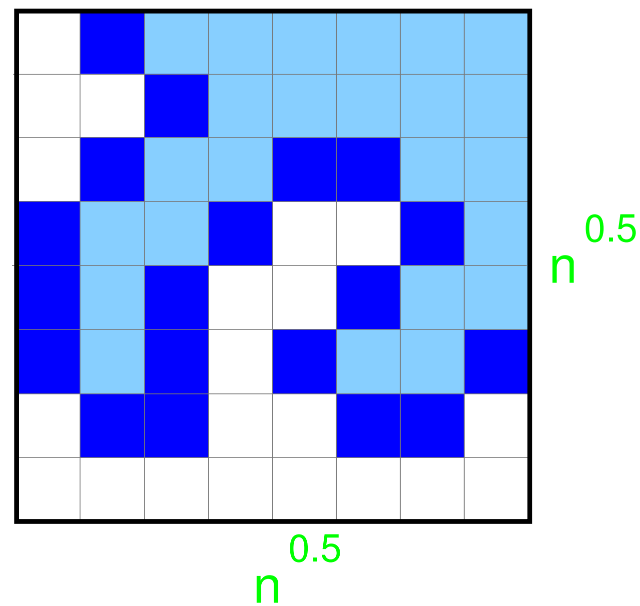




Space Complexity Analysis of Shortest-Path Quadtrees

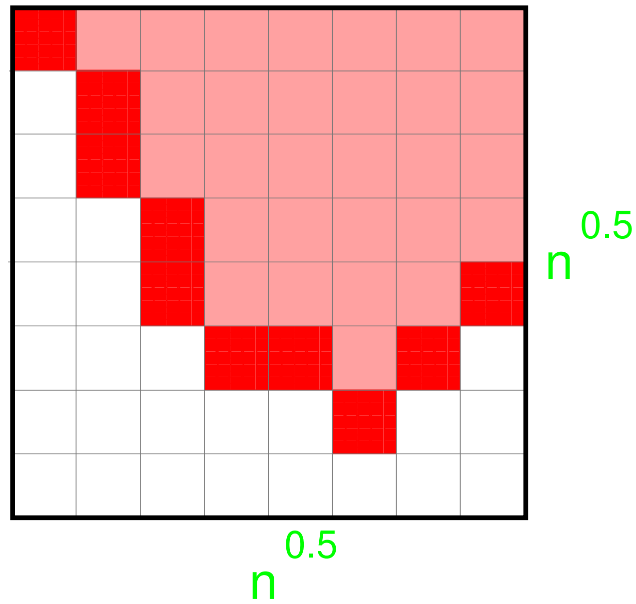
- Consider a spatial network containing N vertices in a square grid of size $N^{0.5} \times N^{0.5}$ and embed it
- Perimeter of a region with <u>monotonic</u> boundary on one of its coordinates is of size $O(N^{0.5})$
- Perimeter of a region with a <u>non-monotonic</u> boundary can be of size $O(N)$
- Assumption: Regions of the shortest-path quadtree have *monotonic* boundaries
- Size of a shortest-path quadtree of a vertex u is $c\sqrt{N}$

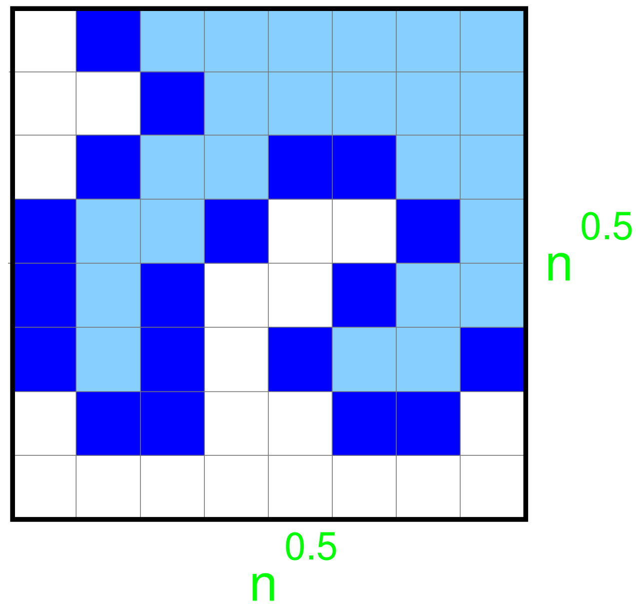




Space Complexity Analysis of Shortest-Path Quadtrees

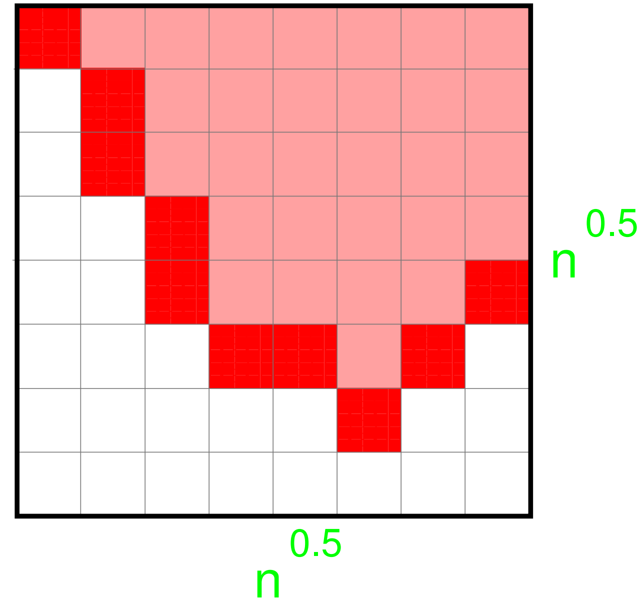
- Consider a spatial network containing N vertices in a square grid of size $N^{0.5} \times N^{0.5}$ and embed it
- Perimeter of a region with monotonic boundary on one of its coordinates is of size $O(N^{0.5})$
- Perimeter of a region with a non-monotonic boundary can be of size $O(N)$
- Assumption: Regions of the shortest-path quadtree have *monotonic* boundaries
- Size of a shortest-path quadtree of a vertex u is $c\sqrt{N}$, where c is a function of the outdegree of u

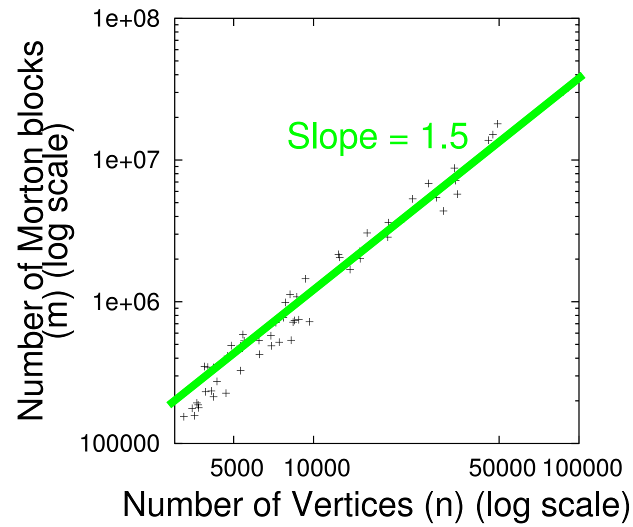




Space Complexity Analysis of Shortest-Path Quadtrees

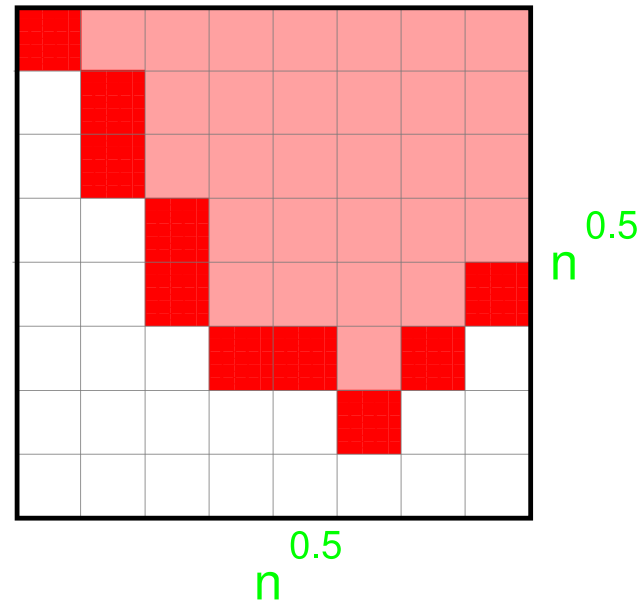
- Consider a spatial network containing N vertices in a square grid of size $N^{0.5} \times N^{0.5}$ and embed it
- Perimeter of a region with monotonic boundary on one of its coordinates is of size $O(N^{0.5})$
- Perimeter of a region with a non-monotonic boundary can be of size $O(N)$
- Assumption: Regions of the shortest-path quadtree have *monotonic* boundaries
- Size of a shortest-path quadtree of a vertex u is $c\sqrt{N}$, where c is a function of the outdegree of u
- Total storage complexity of the SILC framework is $O(N\sqrt{N})$; closely follows empirical results

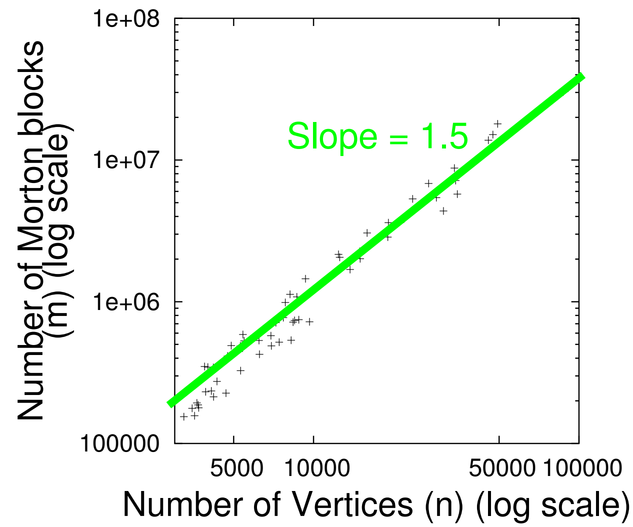




Space Complexity Analysis of Shortest-Path Quadtrees

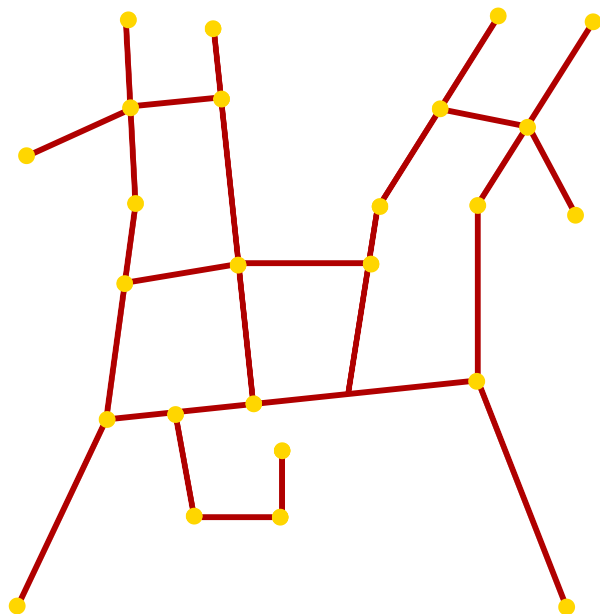
- Consider a spatial network containing N vertices in a square grid of size $N^{0.5} \times N^{0.5}$ and embed it
- Perimeter of a region with monotonic boundary on one of its coordinates is of size $O(N^{0.5})$
- Perimeter of a region with a non-monotonic boundary can be of size $O(N)$
- Assumption: Regions of the shortest-path quadtree have *monotonic* boundaries
- Size of a shortest-path quadtree of a vertex u is $c\sqrt{N}$, where c is a function of the outdegree of u
- Total storage complexity of the SILC framework is $O(N\sqrt{N})$; closely follows empirical results
- Contribution: A mechanism to capture shortest paths in spatial networks based solely on geometry and independent of topology or connectivity





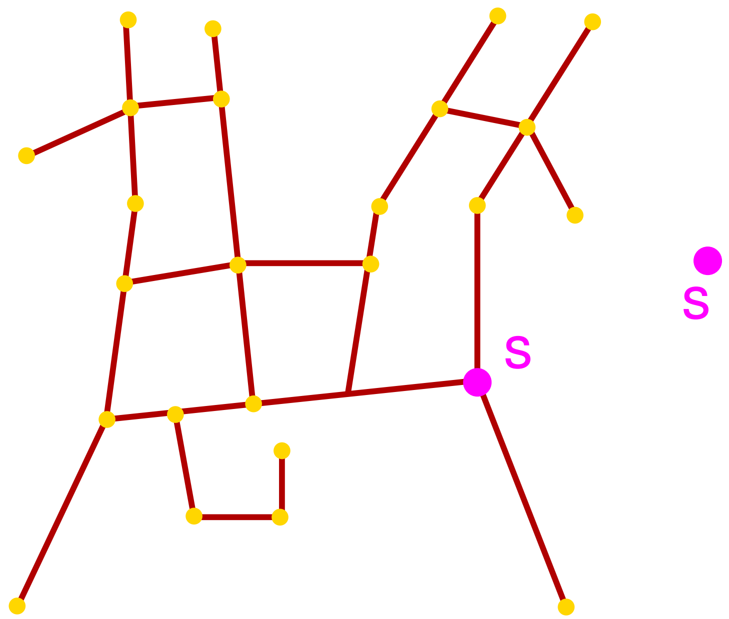
Path Retrieval

- Problem: How to retrieve the shortest path from a



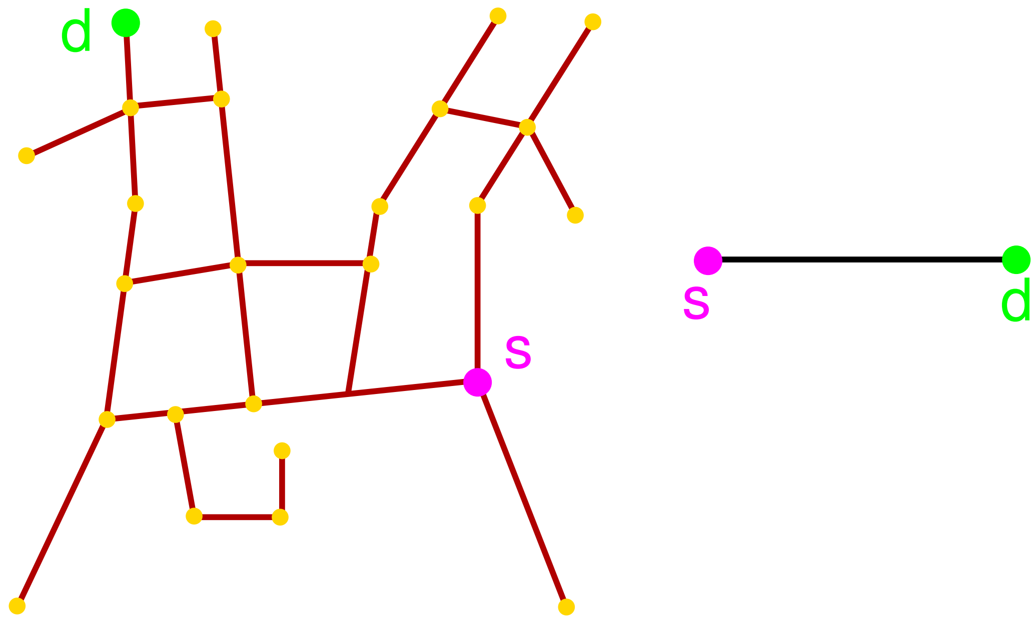
Path Retrieval

- Problem: How to retrieve the shortest path from a source s



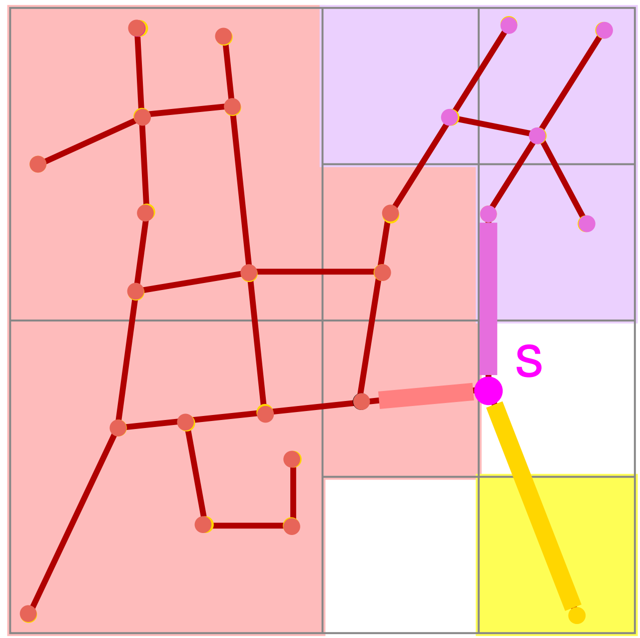
Path Retrieval

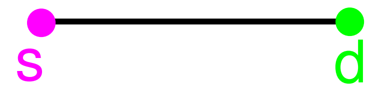
- Problem: How to retrieve the shortest path from a source s to a destination d?



Path Retrieval

- Problem: How to retrieve the shortest path from a source s to a destination d?

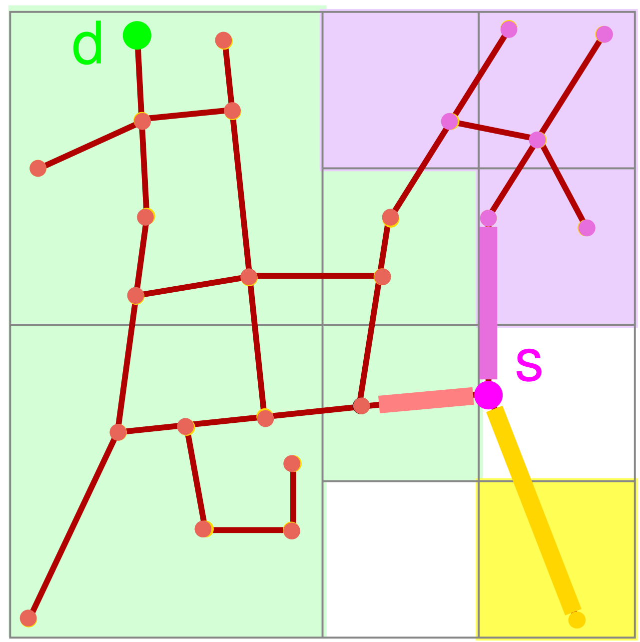


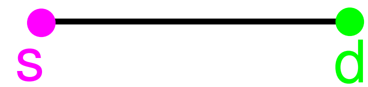


- Retrieve the shortest-path quadtree Q_s corresponding to s

Path Retrieval

- Problem: How to retrieve the shortest path from a source s to a destination d?

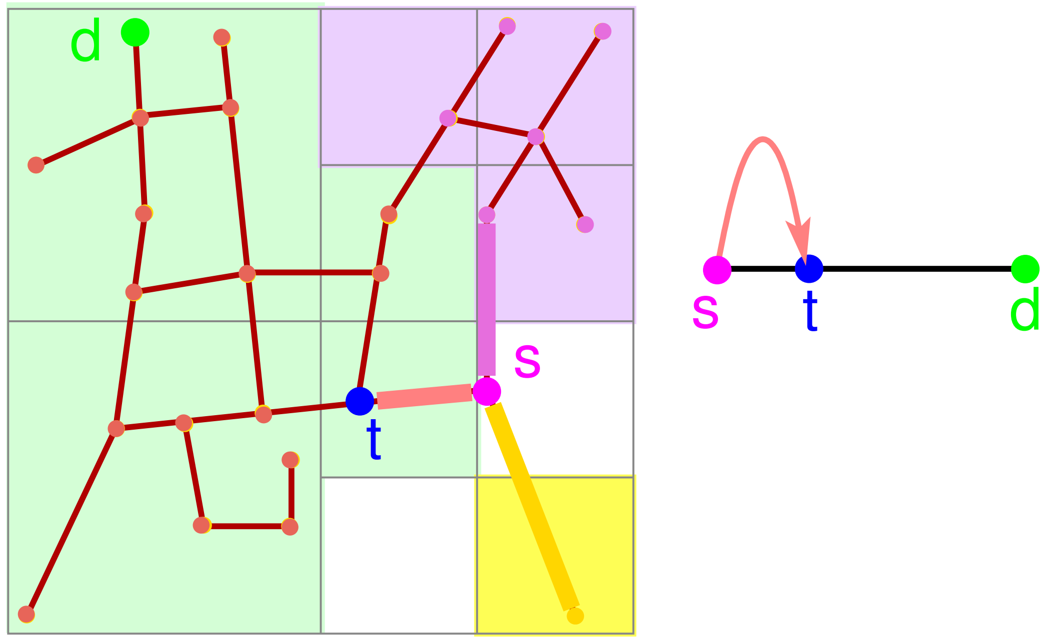




- Retrieve the shortest-path quadtree Q_s corresponding to s
- Find the colored region that contains d in Q_s

Path Retrieval

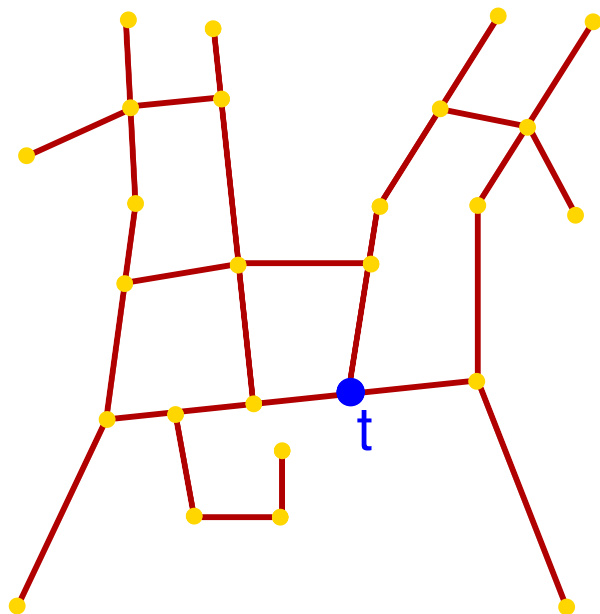
- Problem: How to retrieve the shortest path from a source s to a destination d?

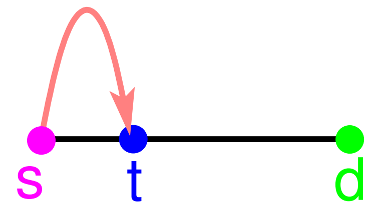


- Retrieve the shortest-path quadtree Q_s corresponding to s
- Find the colored region that contains d in Q_s
- Retrieve the vertex t connected to s in the region containing d in Q_s

Path Retrieval

- Problem: How to retrieve the shortest path from a source s to a destination d?

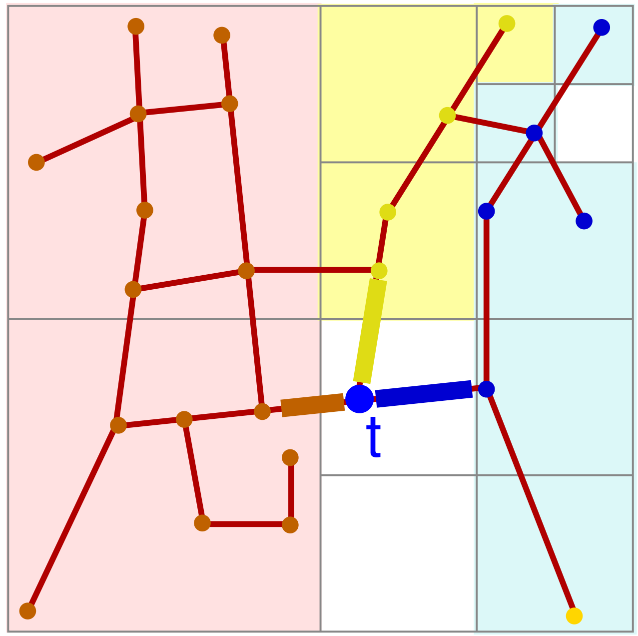


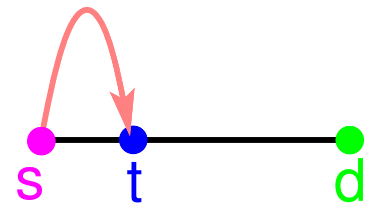


- Retrieve the shortest-path quadtree Q_s corresponding to s
- Find the colored region that contains d in Q_s
- Retrieve the vertex t connected to s in the region containing d in Q_s

Path Retrieval

- Problem: How to retrieve the shortest path from a source s to a destination d?

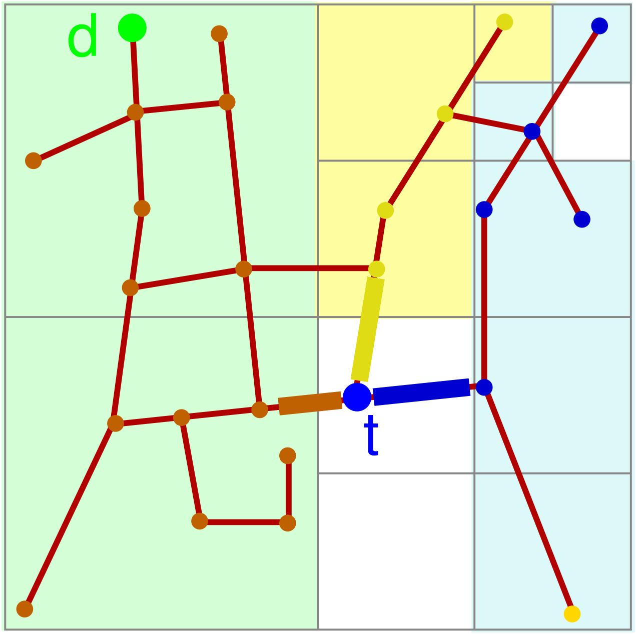


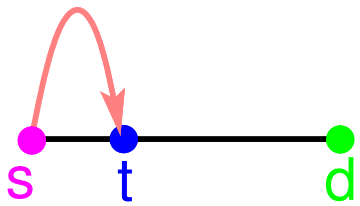


- Retrieve the shortest-path quadtree Q_s corresponding to s
- Find the colored region that contains d in Q_s
- Retrieve the vertex t connected to s in the region containing d in Q_s
- Retrieve the shortest-path quadtree Q_t corresponding to t

Path Retrieval

- Problem: How to retrieve the shortest path from a source s to a destination d?

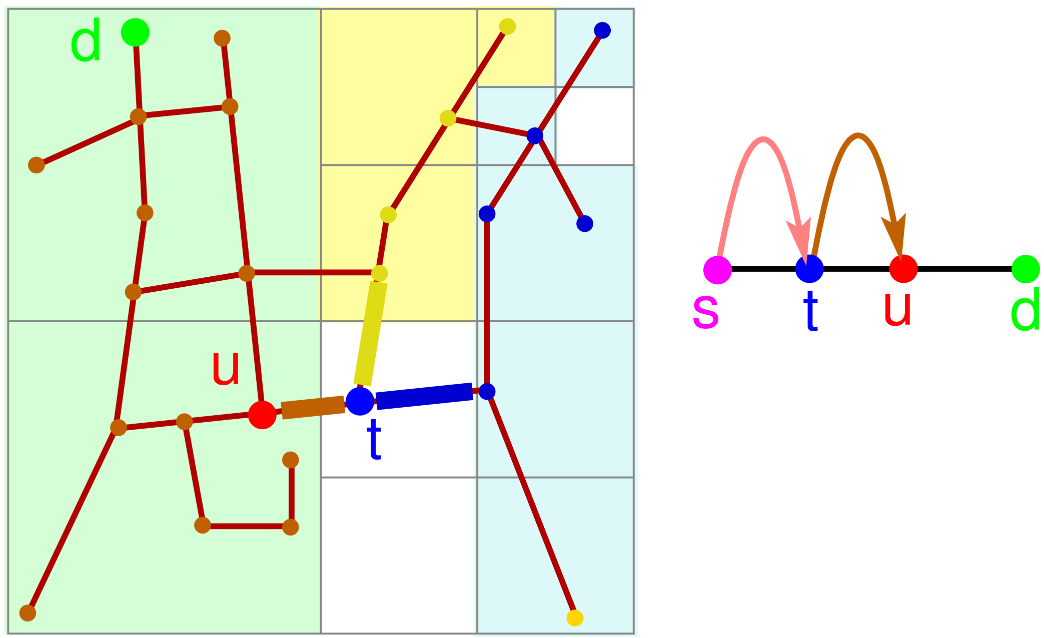




- Retrieve the shortest-path quadtree Q_s corresponding to s
- Find the colored region that contains d in Q_s
- Retrieve the vertex t connected to s in the region containing d in Q_s
- Retrieve the shortest-path quadtree Q_t corresponding to t
- Find the colored region that contains d in Q_t

Path Retrieval

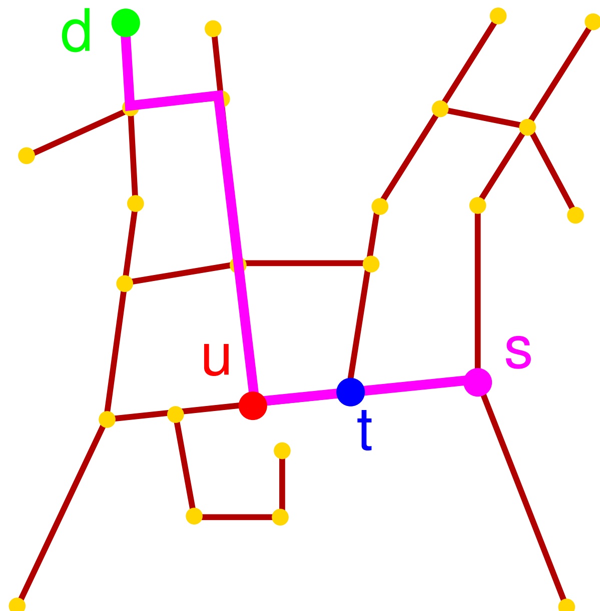
- Problem: How to retrieve the shortest path from a source s to a destination d?

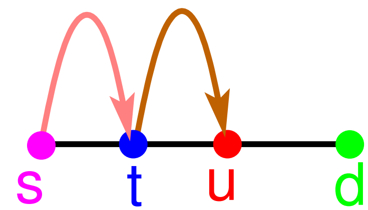


- Retrieve the shortest-path quadtree Q_s corresponding to s
- Find the colored region that contains d in Q_s
- Retrieve the vertex t connected to s in the region containing d in Q_s
- Retrieve the shortest-path quadtree Q_t corresponding to t
- Find the colored region that contains d in Q_t
- Retrieve the vertex u connected to t in the region containing d in Q_t

Path Retrieval

- Problem: How to retrieve the shortest path from a source s to a destination d?

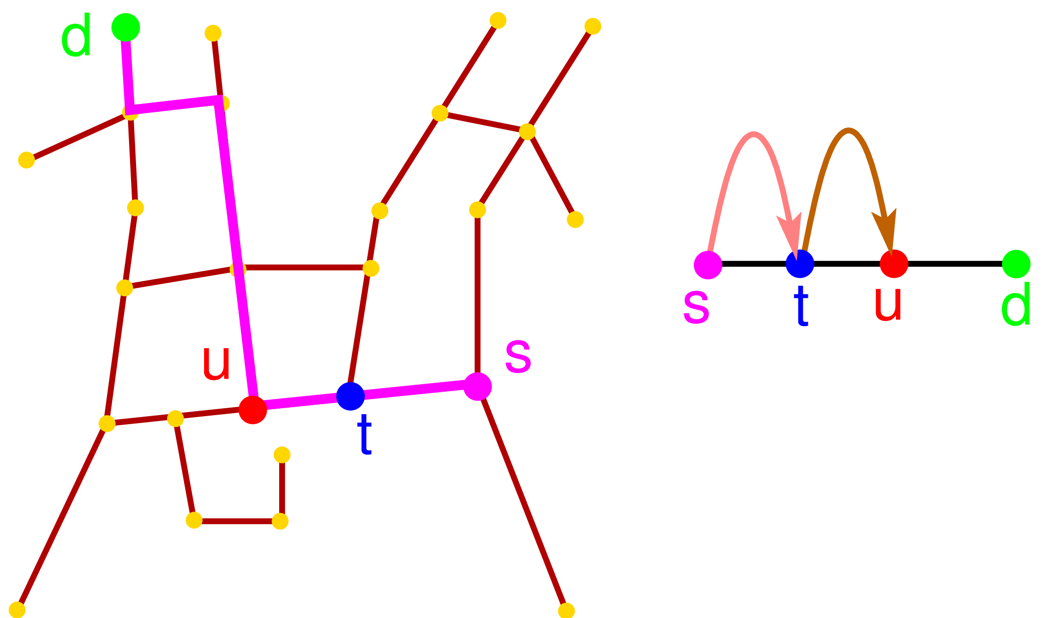




  - Retrieve the shortest-path quadtree Q_s corresponding to s
  - Find the colored region that contains d in Q_s
  - Retrieve the vertex t connected to s in the region containing d in Q_s
  - Retrieve the shortest-path quadtree Q_t corresponding to t
  - Find the colored region that contains d in Q_t
  - Retrieve the vertex u connected to t in the region containing d in Q_t
- Entire shortest path can be retrieved in size-of-path steps

Path Retrieval

- Problem: How to retrieve the shortest path from a source s to a destination d?



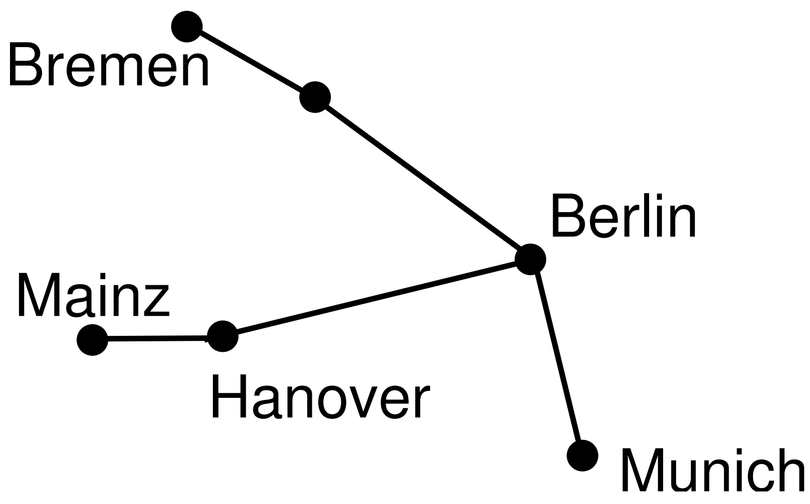
  - Retrieve the shortest-path quadtree Q_s corresponding to s
  - Find the colored region that contains d in Q_s
  - Retrieve the vertex t connected to s in the region containing d in Q_s
  - Retrieve the shortest-path quadtree Q_t corresponding to t
  - Find the colored region that contains d in Q_t
  - Retrieve the vertex u connected to t in the region containing d in Q_t
- Entire shortest path can be retrieved in size-of-path steps
- Network distance between s and d is immediately obtained from shortest path

Progressive Refinement of Distances

- Avoid full shortest path retrievals using progressive refinement
- Idea: Use distance intervals instead of the exact distance
- Progressive refinement: *Improve* interval if query cannot be answered
  - Associate Min/Max distance information with each Morton block
  - Refinement involves finding the next link in the shortest path
  - Worst case: retrieve entire shortest path to answer query
  - Many queries require distance comparison primitives

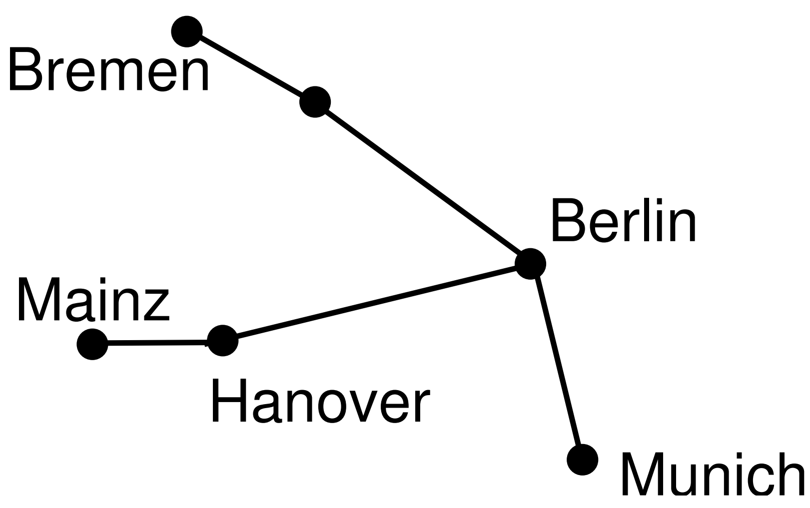
Progressive Refinement of Distances

- Avoid full shortest path retrievals using progressive refinement
- Idea: Use distance intervals instead of the exact distance
- Progressive refinement: *Improve* interval if query cannot be answered
  - Associate Min/Max distance information with each Morton block
  - Refinement involves finding the next link in the shortest path
  - Worst case: retrieve entire shortest path to answer query
  - Many queries require distance comparison primitives
- Example: Is Munich closer to Mainz than Bremen?



Progressive Refinement of Distances

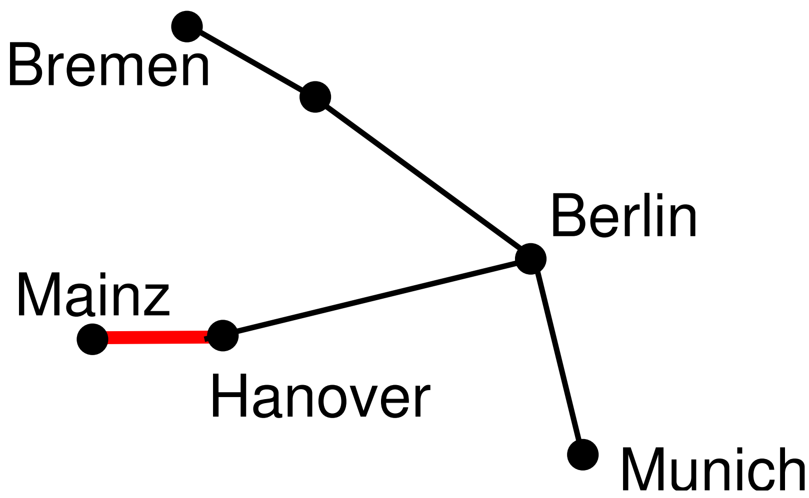
- Avoid full shortest path retrievals using progressive refinement
- Idea: Use distance intervals instead of the exact distance
- Progressive refinement: *Improve* interval if query cannot be answered
    - Associate Min/Max distance information with each Morton block
    - Refinement involves finding the next link in the shortest path
    - Worst case: retrieve entire shortest path to answer query
    - Many queries require distance comparison primitives
- Example: Is Munich closer to Mainz than Bremen?



	Munich	Bremen
Mainz	[10,20]	[15,30]

Progressive Refinement of Distances

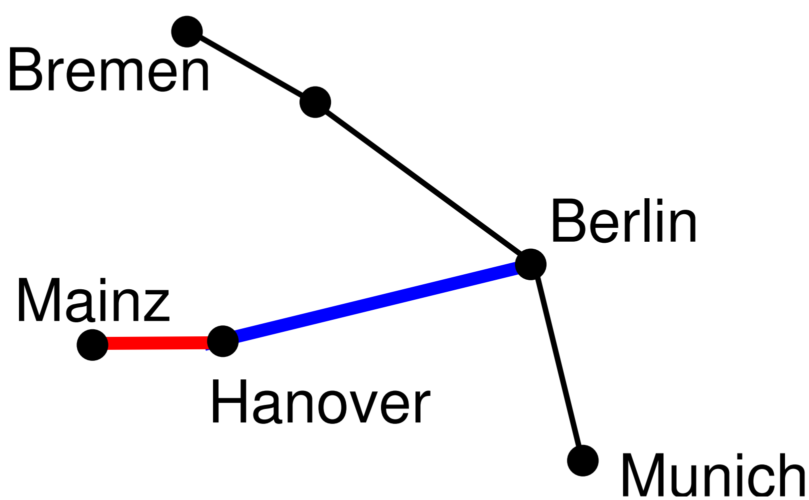
- Avoid full shortest path retrievals using progressive refinement
- Idea: Use distance intervals instead of the exact distance
- Progressive refinement: *Improve* interval if query cannot be answered
  - Associate Min/Max distance information with each Morton block
  - Refinement involves finding the next link in the shortest path
  - Worst case: retrieve entire shortest path to answer query
  - Many queries require distance comparison primitives
- Example: Is Munich closer to Mainz than Bremen?



	Munich	Bremen
Mainz	[10,20]	[15,30]
Hanover	[12,18]	[17,20]

Progressive Refinement of Distances

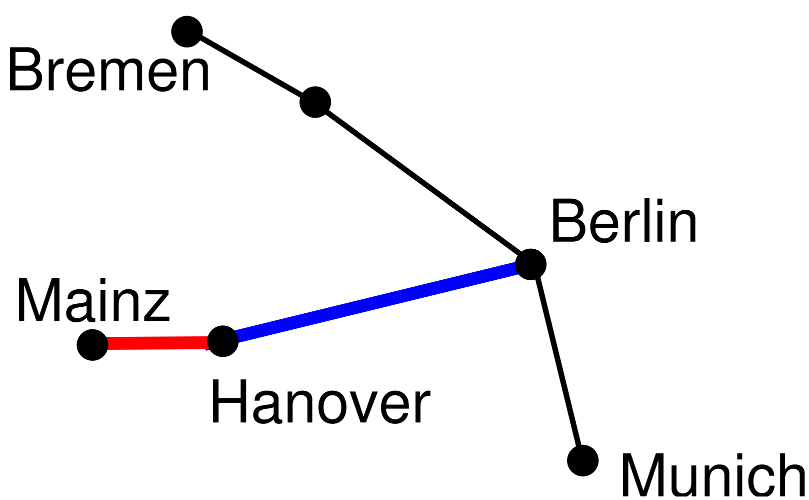
- Avoid full shortest path retrievals using progressive refinement
- Idea: Use distance intervals instead of the exact distance
- Progressive refinement: *Improve* interval if query cannot be answered
  - Associate Min/Max distance information with each Morton block
  - Refinement involves finding the next link in the shortest path
  - Worst case: retrieve entire shortest path to answer query
  - Many queries require distance comparison primitives
- Example: Is Munich closer to Mainz than Bremen?



	Munich	Bremen
Mainz	[10,20]	[15,30]
Hanover	[12,18]	[17,20]
Berlin	[13,15]	[18,19]

Progressive Refinement of Distances

- Avoid full shortest path retrievals using progressive refinement
- Idea: Use distance intervals instead of the exact distance
- Progressive refinement: *Improve* interval if query cannot be answered
  - Associate Min/Max distance information with each Morton block
  - Refinement involves finding the next link in the shortest path
  - Worst case: retrieve entire shortest path to answer query
  - Many queries require distance comparison primitives
- Example: Is Munich closer to Mainz than Bremen?



	Munich	Bremen
Mainz	[10,20]	[15,30]
Hanover	[12,18]	[17,20]
Berlin	[13,15]	[18,19]

- Munich is closer as distance interval via Berlin does not intersect distance interval to Bremen via Berlin

Properties of a Non-Incremental kNN Algorithm

- Neighbors produced in increasing order of distance from q
- Use a priority queue Q of objects and blocks
- Q contains network distance interval $[\delta^-, \delta^+]$ of objects from q
- Additional information stored with each object o in Q
  1. An intermediate vertex u in shortest path from q to u
  2. network distance d from q to u
- Uses another priority queue L in addition to Q
  - Stores k objects found so far in increasing order of δ^+
  - D_k is the maximum of the distance interval of the $k\mathrm{th}$ element in L
  - Idea: Prune elements e from Q such that $\delta_e^- \geq D_k$
- Elements are removed from Q in increasing order of the minimum of their distance interval δ^- from q
  - Objects may be reinserted in Q if $\delta^- < D_k$
  - Terminate when $\delta^- \geq D_k$
- Advantages over Incremental best-first kNN (INN)
  - Smaller size of Q
  - Faster than INN

kNN Algorithm

1. Initialize priority queue Q by inserting the root T
2. Retrieve top element p in Q at each iteration and halt if minimum distance from q is $> D_k$

kNN Algorithm

1. Initialize priority queue Q by inserting the root T
2. Retrieve top element p in Q at each iteration and halt if minimum distance from q is $> D_k$
3. If p is a LEAF block, then replace it with all objects contained within it for which $\delta^- < D_k$ along with their network distance interval from q
   - Also enqueue objects in L if $\delta^+ < D_k$

kNN Algorithm

1. Initialize priority queue Q by inserting the root T
2. Retrieve top element p in Q at each iteration and halt if minimum distance from q is $> D_k$
3. If p is a LEAF block, then replace it with all objects contained within it for which $\delta^- < D_k$ along with their network distance interval from q
   - Also enqueue objects in L if $\delta^+ < D_k$
4. If p is a NONLEAF block, then replace it with all its children blocks for which the minimum distance from q is $< D_k$

kNN Algorithm

1. Initialize priority queue Q by inserting the root T
2. Retrieve top element p in Q at each iteration and halt if minimum distance from q is $> D_k$
3. If p is a LEAF block, then replace it with all objects contained within it for which $\delta^- < D_k$ along with their network distance interval from q
   - Also enqueue objects in L if $\delta^+ < D_k$
4. If p is a NONLEAF block, then replace it with all its children blocks for which the minimum distance from q is $< D_k$
5. If p is an OBJECT, then test the distance interval of p for possible collisions with the current top element of Q

kNN Algorithm

1. Initialize priority queue Q by inserting the root T
2. Retrieve top element p in Q at each iteration and halt if minimum distance from q is $> D_k$
3. If p is a LEAF block, then replace it with all objects contained within it for which $\delta^- < D_k$ along with their network distance interval from q
   - Also enqueue objects in L if $\delta^+ < D_k$
4. If p is a NONLEAF block, then replace it with all its children blocks for which the minimum distance from q is $< D_k$
5. If p is an OBJECT, then test the distance interval of p for possible collisions with the current top element of Q
   - A collision occurs if the distance interval of p intersects the distance interval of the current top element in Q

kNN Algorithm

1. Initialize priority queue Q by inserting the root T
2. Retrieve top element p in Q at each iteration and halt if minimum distance from q is $> D_k$
3. If p is a LEAF block, then replace it with all objects contained within it for which $\delta^- < D_k$ along with their network distance interval from q
   - Also enqueue objects in L if $\delta^+ < D_k$
4. If p is a NONLEAF block, then replace it with all its children blocks for which the minimum distance from q is $< D_k$
5. If p is an OBJECT, then test the distance interval of p for possible collisions with the current top element of Q
   - A collision occurs if the distance interval of p intersects the distance interval of the current top element in Q
   - Collision:

kNN Algorithm

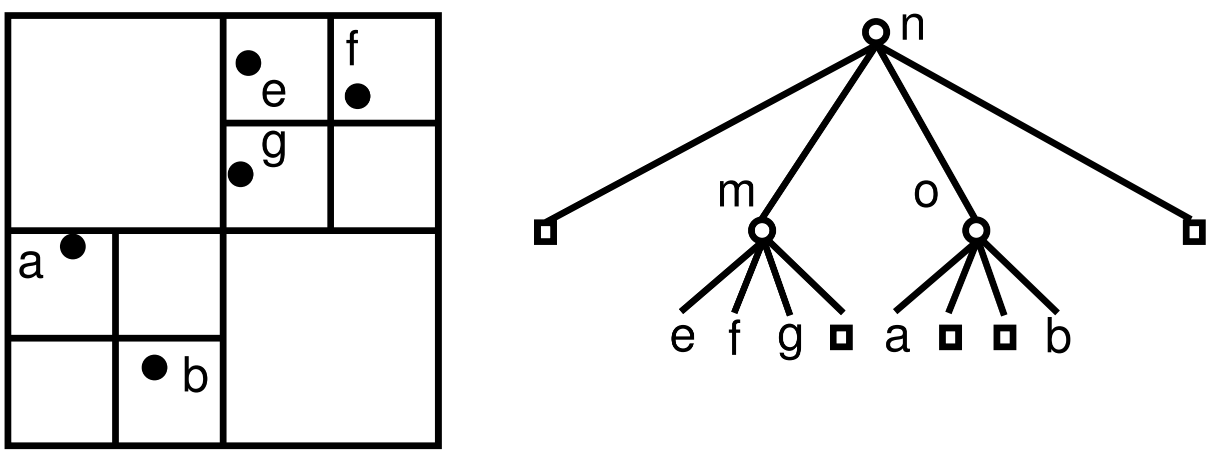
1. Initialize priority queue Q by inserting the root T
2. Retrieve top element p in Q at each iteration and halt if minimum distance from q is $> D_k$
3. If p is a LEAF block, then replace it with all objects contained within it for which $\delta^- < D_k$ along with their network distance interval from q
   - Also enqueue objects in L if $\delta^+ < D_k$
4. If p is a NONLEAF block, then replace it with all its children blocks for which the minimum distance from q is $< D_k$
5. If p is an OBJECT, then test the distance interval of p for possible collisions with the current top element of Q
   - A collision occurs if the distance interval of p intersects the distance interval of the current top element in Q
   - Collision:
     - Remove p from L if $\delta^+ \leq D_k$
     - Apply refinement to improve distance interval of p and reinsert p in L if $\delta^+ \leq D_k$ and in Q if $\delta^- < D_k$ and go to Step 2
   - No collision:

kNN Algorithm

1. Initialize priority queue Q by inserting the root T
2. Retrieve top element p in Q at each iteration and halt if minimum distance from q is $> D_k$
3. If p is a LEAF block, then replace it with all objects contained within it for which $\delta^- < D_k$ along with their network distance interval from q
   - Also enqueue objects in L if $\delta^+ < D_k$
4. If p is a NONLEAF block, then replace it with all its children blocks for which the minimum distance from q is $< D_k$
5. If p is an OBJECT, then test the distance interval of p for possible collisions with the current top element of Q
   - A collision occurs if the distance interval of p intersects the distance interval of the current top element in Q
   - Collision:
     - Remove p from L if $\delta^+ \leq D_k$
     - Apply refinement to improve distance interval of p and reinsert p in L if $\delta^+ \leq D_k$ and in Q if $\delta^- < D_k$ and go to Step 2
   - No collision: p is already one of k nearest neighbors in L (Theorem 1) and go to Step 2

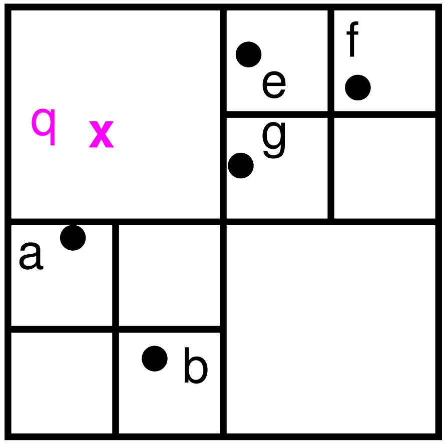
Example of a Non-incremental k Neighbor Search

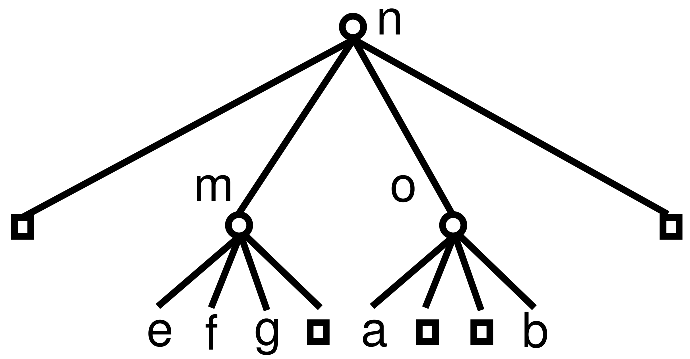
k = 2



Example of a Non-incremental k Neighbor Search

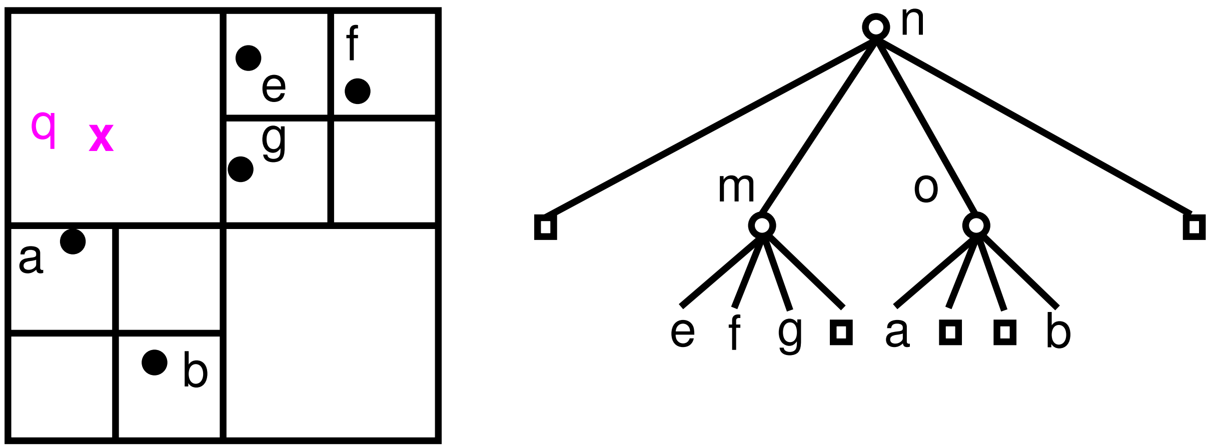
k = 2





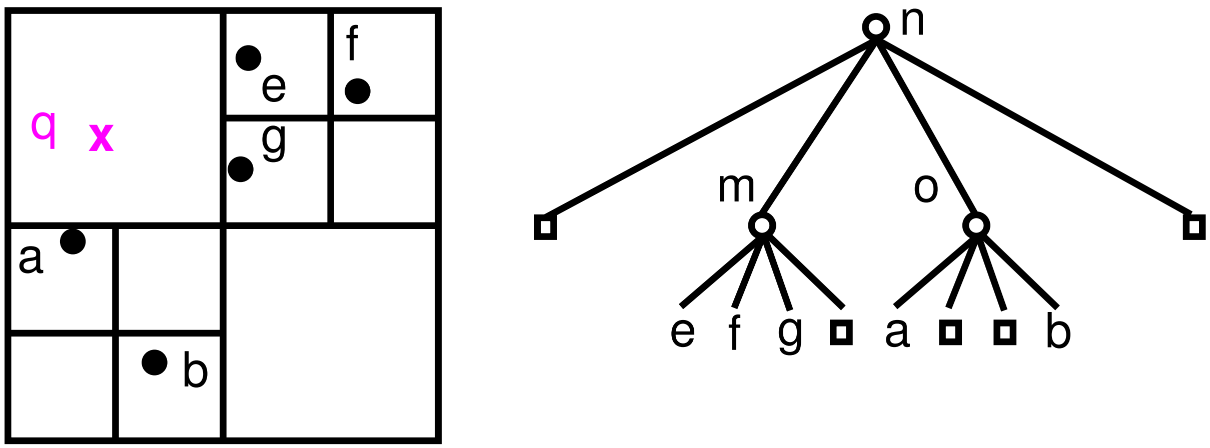
Example of a Non-incremental k Neighbor Search

k = 2

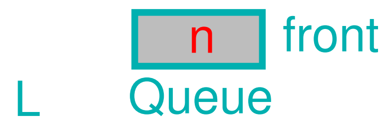


Example of a Non-incremental k Neighbor Search

k = 2

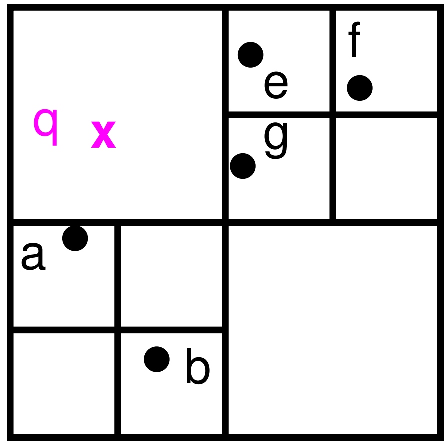


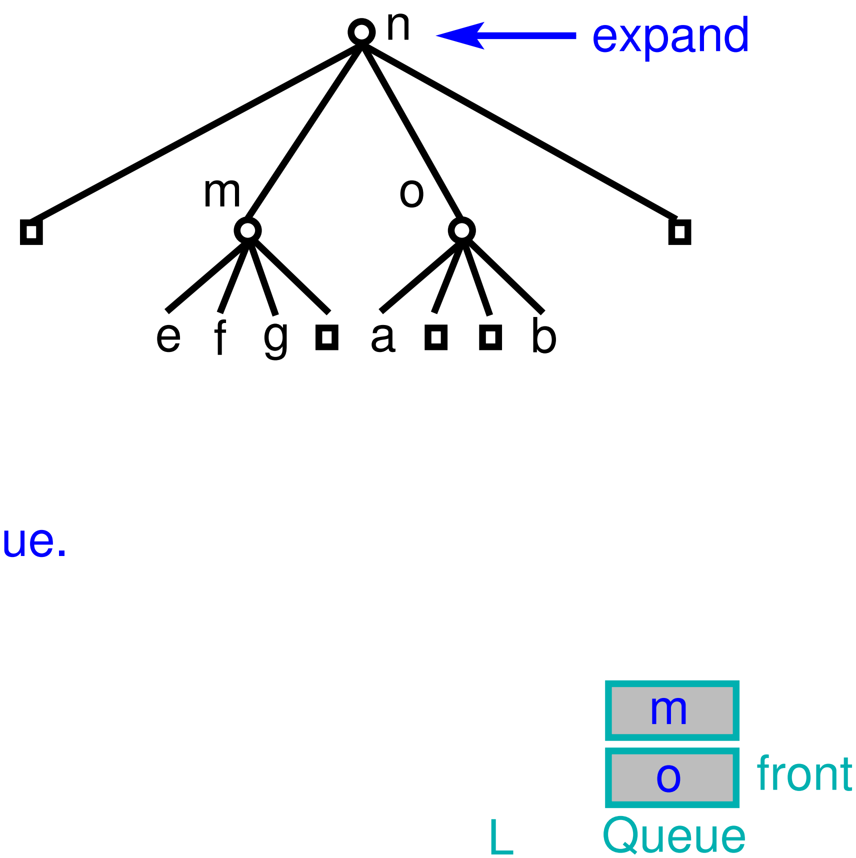
1. Insert n into Queue.



Example of a Non-incremental k Neighbor Search

k = 2

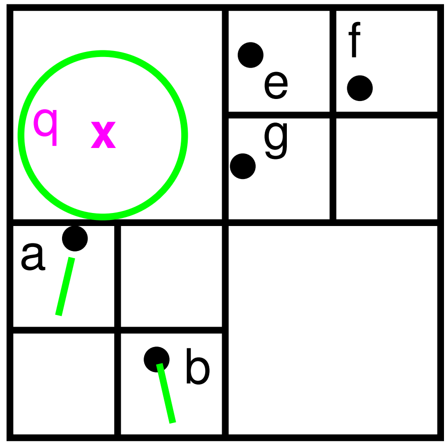


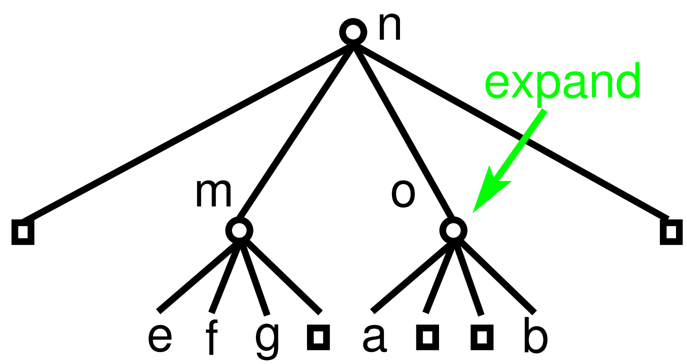


1. Insert n into Queue.
2. Expand n. Insert o,m into Queue.

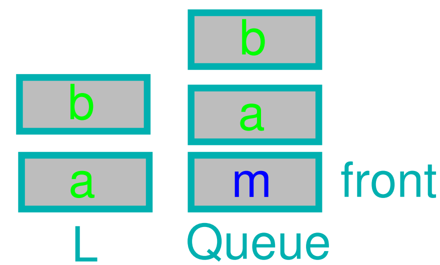
Example of a Non-incremental k Neighbor Search

k = 2

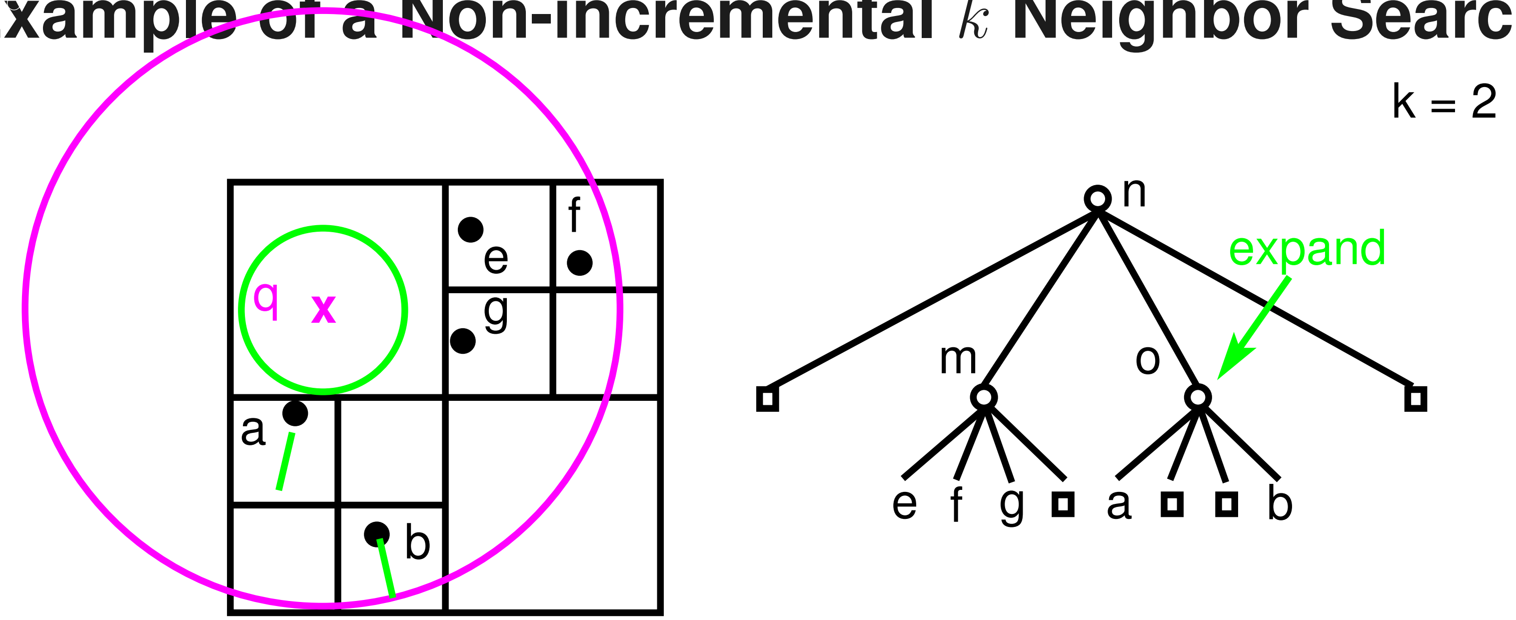




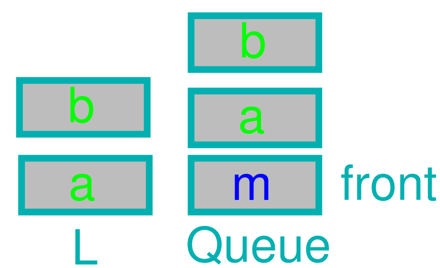
1. Insert n into Queue.
2. Expand n. Insert o,m into Queue.
3. Expand o. Insert a,b into Queue, L.



Example of a Non-incremental k Neighbor Search

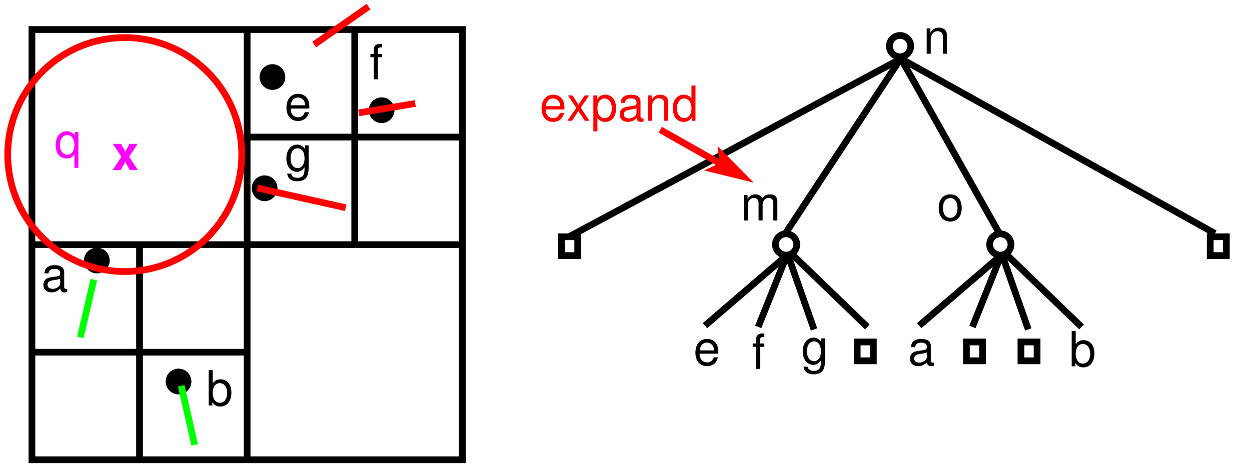


1. Insert n into Queue.
2. Expand n. Insert o,m into Queue.
3. Expand o. Insert a,b into Queue, L. Set D_k.

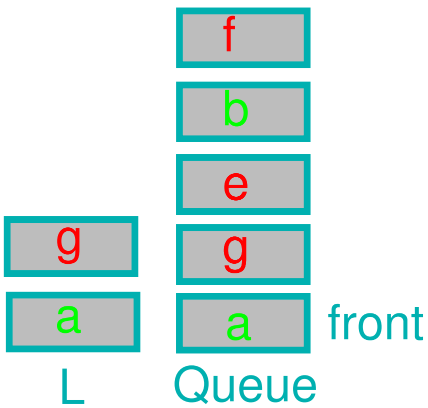


Example of a Non-incremental k Neighbor Search

k = 2

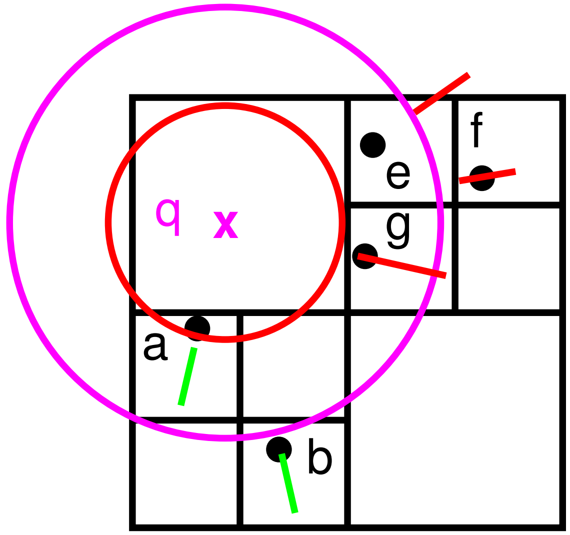


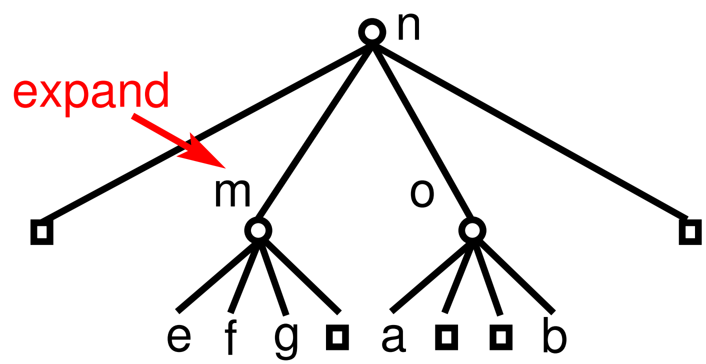
1. Insert n into Queue.
2. Expand n. Insert o,m into Queue.
3. Expand o. Insert a,b into Queue, L. Set D_k.
4. Expand m. Insert g,e,f into Queue and g into L.



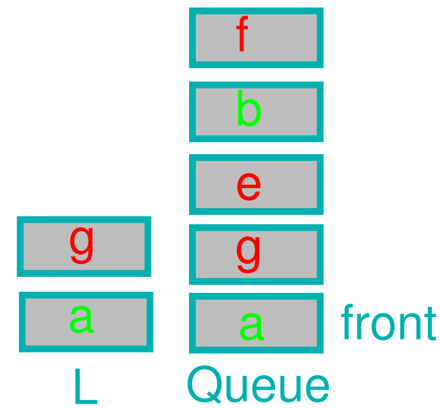
Example of a Non-incremental k Neighbor Search

k = 2



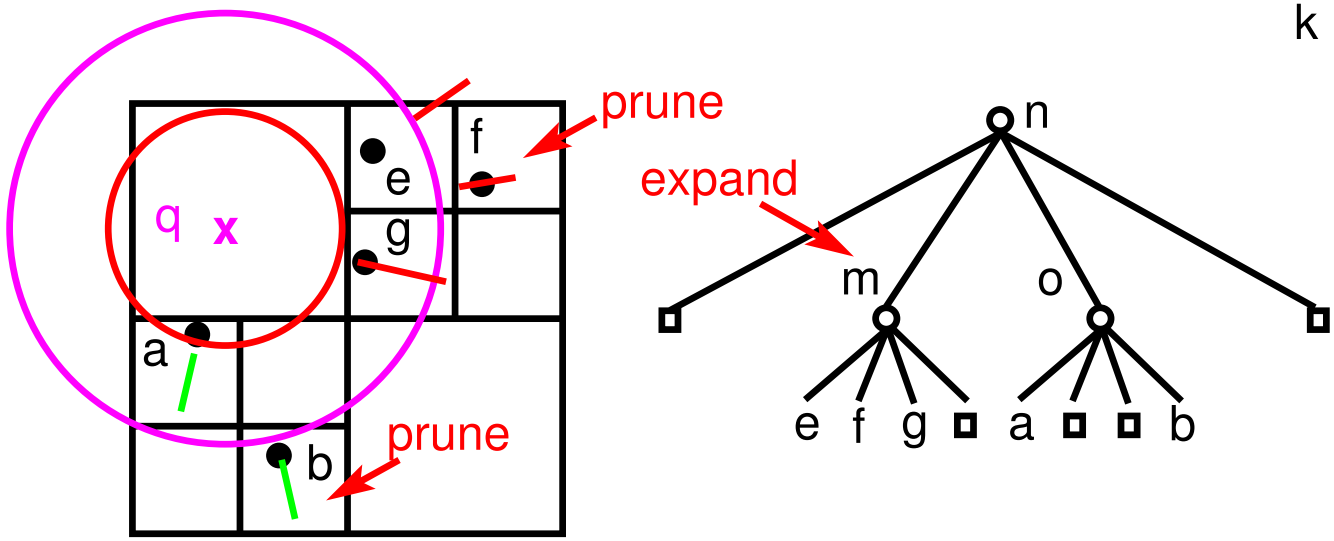


1. Insert n into Queue.
2. Expand n. Insert o,m into Queue.
3. Expand o. Insert a,b into Queue, L. Set D_k.
4. Expand m. Insert g,e,f into Queue and g into L.
   Update D_k.

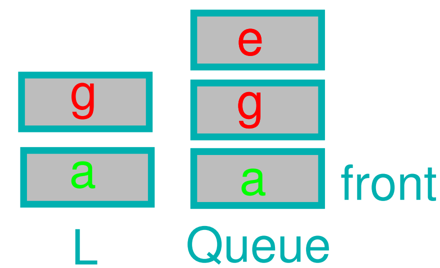


Example of a Non-incremental k Neighbor Search

k = 2

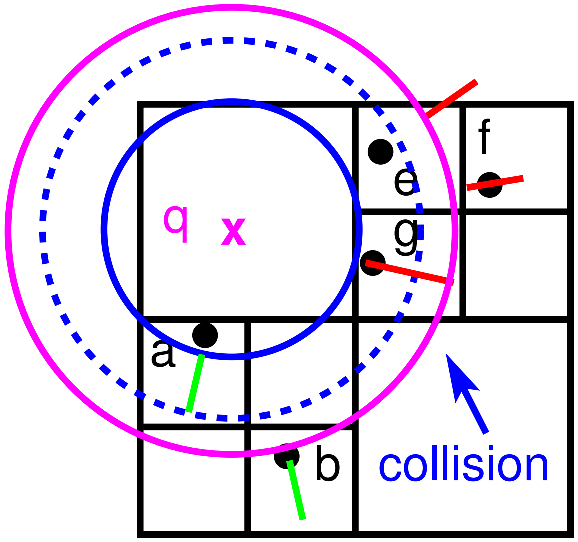


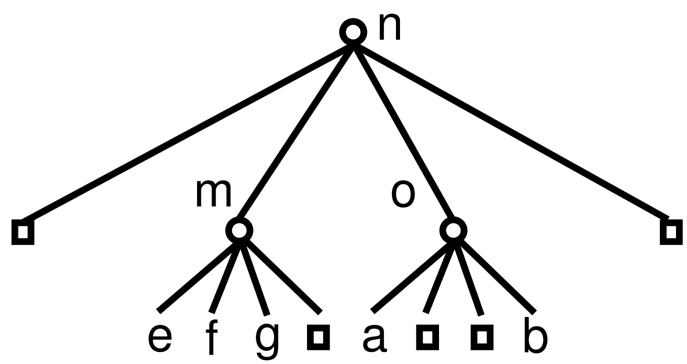
1. Insert n into Queue.
2. Expand n. Insert o,m into Queue.
3. Expand o. Insert a,b into Queue, L. Set D_k.
4. Expand m. Insert g,e,f into Queue and g into L.
   Update D_k. Prune f and b from Queue.



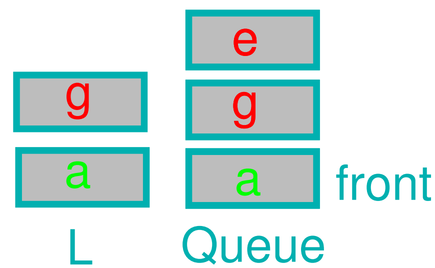
Example of a Non-incremental k Neighbor Search

k = 2



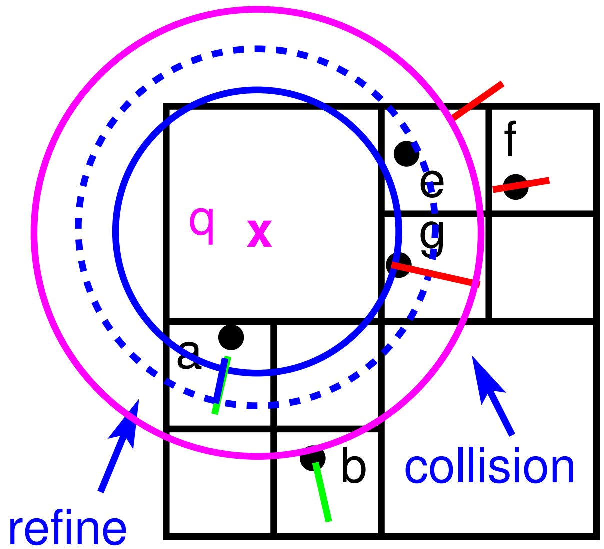


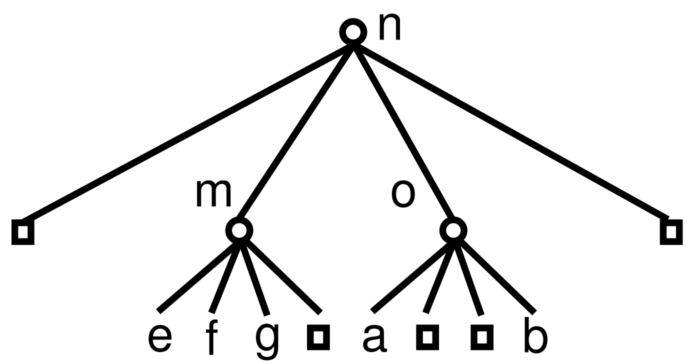
1. Insert n into Queue.
2. Expand n. Insert o,m into Queue.
3. Expand o. Insert a,b into Queue, L. Set D_k.
4. Expand m. Insert g,e,f into Queue and g into L.
   Update D_k. Prune f and b from Queue.
5. Process a. Collision of a with g.



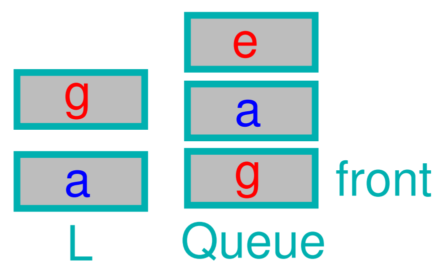
Example of a Non-incremental k Neighbor Search

k = 2



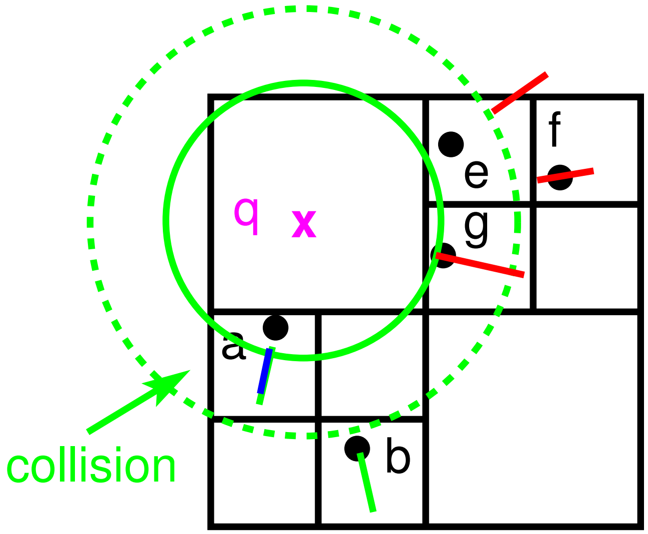


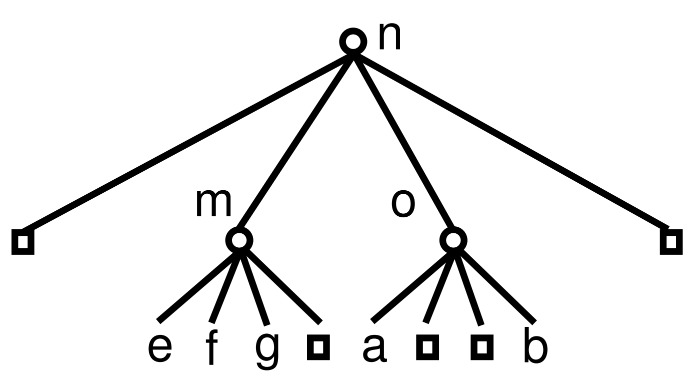
1. Insert n into Queue.
2. Expand n. Insert o,m into Queue.
3. Expand o. Insert a,b into Queue, L. Set D_k.
4. Expand m. Insert g,e,f into Queue and g into L.
   Update D_k. Prune f and b from Queue.
5. Process a. Collision of a with g.
   Refine a. Reinsert a into Queue and L.



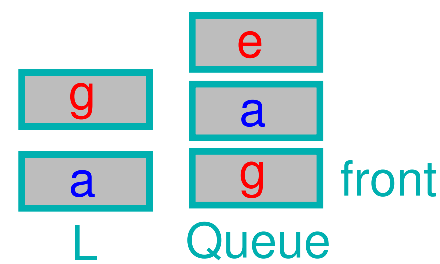
Example of a Non-incremental k Neighbor Search

k = 2



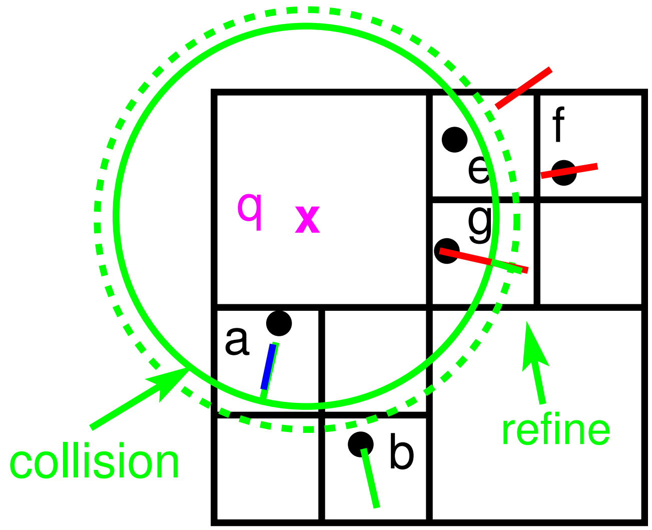


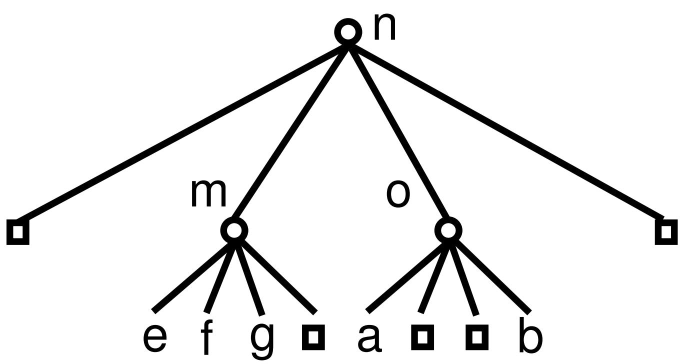
1. Insert n into Queue.
2. Expand n. Insert o,m into Queue.
3. Expand o. Insert a,b into Queue, L. Set D_k.
4. Expand m. Insert g,e,f into Queue and g into L.
   Update D_k. Prune f and b from Queue.
5. Process a. Collision of a with g.
   Refine a. Reinsert a into Queue and L.
6. Process g. Collision of g with a.



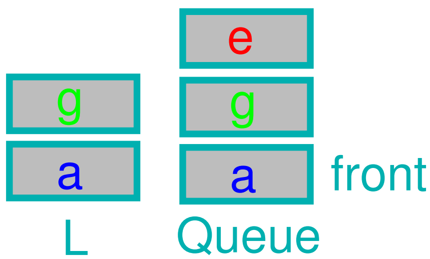
Example of a Non-incremental k Neighbor Search

k = 2



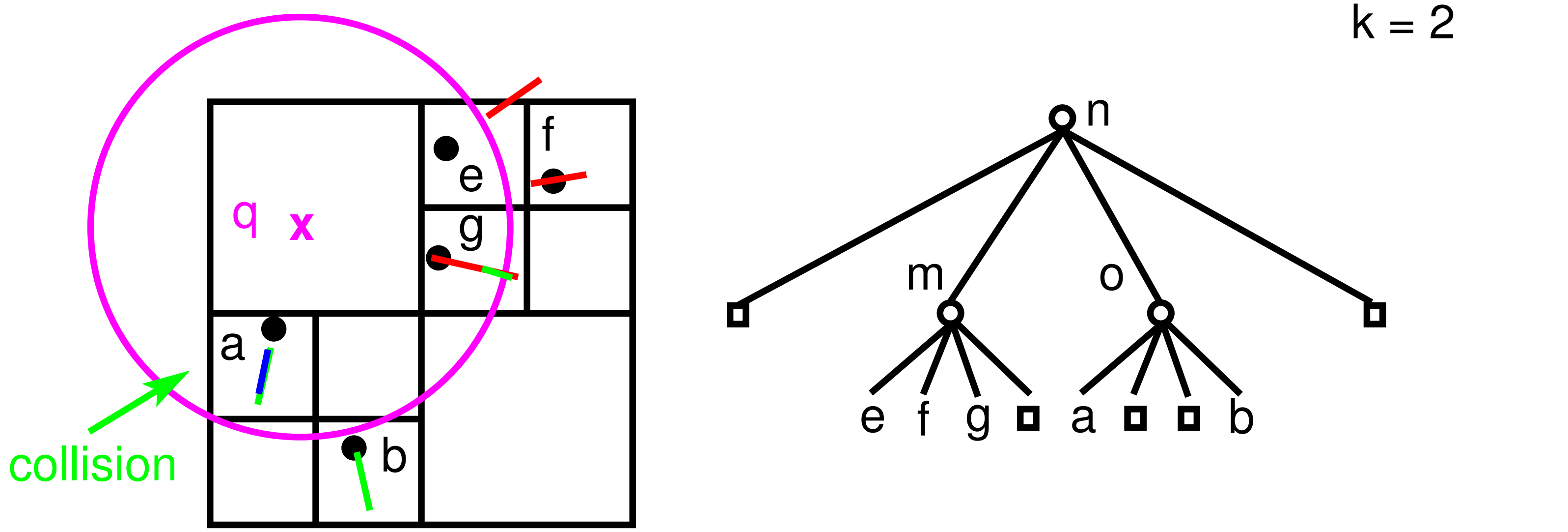


1. Insert n into Queue.
2. Expand n. Insert o,m into Queue.
3. Expand o. Insert a,b into Queue, L. Set D_k.
4. Expand m. Insert g,e,f into Queue and g into L.
   Update D_k. Prune f and b from Queue.
5. Process a. Collision of a with g.
   Refine a. Reinsert a into Queue and L.
6. Process g. Collision of g with a.
   Refine and Reinsert g into Queue and L.

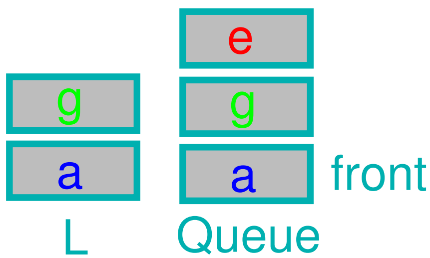


Example of a Non-incremental k Neighbor Search

k = 2

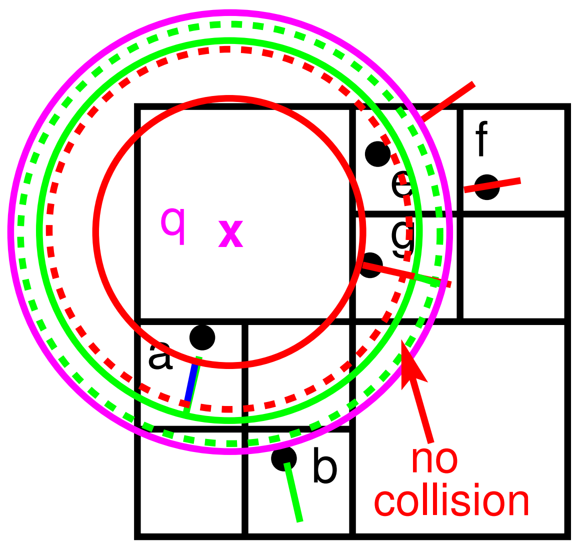


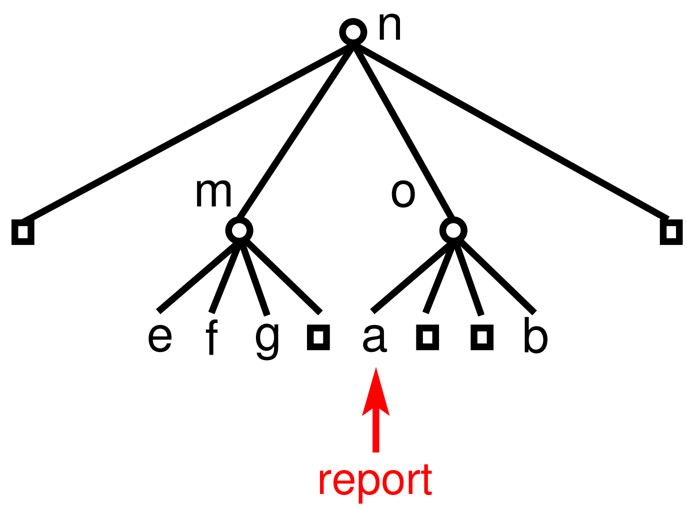
1. Insert n into Queue.
2. Expand n. Insert o,m into Queue.
3. Expand o. Insert a,b into Queue, L. Set D_k.
4. Expand m. Insert g,e,f into Queue and g into L.
   Update D_k. Prune f and b from Queue.
5. Process a. Collision of a with g.
   Refine a. Reinsert a into Queue and L.
6. Process g. Collision of g with a.
   Refine and Reinsert g into Queue and L. Update D_k.



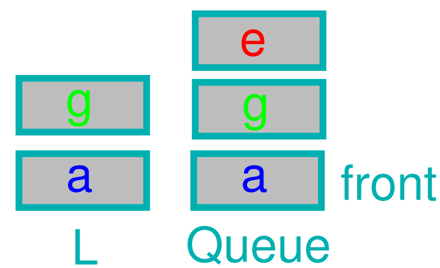
Example of a Non-incremental k Neighbor Search

k = 2

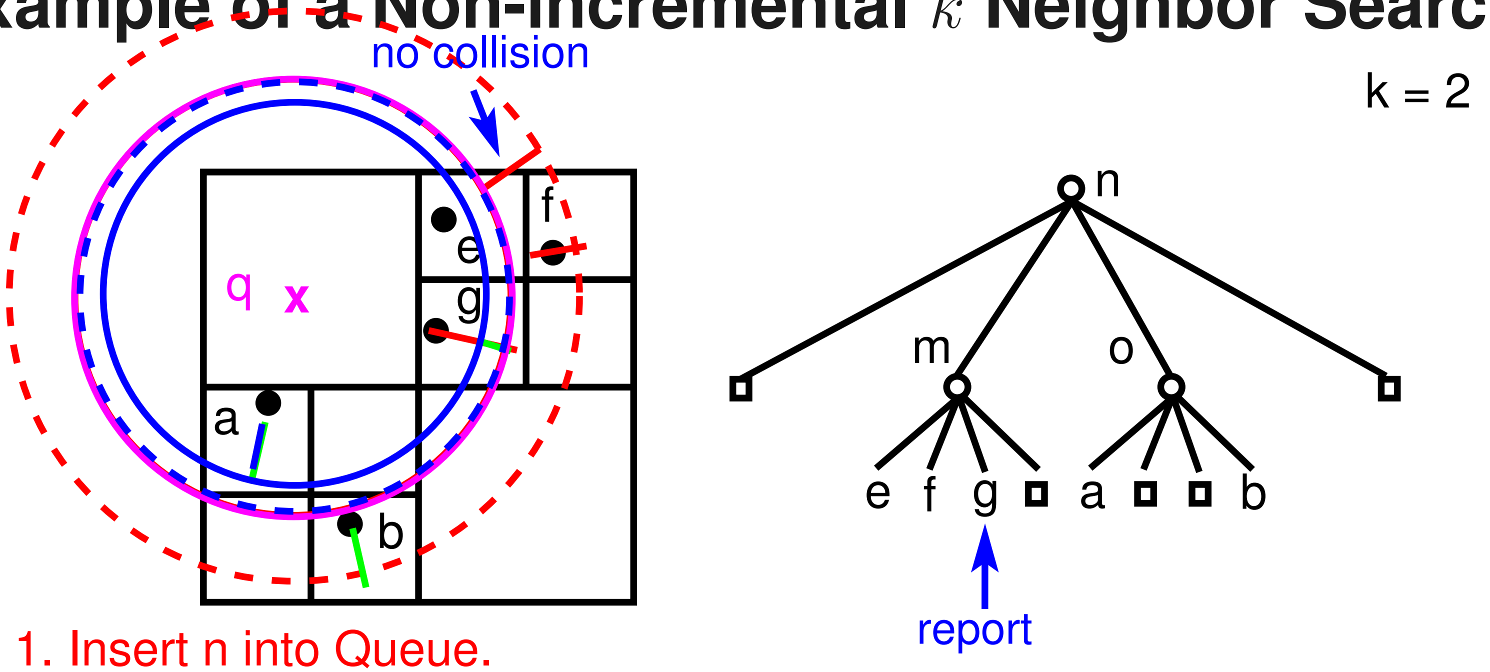




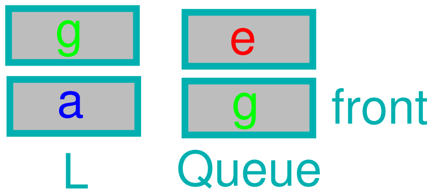
1. Insert n into Queue.
2. Expand n. Insert o,m into Queue.
3. Expand o. Insert a,b into Queue, L. Set D_k.
4. Expand m. Insert g,e,f into Queue and g into L.
   Update D_k. Prune f and b from Queue.
5. Process a. Collision of a with g.
   Refine a. Reinsert a into Queue and L.
6. Process g. Collision of g with a.
   Refine and Reinsert g into Queue and L. Update D_k.
7. Process a. No collision of a with g. No need to refine a further.



Example of a Non-incremental k Neighbor Search

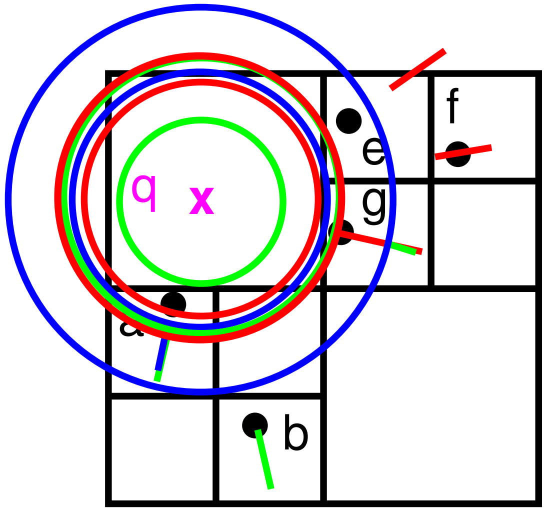


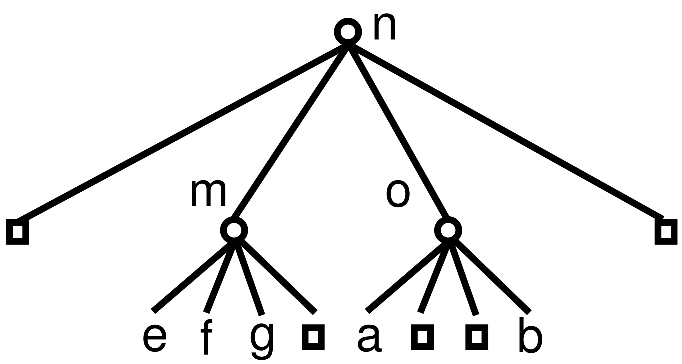
1. Insert n into Queue.
2. Expand n. Insert o,m into Queue.
3. Expand o. Insert a,b into Queue, L. Set D_k.
4. Expand m. Insert g,e,f into Queue and g into L.
   Update D_k. Prune f and b from Queue.
5. Process a. Collision of a with g.
   Refine a. Reinsert a into Queue and L.
6. Process g. Collision of g with a.
   Refine and Reinsert g into Queue and L. Update D_k.
7. Process a. No collision of a with g. No need to refine a further.
8. Process g. No collision of g with e.
   No need to refine g further. Report L.



Example of a Non-incremental k Neighbor Search

k = 2





1. Insert n into Queue.
2. Expand n. Insert o,m into Queue.
3. Expand o. Insert a,b into Queue, L. Set D_k.
4. Expand m. Insert g,e,f into Queue and g into L.
   Update D_k. Prune f and b from Queue.
5. Process a. Collision of a with g.
   Refine a. Reinsert a into Queue and L.
6. Process g. Collision of g with a.
   Refine and Reinsert g into Queue and L. Update D_k.
7. Process a. No collision of a with g. No need to refine a further.
8. Process g. No collision of g with e.
   No need to refine g further. Report L.
   Example of a best–first nearest neighbor algorithm.
   (Search radius to first element in Queue)

front

L Queue

Musings on How Realistic is the Approach

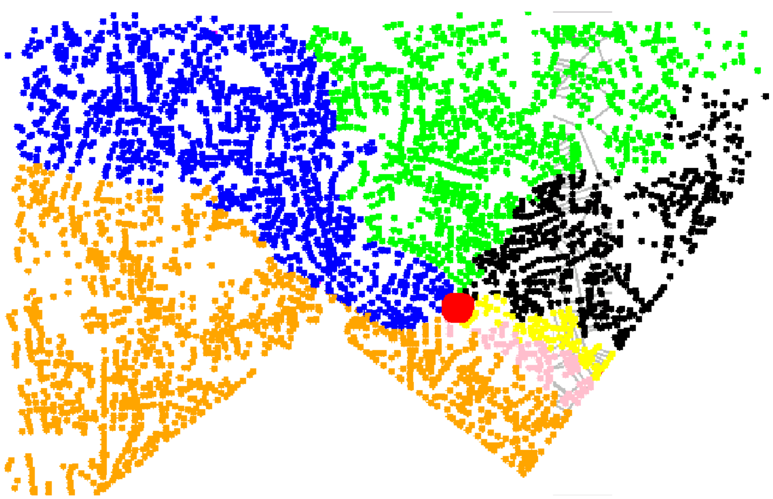
- How about a system for the whole US?
  - 24 million vertices x 10 seconds (say) per shortest path
    - Single machine = 2777 days
    - Google with 0.5 million machines = 480 seconds
    - Modest Cluster of 2000 machines = 1 day, 10 hours
  - Storage shown to be $cN\sqrt{N}$ Morton Blocks
    - N = 24 million vertices, 8 bytes per Morton block, c = 2 from empirical analysis = 1.8 TB
  - Easily Parallelizable: data parallelism
  - Mostly a one-time effort (decoupling)
- Open Challenge: Updates!
  - Changes to spatial network (e.g., road closure)
  - Dynamic traffic information
  - Strategy: How to localize changes to minimize recomputation?
- Approximation Strategies: location based services
  - Shortest-path quadtree on proximal vertices only (say, 100 miles around a vertex)
  - Multiresolution spatial networks
    - Full resolution around a source vertex that gets sparse gradually

Path Coherence Beyond SILC

- The SILC framework captures the path coherence in the shortest paths

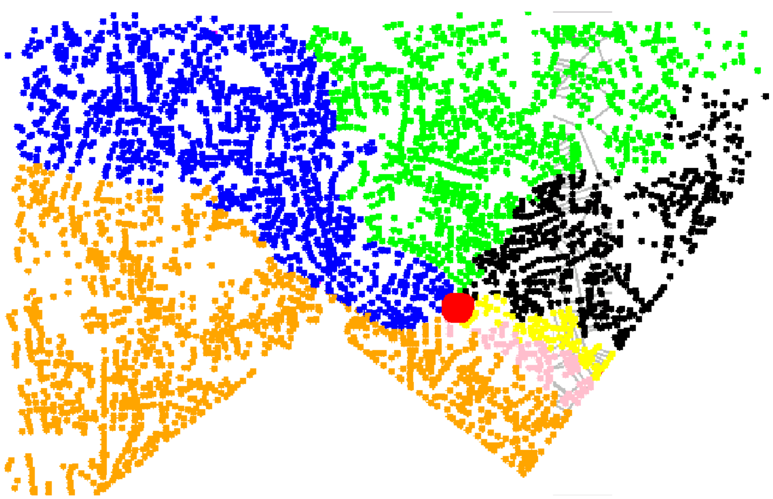
Path Coherence Beyond SILC

- The SILC framework captures the path coherence in the shortest paths
  - Captured: single source vertex to multiple destination vertices



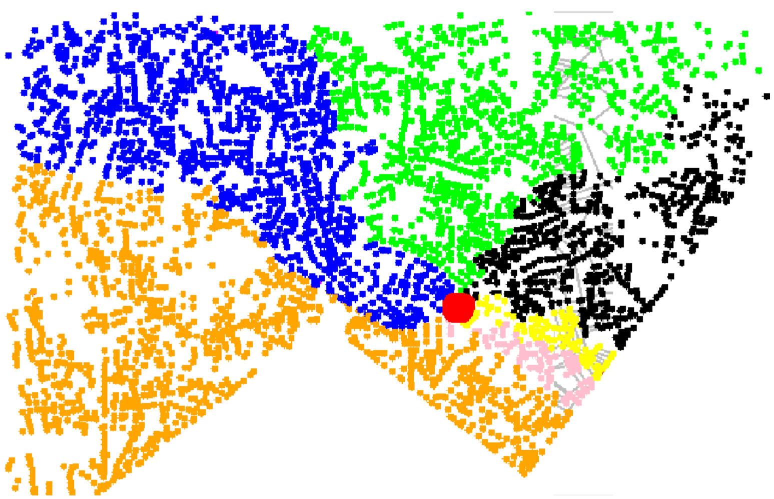
Path Coherence Beyond SILC

- The SILC framework captures the path coherence in the shortest paths
  - Captured: single source vertex to multiple destination vertices
  - Not captured: multiple source vertices to multiple destination vertices



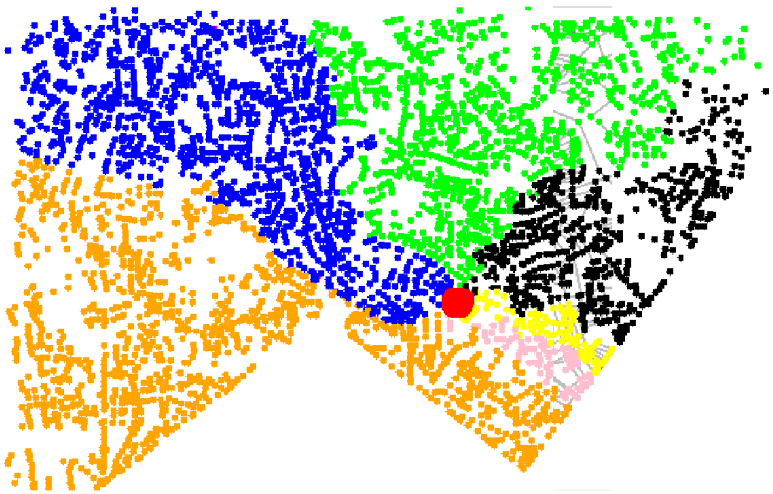
Path Coherence Beyond SILC

- The SILC framework captures the path coherence in the shortest paths
  - Captured: single source vertex to multiple destination vertices
  - Not captured: multiple source vertices to multiple destination vertices
- A new framework: Path Coherent Pairs (PCP)



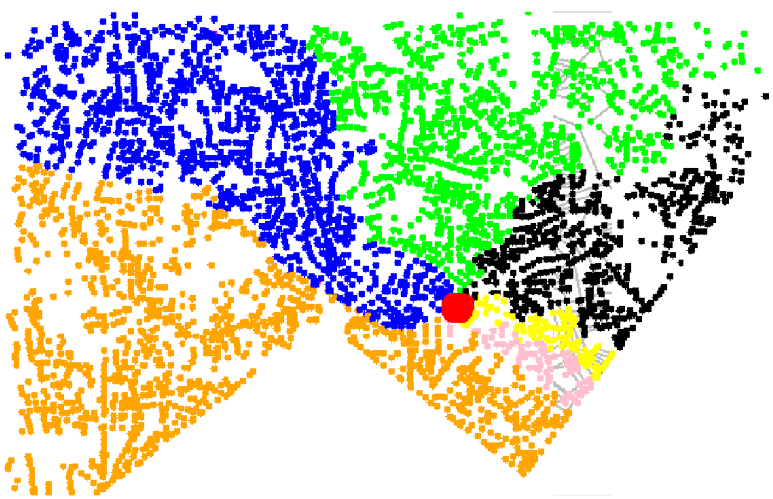
Path Coherence Beyond SILC

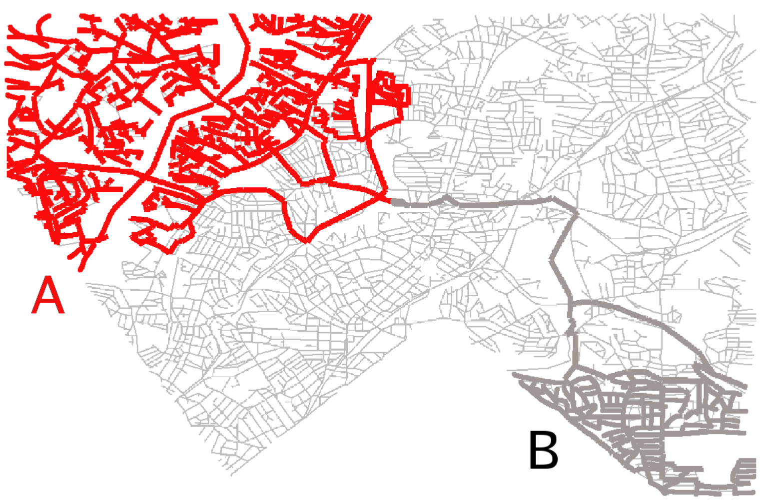
- The SILC framework captures the path coherence in the shortest paths
  - Captured: single source vertex to multiple destination vertices
  - Not captured: multiple source vertices to multiple destination vertices
- A new framework: Path Coherent Pairs (PCP)
  - Example of a path coherent pair denoted by: (   )



Path Coherence Beyond SILC

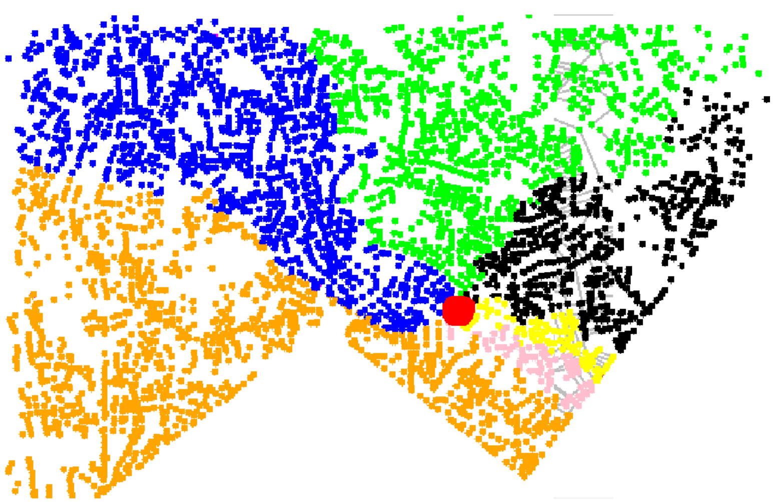
- The SILC framework captures the path coherence in the shortest paths
  - Captured: single source vertex to multiple destination vertices
  - Not captured: multiple source vertices to multiple destination vertices
- A new framework: Path Coherent Pairs (PCP)
  - Example of a path coherent pair denoted by: (A,)
    - A is a set of source vertices

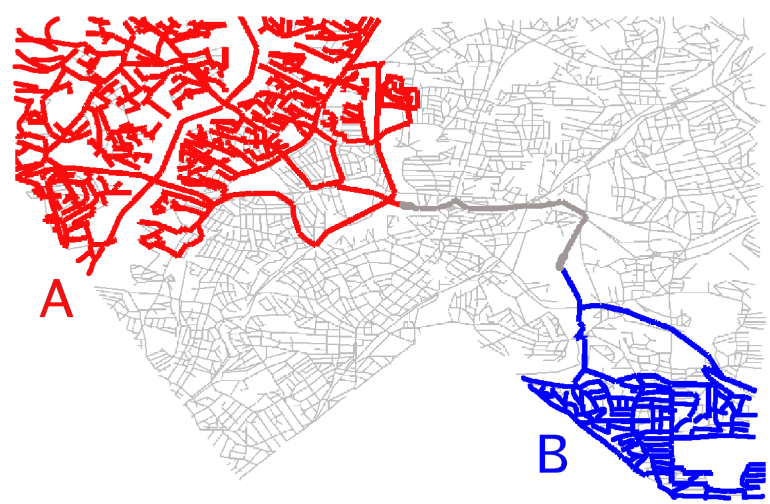




Path Coherence Beyond SILC

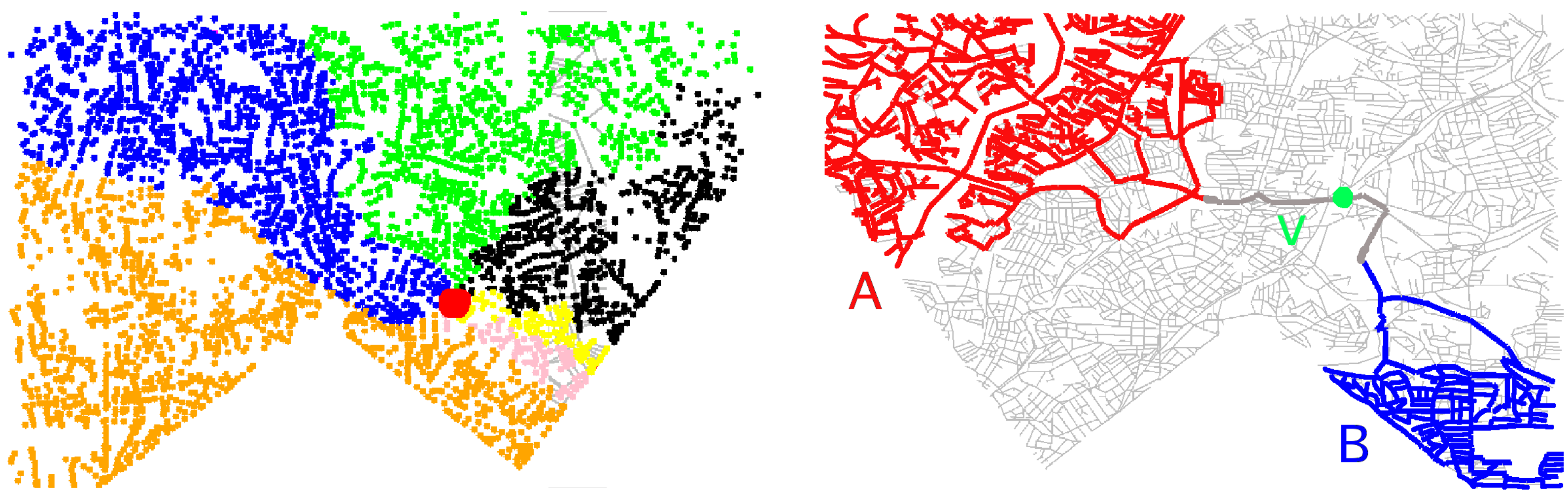
- The SILC framework captures the path coherence in the shortest paths
  - Captured: single source vertex to multiple destination vertices
  - Not captured: multiple source vertices to multiple destination vertices
- A new framework: Path Coherent Pairs (PCP)
  - Example of a path coherent pair denoted by: (A, B,   )
    - A is a set of source vertices
    - B is a set of destination vertices





Path Coherence Beyond SILC

- The SILC framework captures the path coherence in the shortest paths
  - Captured: single source vertex to multiple destination vertices
  - Not captured: multiple source vertices to multiple destination vertices
- A new framework: Path Coherent Pairs (PCP)
  - Example of a path coherent pair denoted by: (A, B, v)
    - A is a set of source vertices
    - B is a set of destination vertices
    - v is a common vertex to all pairs of shortest paths



Finding Path Coherent Pairs in Spatial Networks

Finding Path Coherent Pairs in Spatial Networks

- Source Vertices:

Finding Path Coherent Pairs in Spatial Networks

- Source Vertices: Washington, DC (D)

Finding Path Coherent Pairs in Spatial Networks

- Source Vertices: Washington, DC (D) , New York (N)

Finding Path Coherent Pairs in Spatial Networks

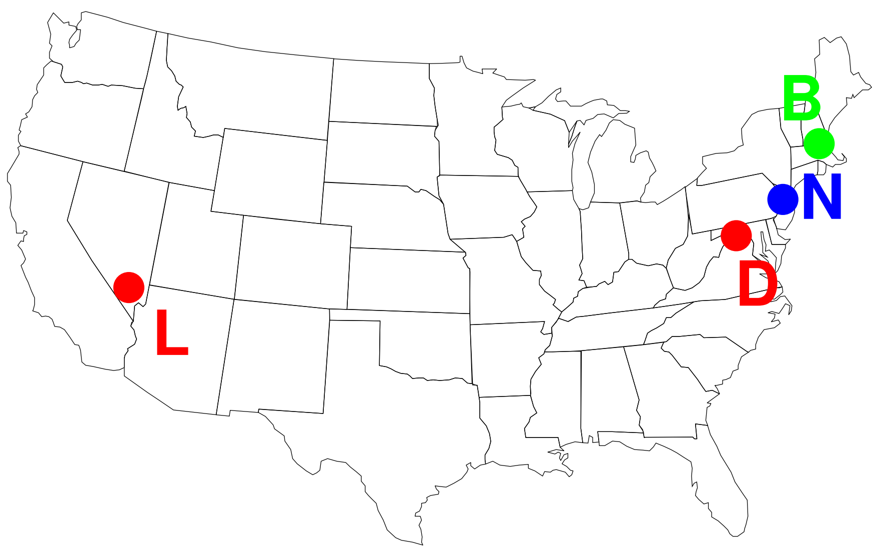
- Source Vertices: Washington, DC (D) , New York (N) , Boston (B)

Finding Path Coherent Pairs in Spatial Networks

- Source Vertices: Washington, DC (D) , New York (N) , Boston (B)
- Destination vertices:

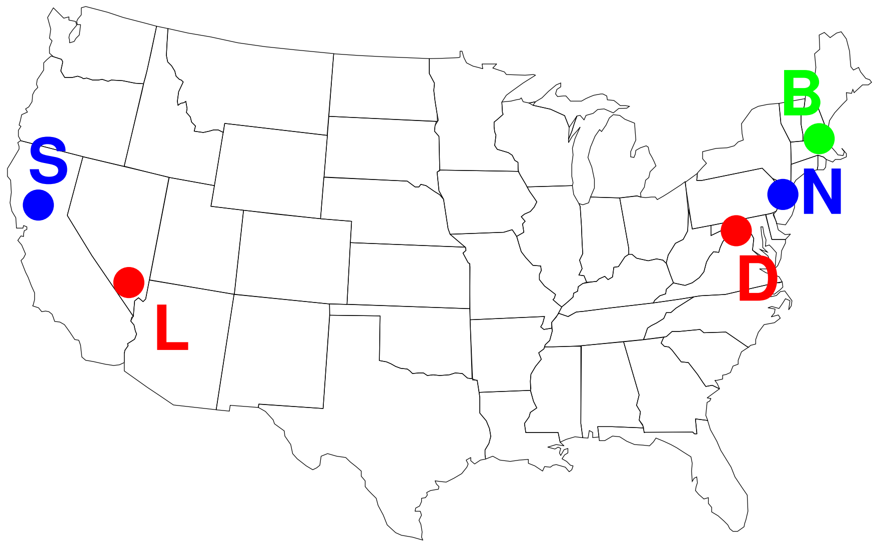
Finding Path Coherent Pairs in Spatial Networks

- Source Vertices: Washington, DC (D) , New York (N) , Boston (B)
- Destination vertices: Las Vegas (L)



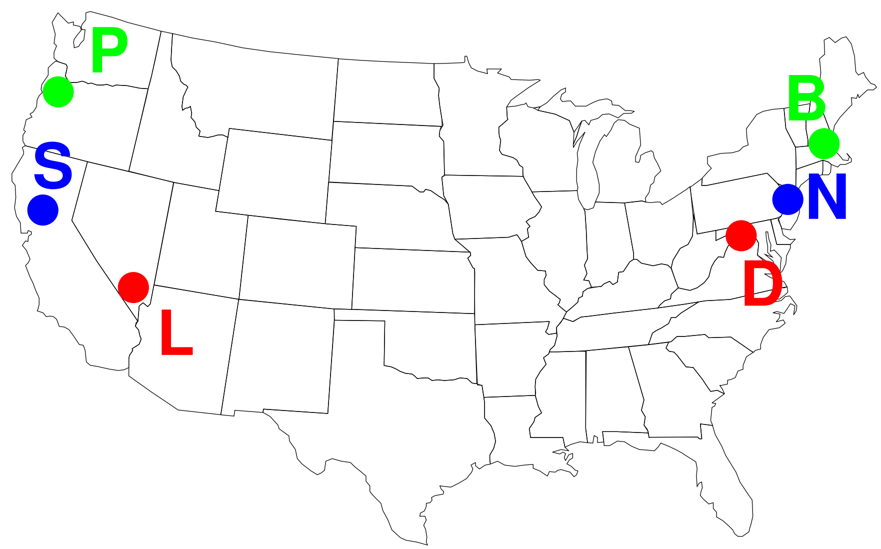
Finding Path Coherent Pairs in Spatial Networks

- Source Vertices: Washington, DC (D) , New York (N) , Boston (B)
- Destination vertices: Las Vegas (L) , Sacramento (S)



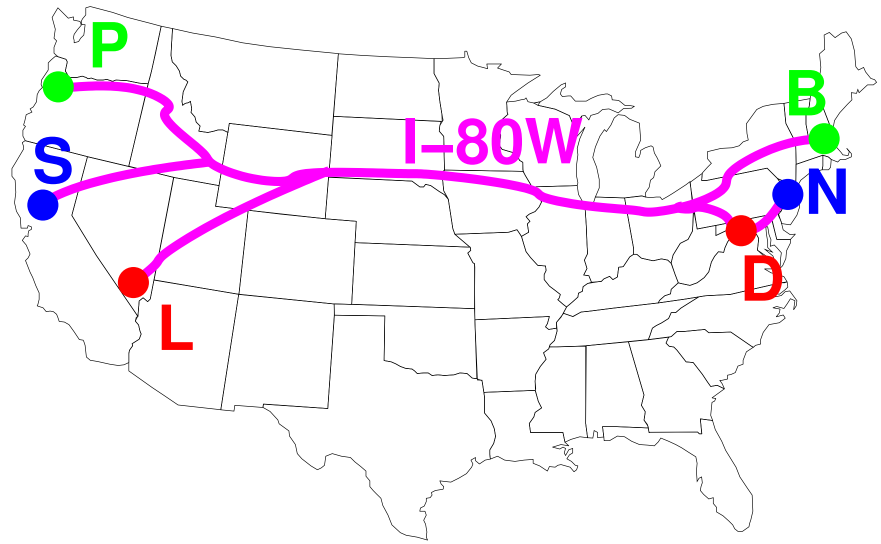
Finding Path Coherent Pairs in Spatial Networks

- Source Vertices: Washington, DC (D) , New York (N) , Boston (B)
- Destination vertices: Las Vegas (L) , Sacramento (S) , Portland (P)



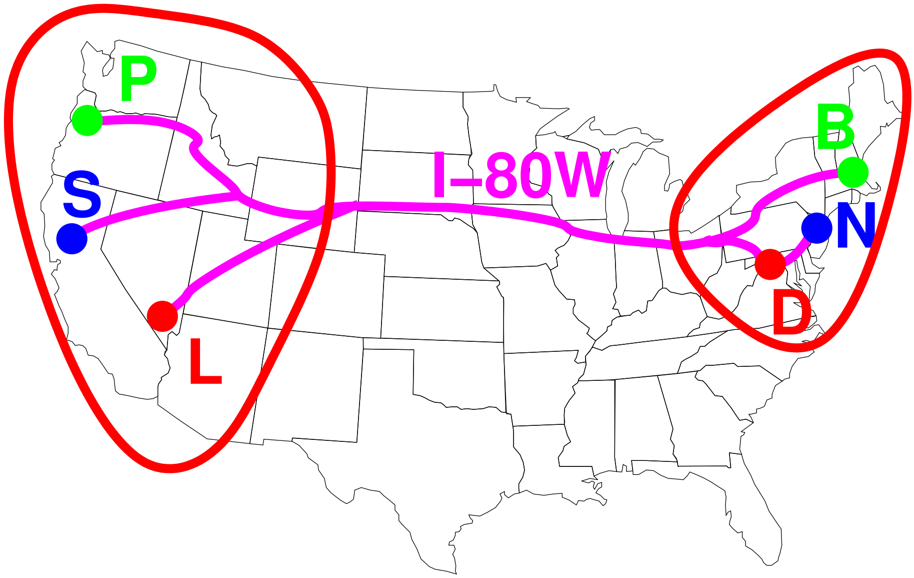
Finding Path Coherent Pairs in Spatial Networks

- Source Vertices: Washington, DC (D) , New York (N) , Boston (B)
- Destination vertices: Las Vegas (L) , Sacramento (S) , Portland (P)
- Anyone driving from “North-East” to “North-West” US uses I-80W



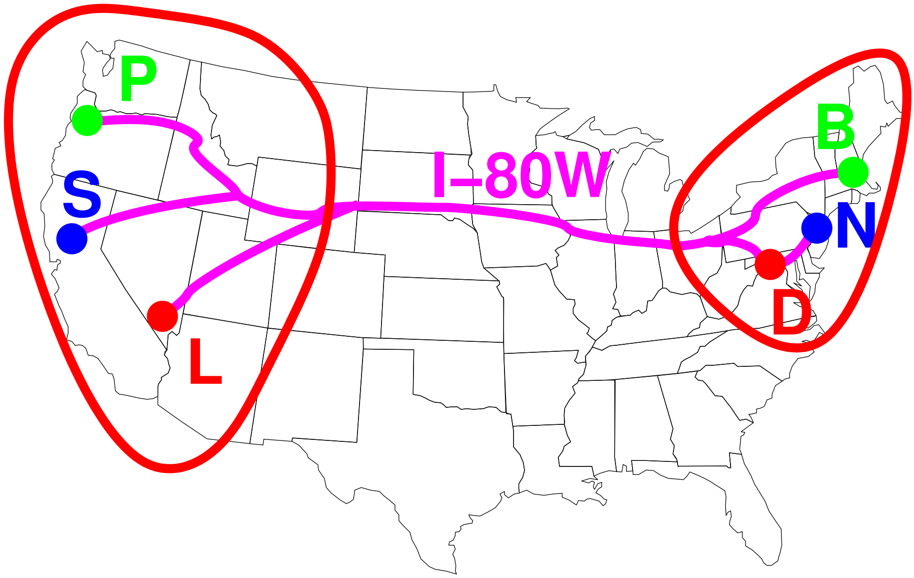
Finding Path Coherent Pairs in Spatial Networks

- Source Vertices: Washington, DC (D) , New York (N) , Boston (B)
- Destination vertices: Las Vegas (L) , Sacramento (S) , Portland (P)
- Anyone driving from “North-East” to “North-West” US uses I-80W
- Capture shortest paths from one million (say) sources in “North-East” to one million (say) destinations in “North-West” using $O(1)$ storage



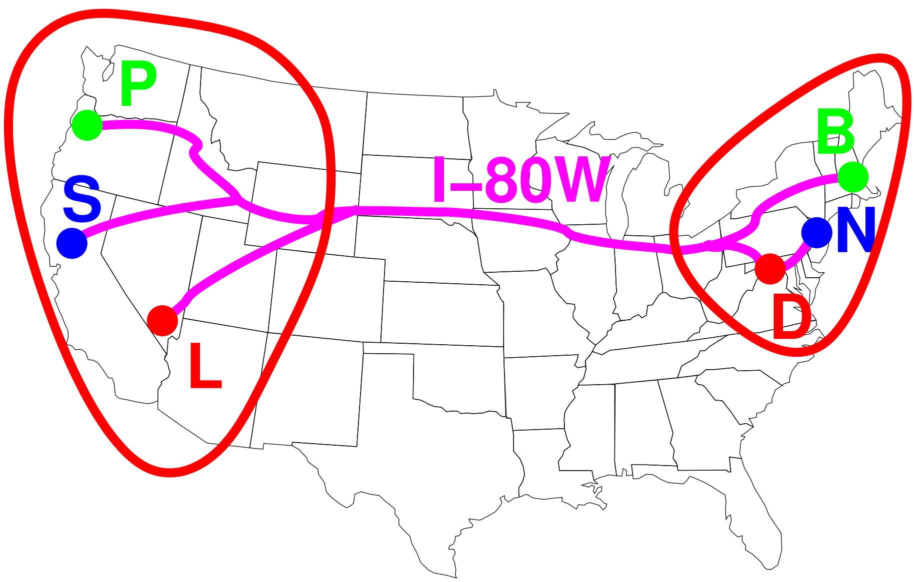
Finding Path Coherent Pairs in Spatial Networks

- Source Vertices: Washington, DC (D) , New York (N) , Boston (B)
- Destination vertices: Las Vegas (L) , Sacramento (S) , Portland (P)
- Anyone driving from "North-East" to "North-West" US uses I-80W
- Capture shortest paths from one million (say) sources in "North-East" to one million (say) destinations in "North-West" using $O(1)$ storage
- Intuition: Sources "sufficiently far" from destinations share common vertices in their shortest paths



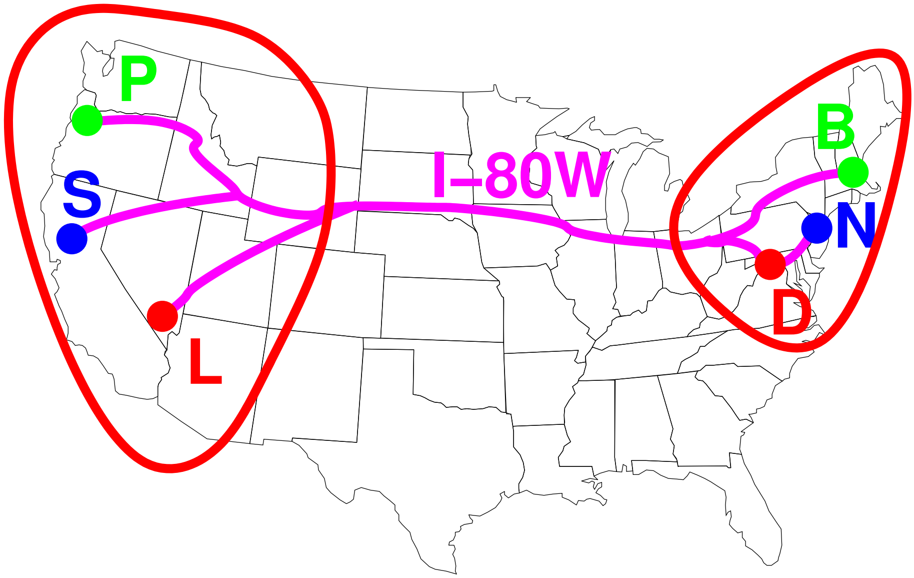
Finding Path Coherent Pairs in Spatial Networks

- Source Vertices: Washington, DC (D) , New York (N) , Boston (B)
- Destination vertices: Las Vegas (L) , Sacramento (S) , Portland (P)
- Anyone driving from “North-East” to “North-West” US uses I-80W
- Capture shortest paths from one million (say) sources in “North-East” to one million (say) destinations in “North-West” using $O(1)$ storage
- Intuition: Sources “sufficiently far” from destinations share common vertices in their shortest paths
- Decompose road network into PCPs:
  - Any vertex pair is contained in exactly one set in the shape of a dumbbell
  - All N^2 shortest paths are captured using $O(s^d N)$ storage where s is a small constant



Finding Path Coherent Pairs in Spatial Networks

- Source Vertices: Washington, DC (D) , New York (N) , Boston (B)
- Destination vertices: Las Vegas (L) , Sacramento (S) , Portland (P)
- Anyone driving from "North-East" to "North-West" US uses I-80W
- Capture shortest paths from one million (say) sources in "North-East" to one million (say) destinations in "North-West" using $O(1)$ storage
- Intuition: Sources "sufficiently far" from destinations share common vertices in their shortest paths
- Decompose road network into PCPs:
  - Any vertex pair is contained in exactly one set in the shape of a dumbbell
  - All N^2 shortest paths are captured using $O(s^d N)$ storage where s is a small constant



- Key idea is the analogy to the well-separated pairs in computational geometry

SET OPERATIONS ON QUADTREES

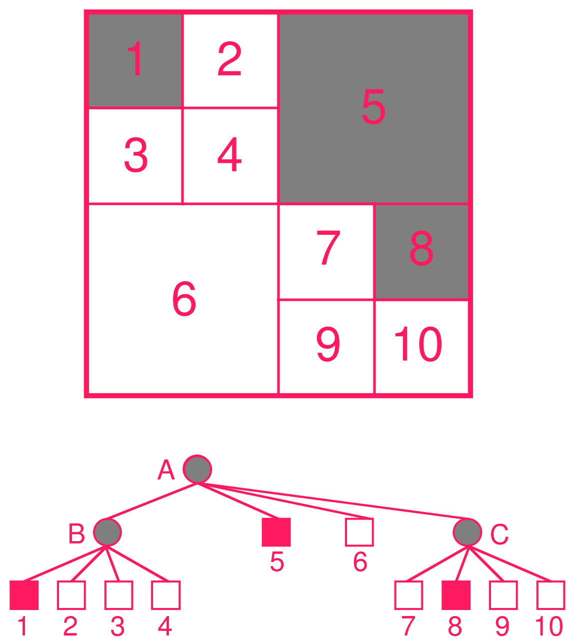
- UNION(S,T) : traverse S and T in tandem

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

SET OPERATIONS ON QUADTREES

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

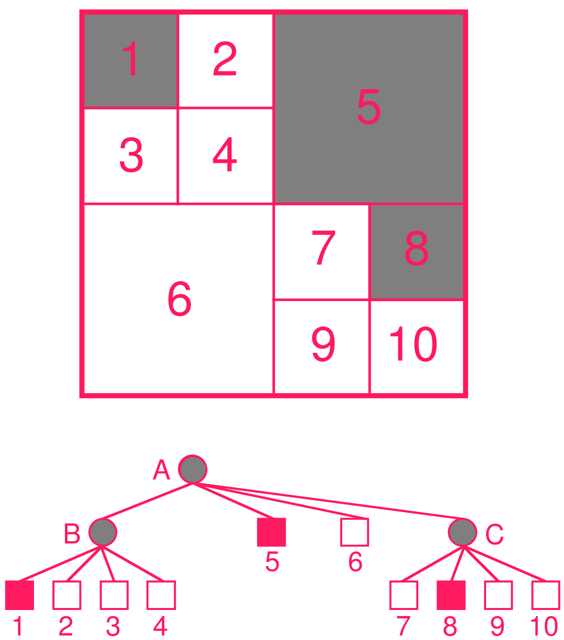
1. GRAY(S) ● :
   - GRAY(T) ● : recursively process subtrees and merge if all resulting sons are BLACK
   - BLACK(T) ● : result is T
   - WHITE(T) ○ : result is S
2. BLACK(S) ● : result is S
3. WHITE(S) ○ : result is T

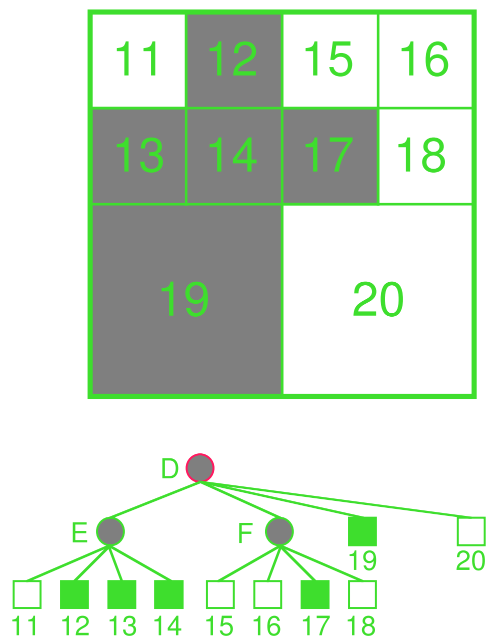


SET OPERATIONS ON QUADTREES

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

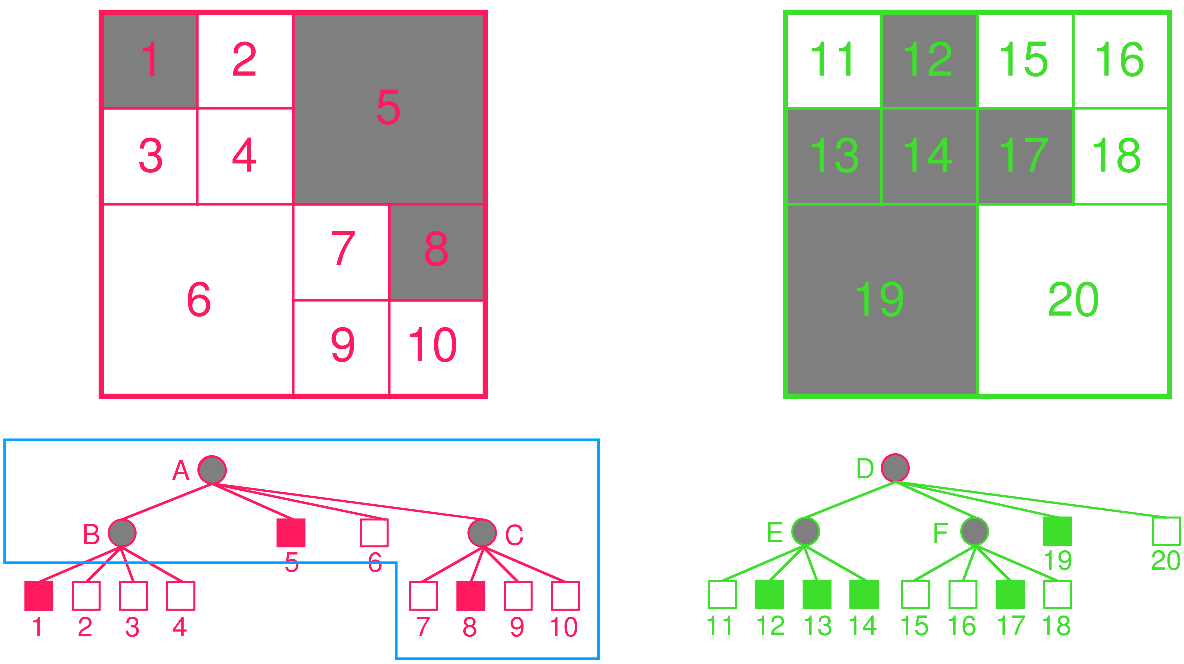
1. GRAY(S) ● :
   - GRAY(T) ● : recursively process subtrees and merge if all resulting sons are BLACK
   - BLACK(T) ● : result is T
   - WHITE(T) ○ : result is S
2. BLACK(S) ● : result is S
3. WHITE(S) ○ : result is T





SET OPERATIONS ON QUADTREES

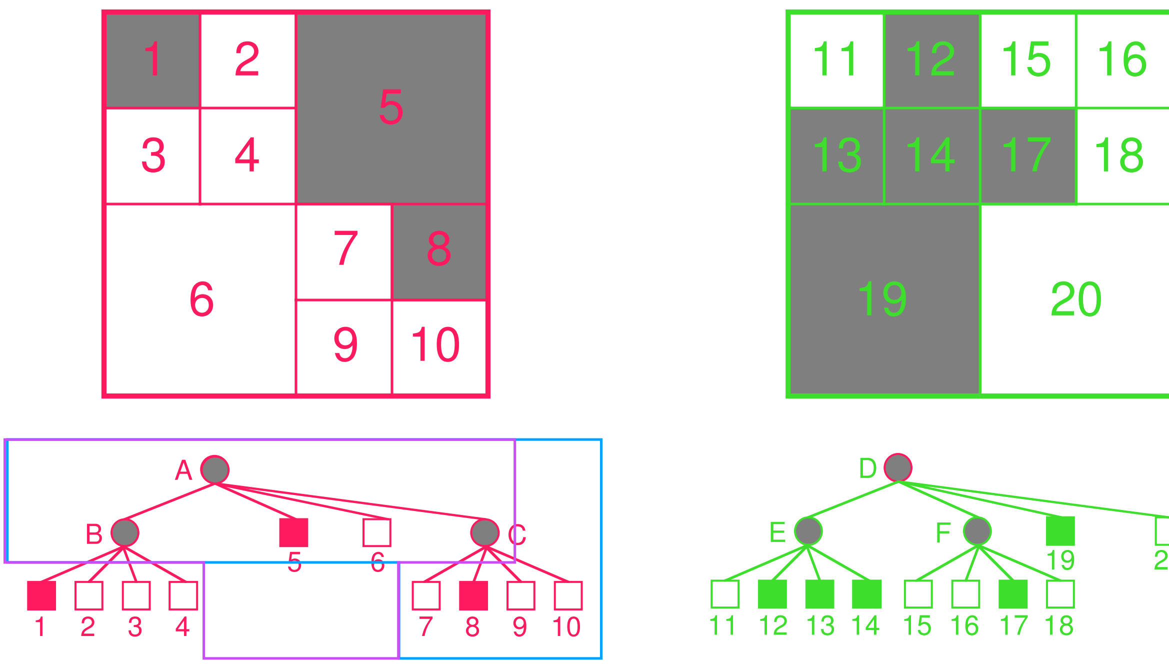
- UNION(S,T) : traverse S and T in tandem
  1. GRAY(S) ● :
     - GRAY(T) ● : recursively process subtrees and merge if all resulting sons are BLACK
     - BLACK(T) ● : result is T
     - WHITE(T) ○ : result is S
  2. BLACK(S) ● : result is S
  3. WHITE(S) ○ : result is T



SET OPERATIONS ON QUADTREES

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

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

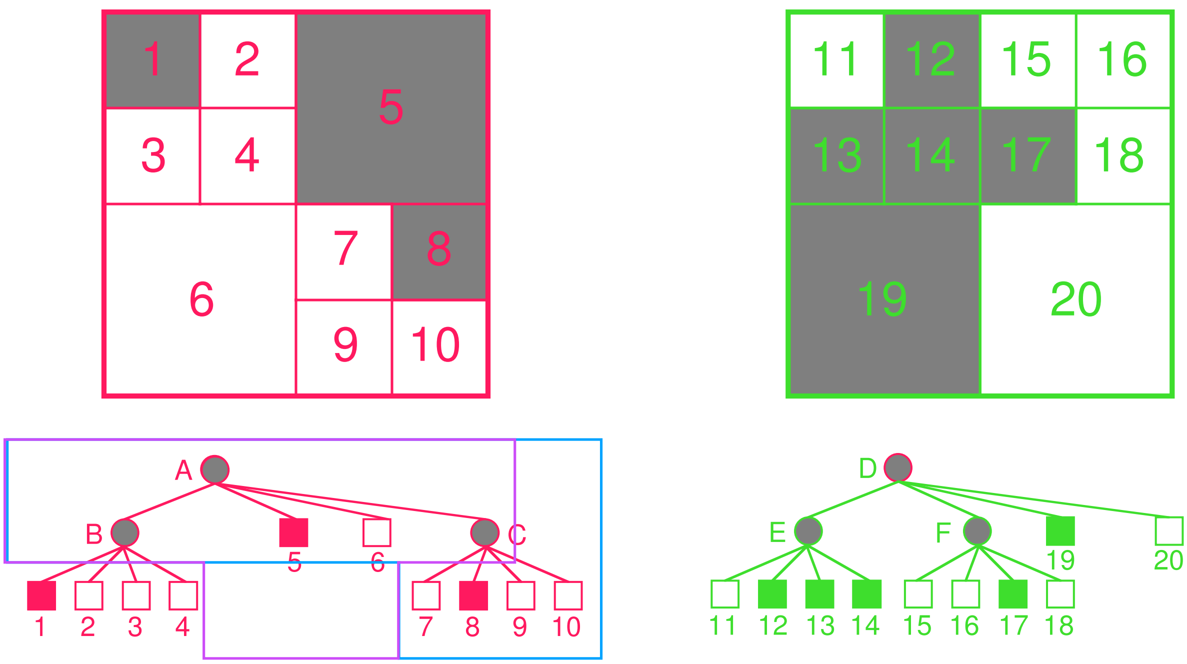


- INTERSECTION: interchange roles of BLACK and WHITE in UNION

SET OPERATIONS ON QUADTREES

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

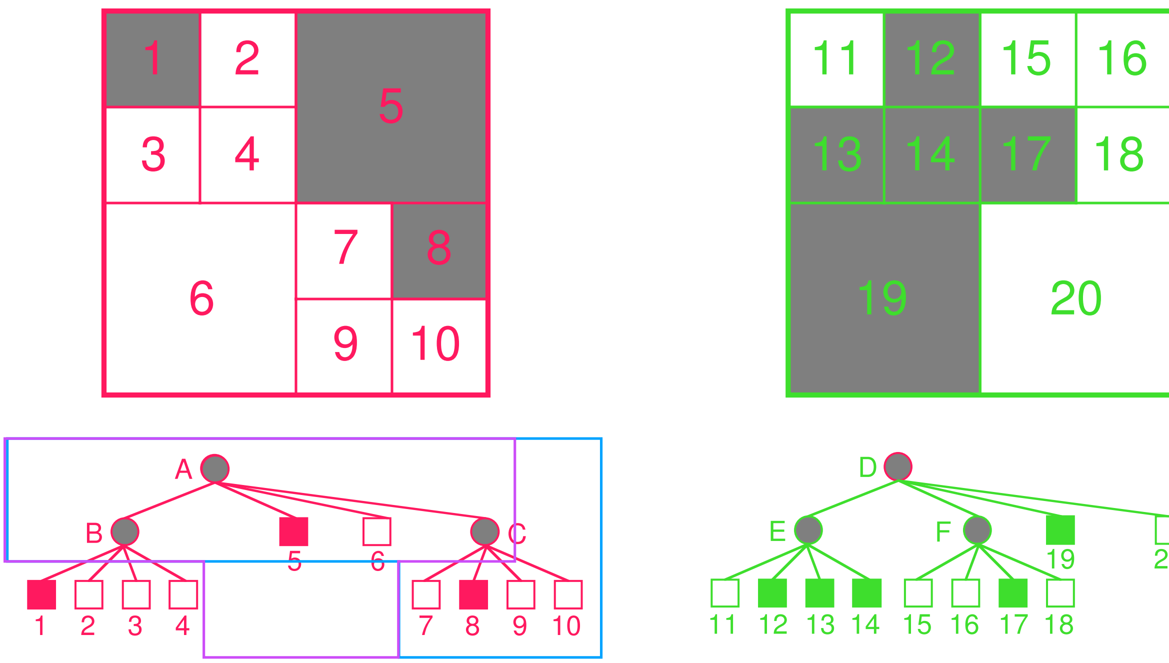
  1. GRAY(S) ● :
     - GRAY(T) ● : recursively process subtrees and merge if all resulting sons are BLACK
     - BLACK(T) ● : result is T
     - WHITE(T) ○ : result is S
  2. BLACK(S) ● : result is S
  3. WHITE(S) ○ : result is T



- INTERSECTION: interchange roles of BLACK and WHITE in UNION
- Execution time is bounded by sum of nodes in two input trees but may be less if don't create a new copy as really just the sum of the minimum of the number of nodes at corresponding levels of the two quadtrees

SET OPERATIONS ON QUADTREES

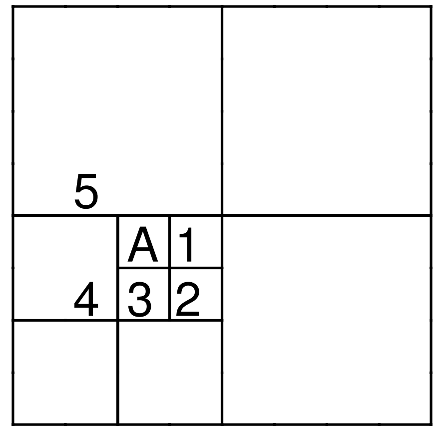
- UNION(S,T) : traverse S and T in tandem
  1. GRAY(S) ● :
     - GRAY(T) ● : recursively process subtrees and merge if all resulting sons are BLACK
     - BLACK(T) ● : result is T
     - WHITE(T) ○ : result is S
  2. BLACK(S) ● : result is S
  3. WHITE(S) ○ : result is T



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

NEIGHBOR FINDING OPERATIONS USING QUADTREES

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

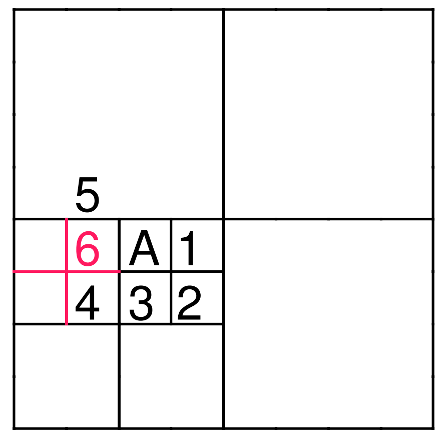


A has 5 neighbors

- Desirable to be able to locate neighbors in a manner that
  1. is position-independent
  2. is size-independent
  3. makes no use of additional links to adjacent nodes (e.g., ropes and nets a la Hunter)
  4. just uses the structure of the tree or configuration of the blocks

NEIGHBOR FINDING OPERATIONS USING QUADTREES

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

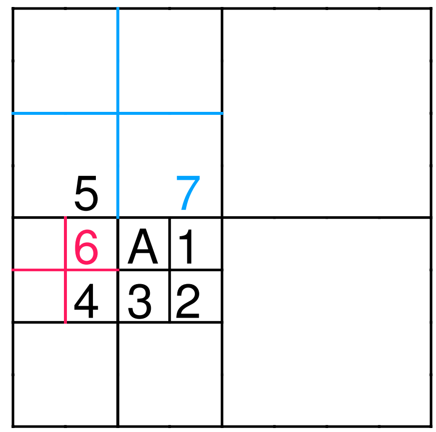


A has ~~5~~ 6 neighbors

- Desirable to be able to locate neighbors in a manner that
  1. is position-independent
  2. is size-independent
  3. makes no use of additional links to adjacent nodes (e.g., ropes and nets a la Hunter)
  4. just uses the structure of the tree or configuration of the blocks

NEIGHBOR FINDING OPERATIONS USING QUADTREES

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

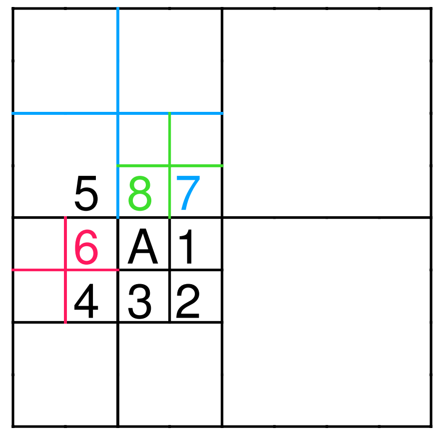


A has ~~5~~ ~~6~~ 7 neighbors

- Desirable to be able to locate neighbors in a manner that
  1. is position-independent
  2. is size-independent
  3. makes no use of additional links to adjacent nodes (e.g., ropes and nets a la Hunter)
  4. just uses the structure of the tree or configuration of the blocks

NEIGHBOR FINDING OPERATIONS USING QUADTREES

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

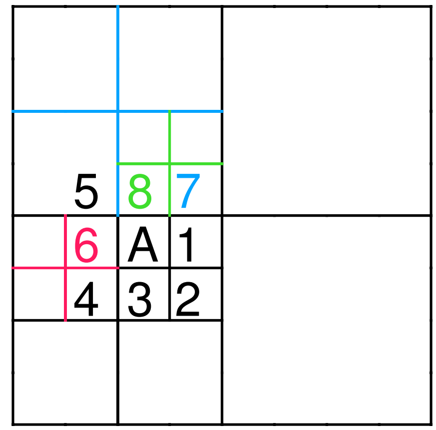


A has ~~5~~ ~~6~~ ~~7~~ 8 neighbors

- Desirable to be able to locate neighbors in a manner that
  1. is position-independent
  2. is size-independent
  3. makes no use of additional links to adjacent nodes (e.g., ropes and nets a la Hunter)
  4. just uses the structure of the tree or configuration of the blocks

NEIGHBOR FINDING OPERATIONS USING QUADTREES

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



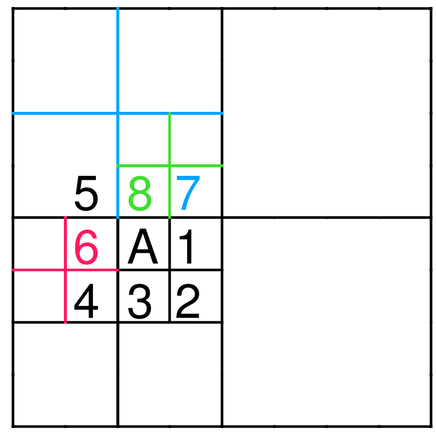
A has ~~5~~ ~~6~~ ~~7~~ 8 neighbors

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

  2. partial overlap of two blocks B and C with A is impossible since a quadtree is constructed by recursively splitting blocks into blocks that have side lengths that are powers of 2

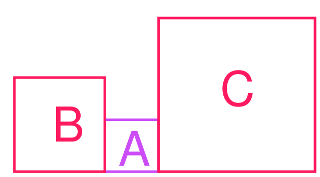
NEIGHBOR FINDING OPERATIONS USING QUADTREES

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

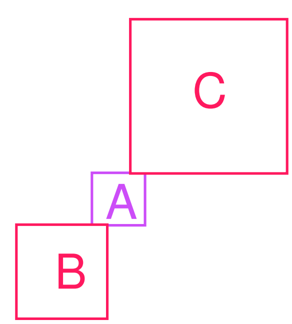


A has ~~5~~ ~~6~~ ~~7~~ 8 neighbors

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



  2. partial overlap of two blocks B and C with A is impossible since a quadtree is constructed by recursively splitting blocks into blocks that have side lengths that are powers of 2

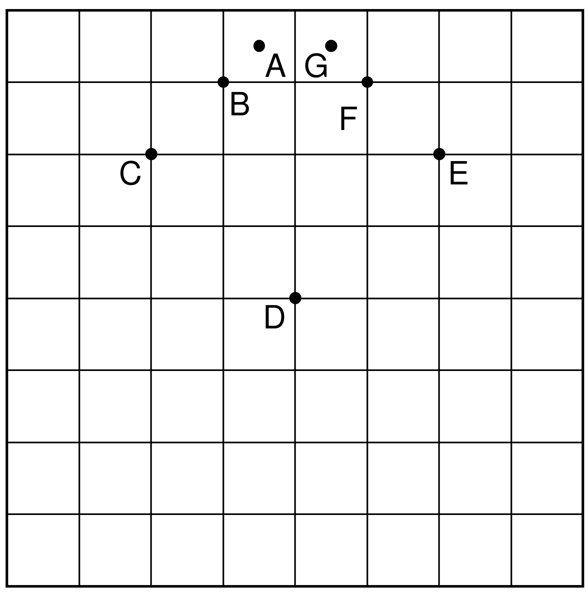


FINDING LATERAL NEIGHBORS OF EQUAL SIZE

Algorithm: based on finding the nearest common ancestor

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

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

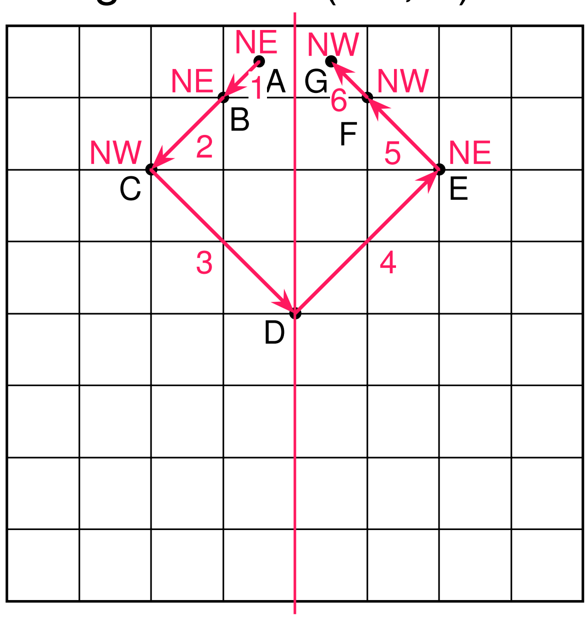


FINDING LATERAL NEIGHBORS OF EQUAL SIZE

Algorithm: based on finding the nearest common ancestor

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

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

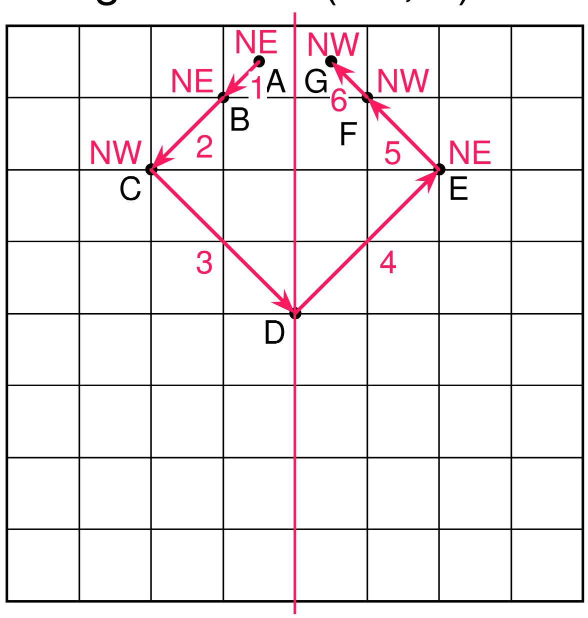


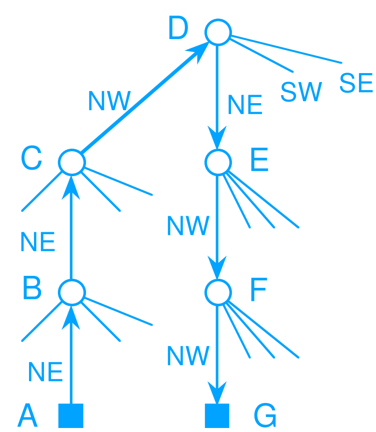
FINDING LATERAL NEIGHBORS OF EQUAL SIZE

Algorithm: based on finding the nearest common ancestor

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

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



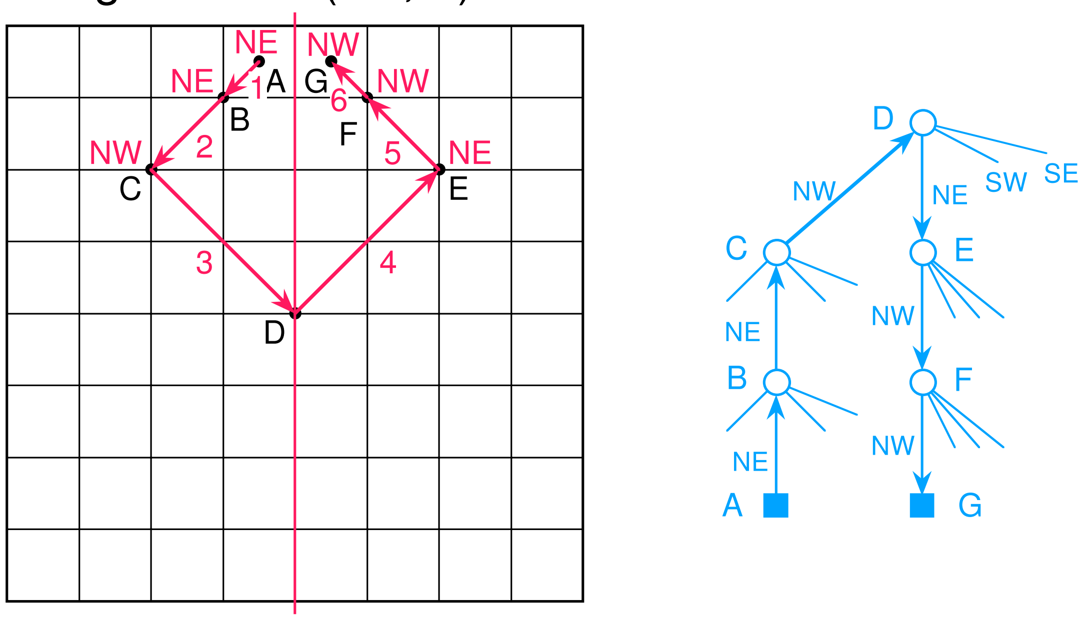


FINDING LATERAL NEIGHBORS OF EQUAL SIZE

Algorithm: based on finding the nearest common ancestor

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

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



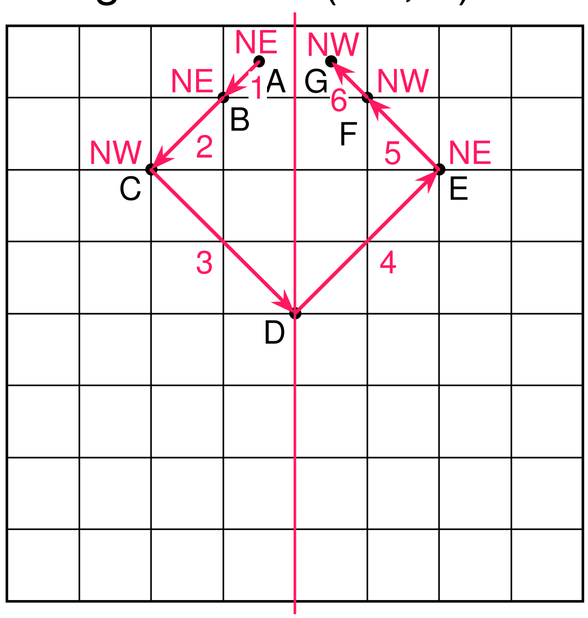

```
node procedure EQUAL_LATERAL_NEIGHBOR(P,D);
/* Find = size neighbor of P in direction D */
begin
  value pointer node P;
  value direction D;
  return(SON(if ADJ(D,SONTYPE(P)) then
                 EQUAL_LATERAL_NEIGHBOR(FATHER(P),D)
             else FATHER(P),
             REFLECT(D,SONTYPE(P))));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

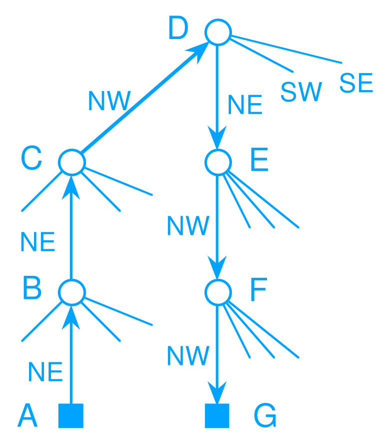
FINDING LATERAL NEIGHBORS OF EQUAL SIZE

Algorithm: based on finding the nearest common ancestor

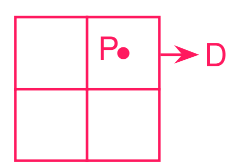
1. Ascend the tree if the node is a son of the same type as the direction of the neighbor (ADJ)
2. Otherwise, the father F is the nearest common ancestor and retrace the path starting at F making mirror image moves about the edge shared by the neighboring blocks

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






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



ADJ(A,B)

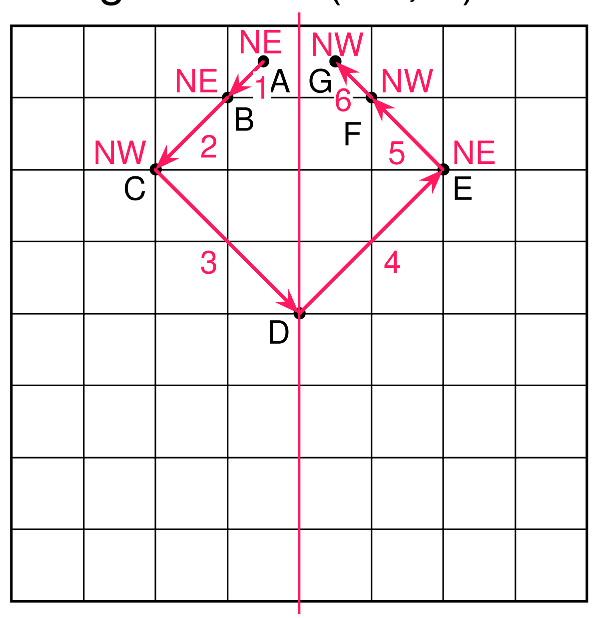
A \ B	NW	NE	SW	SE
N	T	T	F	F
E	F	T	F	T
S	F	F	T	T
W	T	F	T	F

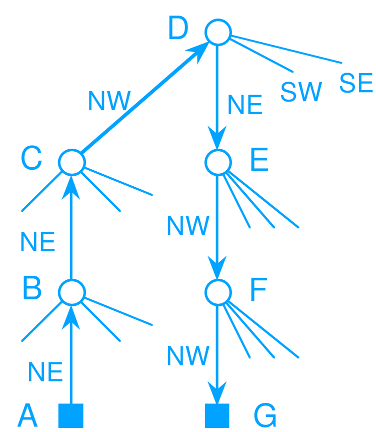
FINDING LATERAL NEIGHBORS OF EQUAL SIZE

Algorithm: based on finding the nearest common ancestor

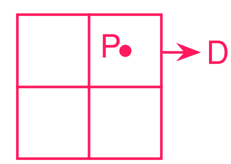
1. Ascend the tree if the node is a son of the same type as the direction of the neighbor (ADJ)
2. Otherwise, the father F is the nearest common ancestor and retrace the path starting at F making mirror image moves about the edge shared by the neighboring blocks

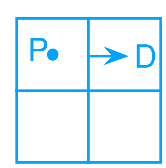
Ex: E neighbor of A (i.e., G)






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





ADJ(A,B)

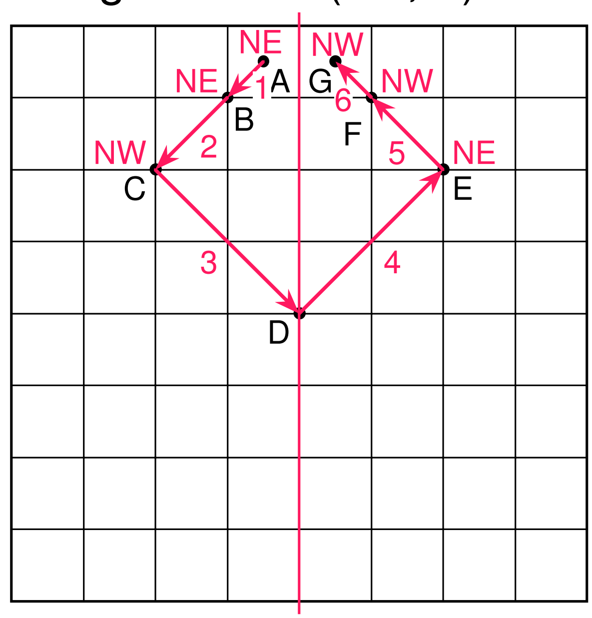
A \ B	NW	NE	SW	SE
N	T	T	F	F
E	F	T	F	T
S	F	F	T	T
W	T	F	T	F

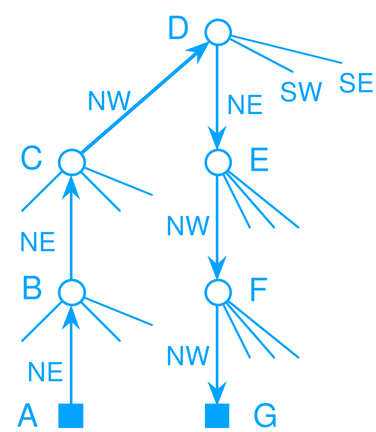
FINDING LATERAL NEIGHBORS OF EQUAL SIZE

Algorithm: based on finding the nearest common ancestor

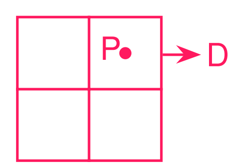
1. Ascend the tree if the node is a son of the same type as the direction of the neighbor (ADJ)
2. Otherwise, the father F is the nearest common ancestor and retrace the path starting at F making mirror image moves about the edge shared by the neighboring blocks

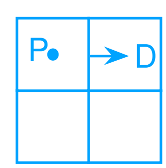
Ex: E neighbor of A (i.e., G)






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





ADJ(A,B)

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

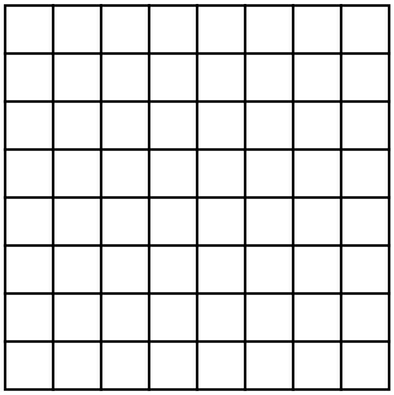
REFLECT(A,B)

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

ANALYSIS OF NEIGHBOR FINDING

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

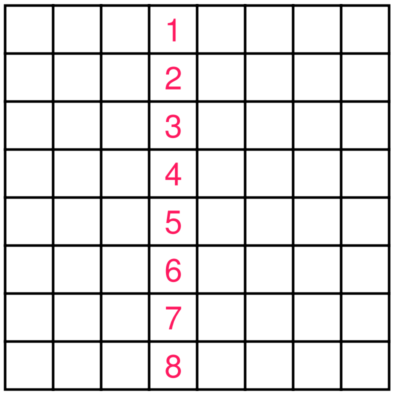
ANALYSIS OF FINDING LATERAL NEIGHBORS



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

NCA = nearest common ancestor

ANALYSIS OF FINDING LATERAL NEIGHBORS



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

NCA = nearest common ancestor

1–8 have NCA at level 3

ANALYSIS OF FINDING LATERAL NEIGHBORS

	9		1		17		
	10		2		18		
	11		3		19		
	12		4		20		
	13		5		21		
	14		6		22		
	15		7		23		
	16		8		24		

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

NCA = nearest common ancestor

1–8 have NCA at level 3

9–24 have NCA at level 2

ANALYSIS OF FINDING LATERAL NEIGHBORS

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

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

NCA = nearest common ancestor

1–8 have NCA at level 3

9–24 have NCA at level 2

25–56 have NCA at level 1

ANALYSIS OF FINDING LATERAL NEIGHBORS

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

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

NCA = nearest common ancestor

1–8 have NCA at level 3

9–24 have NCA at level 2

25–56 have NCA at level 1

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

Proof:

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

  2^{n-i} rows

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

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

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

  …

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

ANALYSIS OF FINDING LATERAL NEIGHBORS

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

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

NCA = nearest common ancestor

1–8 have NCA at level 3

9–24 have NCA at level 2

25–56 have NCA at level 1

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

Proof:

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

  2^{n-i} rows

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

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

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

  …

  $2^{n-i} \cdot 2^{n-i-1}$ have NCA at level $i+1$
- For node A at level i, direction D, and the NCA at level j, $2 \cdot (j-i)$ nodes are visited in locating an equal-sized neighbor at level i

ANALYSIS OF FINDING LATERAL NEIGHBORS

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

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

NCA = nearest common ancestor

1–8 have NCA at level 3

9–24 have NCA at level 2

25–56 have NCA at level 1

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

Proof:

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

  2^{n-i} rows

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

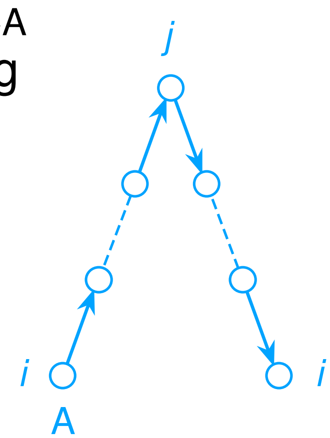
  $2^{n-i} \cdot 2^0$ have NCA at level n

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

  …

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

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



ANALYSIS OF FINDING LATERAL NEIGHBORS

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

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

NCA = nearest common ancestor

1–8 have NCA at level 3

9–24 have NCA at level 2

25–56 have NCA at level 1

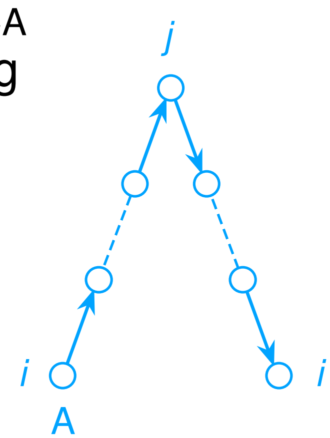
Theorem: average number of nodes visited by `EQUAL_LATERAL_NEIGHBOR` is ≤ 4

Proof:

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

  2^{n-i} rows
  $2^{n-i} - 1$ adjacencies per row
  $2^{n-i} \cdot 2^0$ have NCA at level n
  $2^{n-i} \cdot 2^1$ have NCA at level $n-1$
  …
  $2^{n-i} \cdot 2^{n-i-1}$ have NCA at level $i+1$

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



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

nodes are visited on the average ≤ 4

Outline

1. Introduction
2. Points
3. Lines
4. Regions, Volumes, and Surfaces
5. Bounding Box Hierarchies
6. Rectangles
7. Surfaces and Volumes
8. Metric Data
9. Operations
10. Indexing Spatiotextual Data
11. Example system

Extend GIS Notions to Textually-Specified Spatial Data

- Spatial data specification
  1. Usually geometrically
  2. But could also be done textually
     - Advantage: text is a polymorphic type
       - Ex: “Los Angeles” can denote either an area or a point
     - Disadvantage: ambiguity
       - Ex: “Paris, France” or “Paris, Texas”
- Location-based vs: feature-based queries
  1. Location-based: all documents/topics mentioning location/region R
     - Equivalently, top K topics in location/region R
     - Specify R by direct manipulation like a rectangular window
  2. Feature-based: all locations/regions mentioned in topic T articles
     - Equivalently, top K locations mentioned in articles about topic T
     - T is not necessarily known a priori
     - Topics are ranked by importance which could be defined by the number of documents that comprise them
- Extend further to spatial data specified by direct manipulation actions such as pointing, pan, and zoom

Power of Spatial Synonyms

- Enables search for data when not exactly sure of what we are seeking, or what should be the answer to the query
  - Ex: Seek a “Rock Concert in Manhattan”
  - “Rock Concerts” in “Harlem” or “New York City” are good answers when no such events can be found in “Manhattan” as they correspond to approximate synonyms:
    - “Harlem” by virtue of proximity, and
    - “New York City” by virtue of a containment relationship

General Geotagging Issues

1. Toponym recognition: identify geographical references in text
   - Does “Jefferson” refer to a person or a geographical location?
2. Toponym resolution: disambiguate a geographical reference
   - Does “London” mean “London, UK”, “London, Ontario”, or one of 2570 other instances of “London” in our gazetteer?
3. Determine spatial focus of a document
   - Is “Singapore” relevant to a news article about “Hurricane Katrina”?
   - Not so, if article appeared in “Singapore Strait Times”

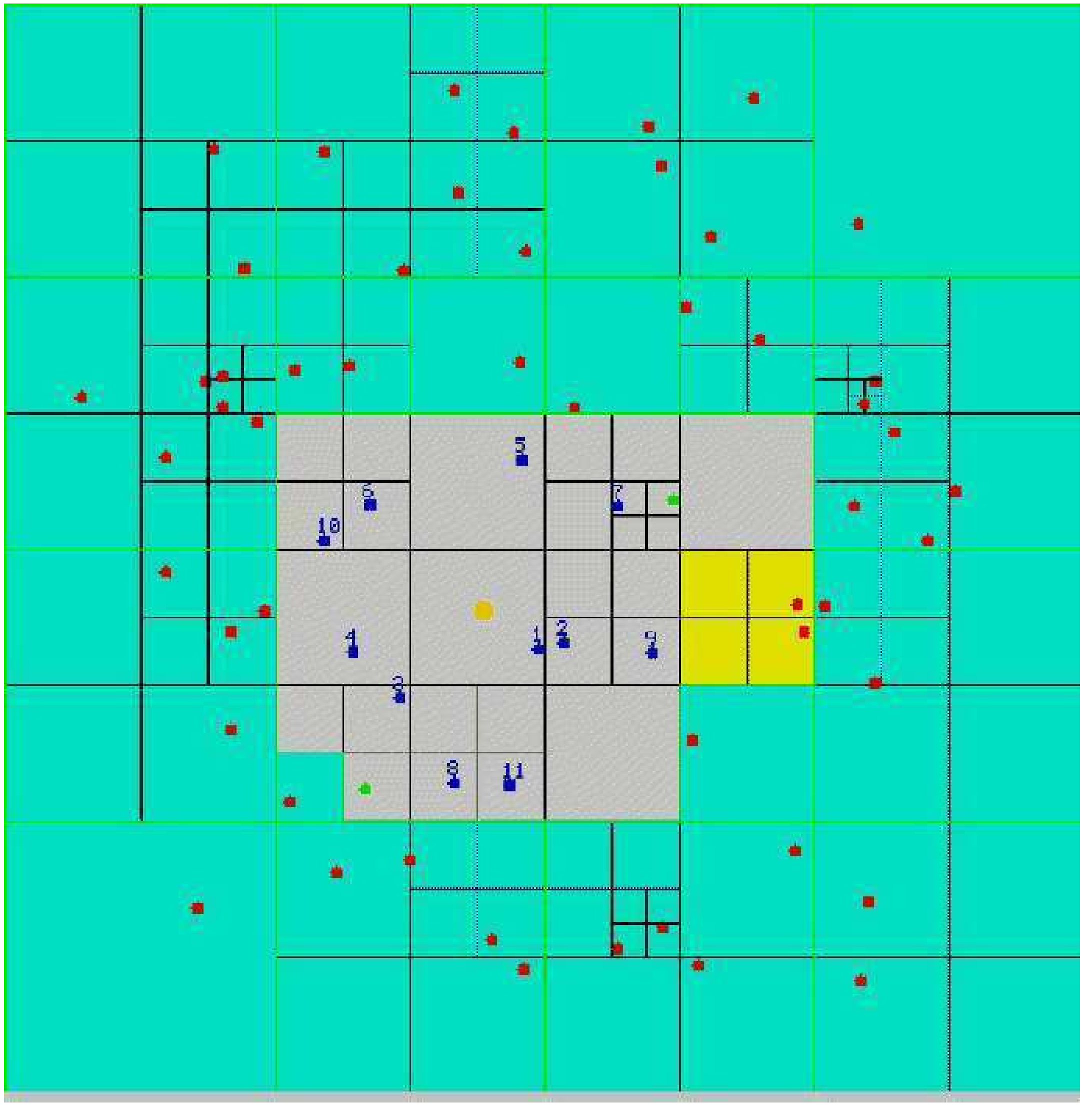
Mechanics of Geotagging

1. Goal: high recall in toponym recognition (i.e., not missing toponyms) at expense of precision
   - Rectify by subsequent use of toponym resolution which can (and will) also be used to filter erroneous location interpretations
2. Toponym recognition: 2 stages
   - Finding toponyms
   - Filtering toponyms: postprocessing to remove errors in recognition
3. Toponym resolution
   - Use local lexicons containing locations that can be specified without all of their containers (derived from articles from a particular news source) to determine spatial reader scopes for particular sources
     - E.g., "Dublin" implies “Dublin, Ohio” for readers of a news source in “Columbus, Ohio”
   - Use Wikipedia articles to find concepts related to particular locations so that the presence of these concepts in conjunction with an ambiguous reference to a location can be properly resolved
     - E.g., mention of “White House” in conjunction with “Washington” to provide evidence for resolving as “Washington, D.C.”

Outline

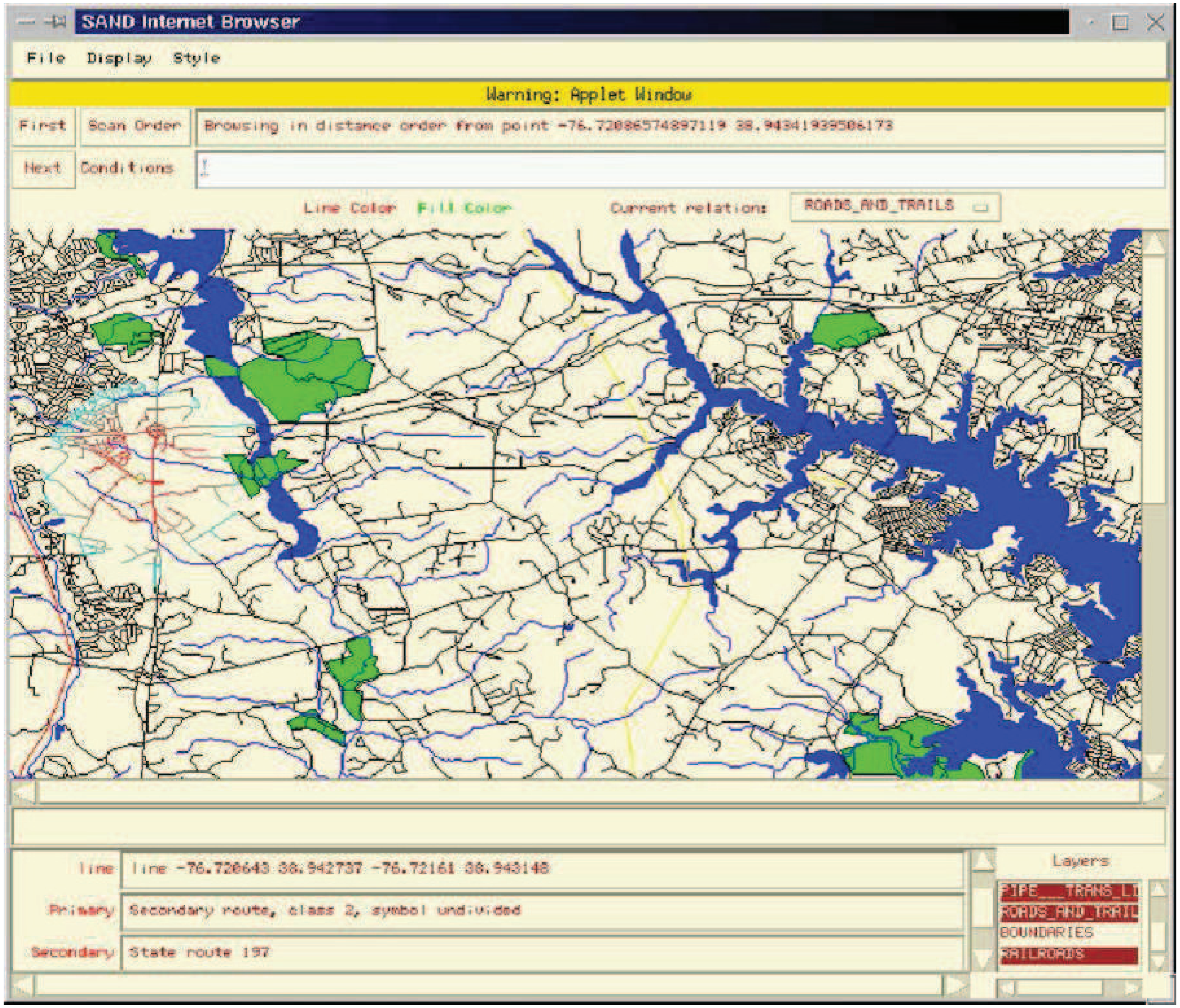
1. Introduction
2. Points
3. Lines
4. Regions, Volumes, and Surfaces
5. Bounding Box Hierarchies
6. Rectangles
7. Surfaces and Volumes
8. Metric Data
9. Operations
10. Indexing Spatiotextual Data
11. Example system

VASCO Spatial Applet



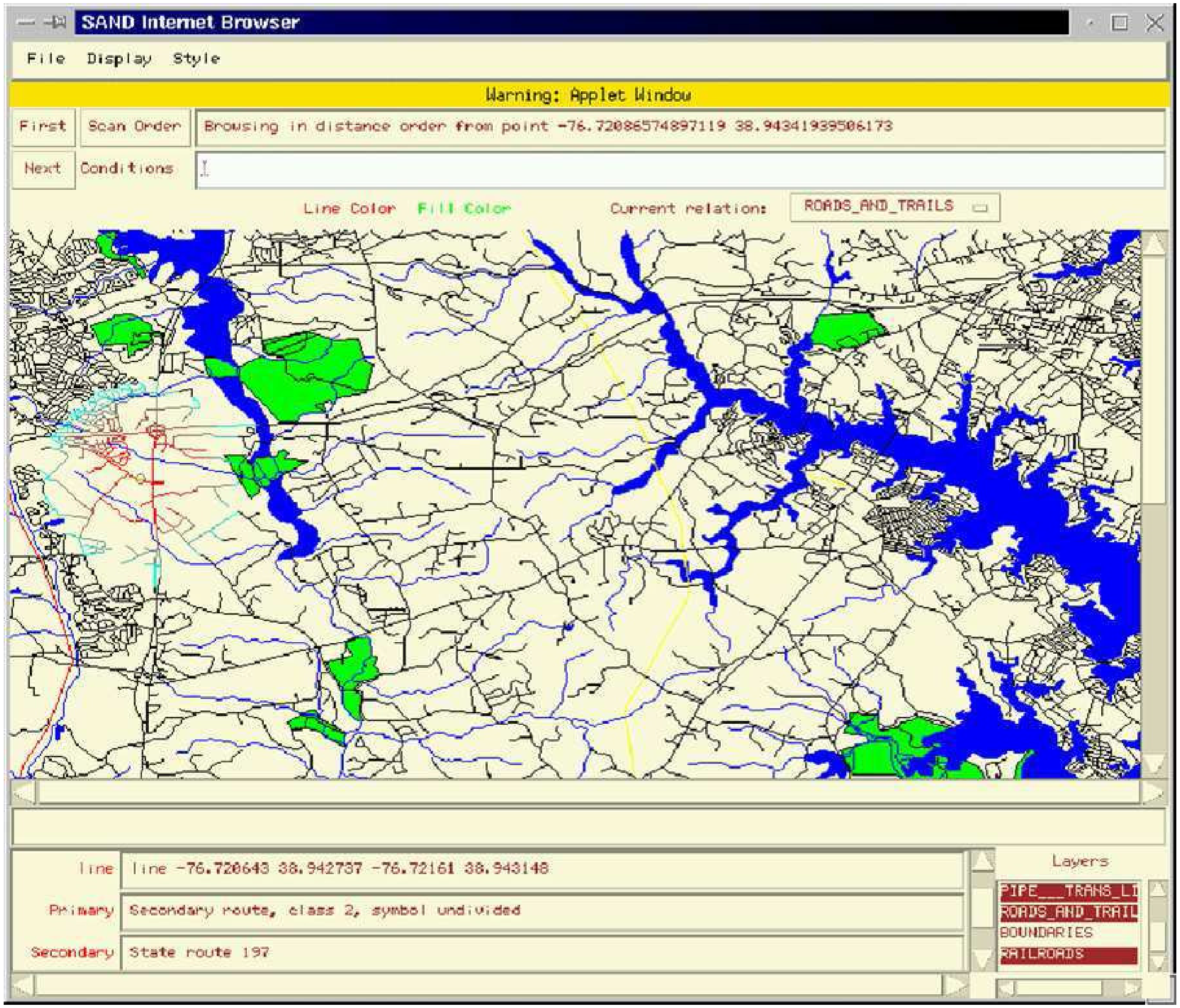
http://www.cs.umd.edu/~hjs/quadtree/index.html

SAND Internet Browser



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

SAND Internet Browser

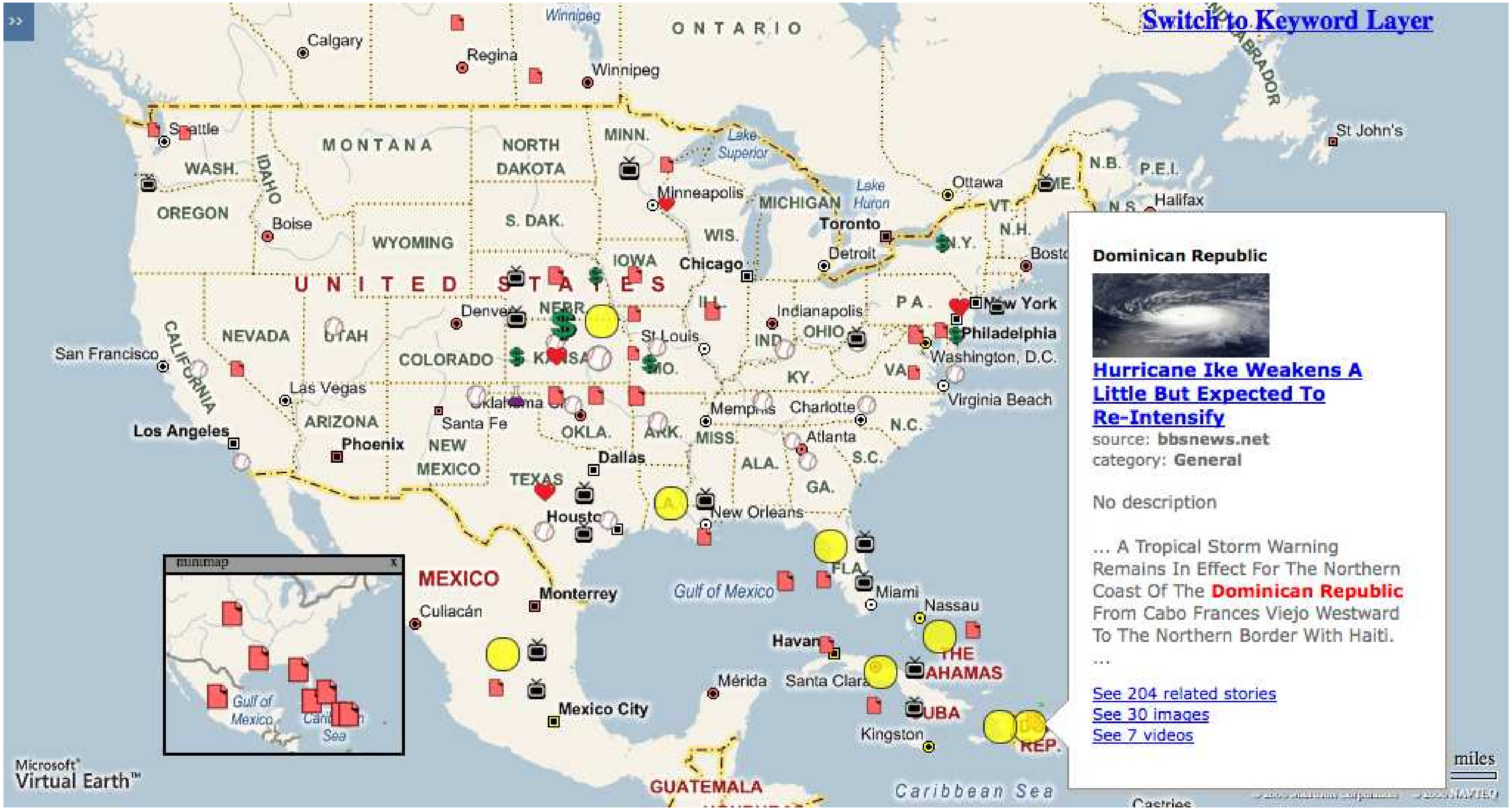


http://www.cs.umd.edu/~brabec/sandjava/

NewsStand: Spatio-Textual Aggregation of News and Display

1. Crawls the web looking for news sources and feeds
   - Indexing 8,000 news sources
   - About 50,000 news articles per day
2. Aggregate news articles by both content similarity and location
   - Articles about the same event are grouped into clusters
3. Rank clusters by importance which is based on:
   - Number of articles in cluster
   - Number of unique newspapers in cluster
   - Event's rate of propagation to other newspapers
4. Associate each cluster with its geographic focus or foci
5. Display each cluster at the positions of the geographic foci
6. Other options:
   - Category (e.g., General, Business, SciTech, Entertainment, Health, Sports)
   - Image and video galleries
   - Map stories by people, disease, etc.
   - User-generated news (e.g., Social networks such as Twitter)

NewsStand



`http://newsstand.umiacs.umd.edu`

`http://newsstand.umiacs.umd.edu/news/light`

TwitterStand: News from Tweets

- News gathering system using Twitter
- Twitter is a popular social networking website
  - Tweets are 140 character messages akin to SMS
  - Mostly non-news, often frivolous
- TwitterStand is a spontaneous news medium
  - Idea: users of Twitter help to gather news
    - Distributed news gathering
  - Scooping tool bypassing reporters or newspapers
    - E.g., Michael Jackson's death, Iranian election, Haitian earthquake

TwitterStand: News from Tweets

- News gathering system using Twitter
- Twitter is a popular social networking website
  - Tweets are 140 character messages akin to SMS
  - Mostly non-news, often frivolous
- TwitterStand is a spontaneous news medium
  - Idea: users of Twitter help to gather news
    - Distributed news gathering
  - Scooping tool bypassing reporters or newspapers
    - E.g., Michael Jackson's death, Iranian election, Haitian earthquake
- Key challenges:
  - Managing the deluge
    - Twitter is a noisy medium as most of the Tweets are not news
    - Challenge: extract news Tweets from mountain of non-news Tweets
  - Tweets are coming at a furious pace
  - Tweets capture the pulse of the moment
    - So, not a good strategy to store and process them in batches
    - TwitterStand uses online algorithms
    - Works without access to entire dataset (i.e., being offline)
    - Determine spatial focus of stories enabling news reading on map

Live Demo: TwitterStand System

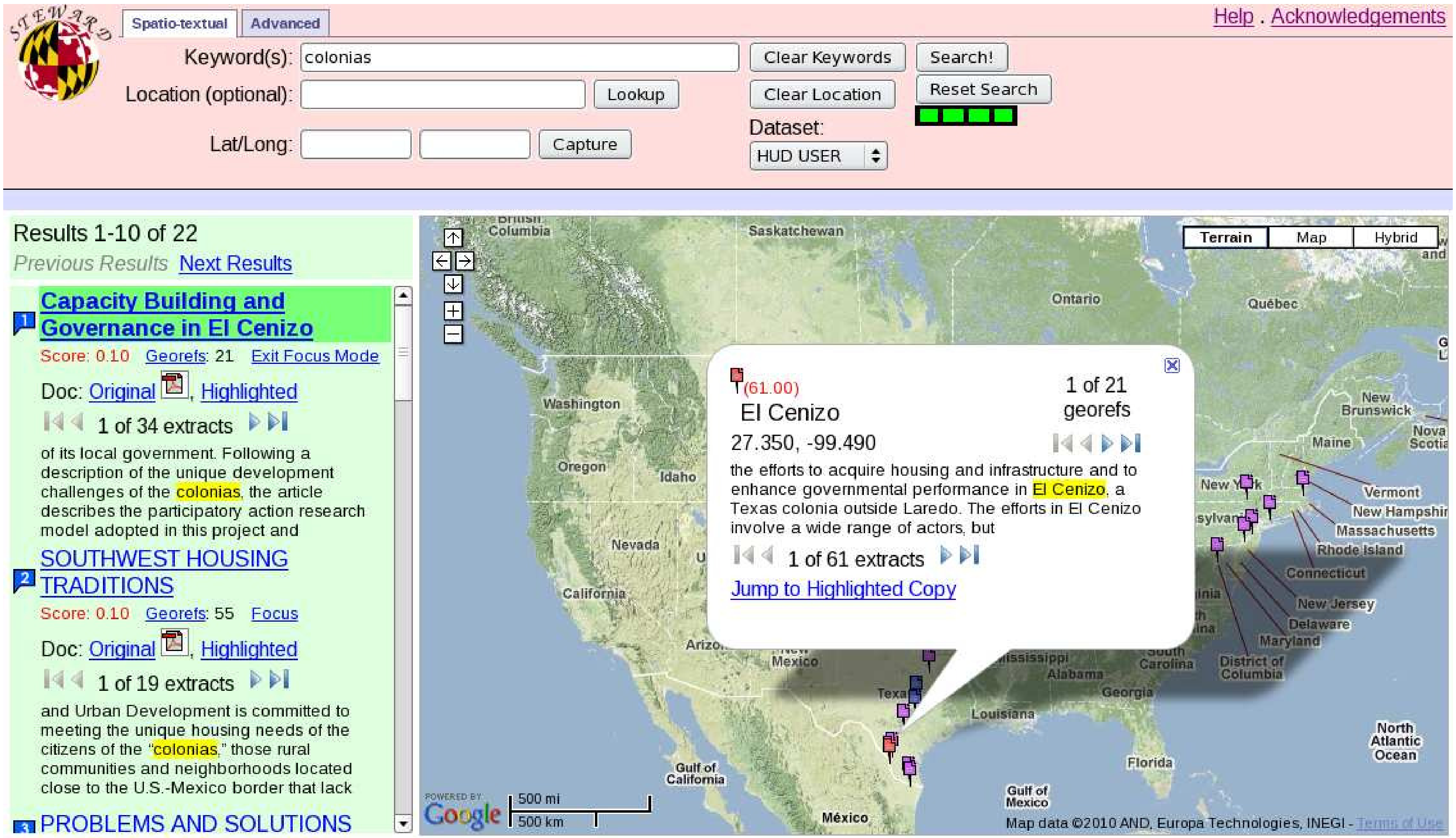
`http://twitterstand.umiacs.umd.edu/`

- What people are tweeting about rather than where they are tweeting from

STEWARD: A Spatio-Textual Search Engine

1. Spatio-Textual Extraction on the Web Aiding Retrieval of Documents
2. Sample spatio-textual query:
   - Keyword: “rock concert”
   - Location: near “College Park, MD’
3. Result documents are relevant to both keyword and location
   - Mention of rock concert
   - Spatial focus near “College Park, MD”
4. Issues with results from conventional search engines:
   - Is it the intended “College Park”?
   - What about spatial synonyms such as rock concerts in “Hyattsville” or “Greenbelt”?
   - Don’t usually understand the various forms of specifying geographic content
   - More than just postal addresses!
   - Results often based on other measures, e.g., link structure
5. Applied to HUD USER, PubMed, ProMED-mail, and news

Live Demo: STEWARD System



http://steward.umiacs.umd.edu

References

1. [Same06] H. Samet. Foundations of Multidimensional and Metric Data Structures. Morgan-Kaufmann, San Francisco, CA, USA, 2006.
2. [Same90a] H. Samet. The Design and Analysis of Spatial Data Structures, Addison-Wesley, Reading, MA, 1990.
3. [Same90b] H. Samet. Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS, Addison-Wesley, Reading, MA, 1990.
4. [Same08a] H. Samet. A Sorting Approach to Indexing Spatial Data, International Journal of Shape Modeling 14, 1(June 2008), pp. 15–37.
5. [Same08b] H. Samet, J. Sankaranarayanan, and H. Alborzi. Scalable network distance browsing in spatial databases, In Proc. of SIGMOD, pp. 43–54, Vancouver, Canada, Jun 2008. **(2008 ACM SIGMOD Best Paper Award)**
6. [Sank09] J. Sankaranarayanan, H. Samet, and H. Alborzi. Path oracles for spatial networks, In Proc. of VLDB, vol 2, pp. 1210–1221, Lyon, France, Aug 2009.
7. [Sank10] J. Sankaranarayanan and H. Samet. Query processing using distance oracles for spatial networks, IEEE Transactions on Knowledge and Data Engineering, 22(8):1158–1175, Aug 2010. **(Best Papers of ICDE 2009 Special Issue)**

Appeared in International Journal of Shape Modeling 14, *1(June 2008), pp. 15-37.*

A Sorting Approach to Indexing Spatial Data*

HANAN SAMET
Center for Automation Research
Institute for Advanced Computer Studies
Computer Science Department
University of Maryland
College Park, Maryland 20742, USA
hjs@cs.umd.edu http://www.cs.umd.edu/~hjs

August 9, 2010

Abstract

Spatial data is distinguished from conventional data by having extent. Therefore, spatial queries involve both the objects and the space that they occupy. The handling of queries that involve spatial data is facilitated by building an index on the data. The traditional role of the index is to sort the data, which means that it orders the data. However, since generally no ordering exists in dimensions greater than 1 without a transformation of the data to one dimension, the role of the sort process is one of differentiating between the data and what is usually done is to sort the spatial objects with respect to the space that they occupy. The resulting ordering is usually implicit rather than explicit so that the data need not be resorted (i.e., the index need not be rebuilt) when the queries change (e.g., the query reference objects). The index is said to order the space and the characteristics of such indexes are explored further.

1 Introduction

The representation of multidimensional data is an important issue in solid modeling as well as in many other diverse fields including computer-aided design (CAD), computational geometry, finite-element analysis, and computer graphics (e.g., [44, 45, 47]). The main motivation in choosing an appropriate representation is to facilitate operations such as search. This means that the representation involves sorting the data in some manner to make it more accessible. In fact, the term *access structure* or *index* is often used as an alternative to the term *data structure* in order to emphasize the importance of the connection to sorting.

The most common definition of "multidimensional data" is a collection of points in a higher dimensional space (i.e., greater than 1). These points can represent locations and objects in space as well as more general records where each attribute (i.e., field) corresponds to a dimension and only some, or even none, of the attributes are locational. As an example of nonlocational point data, consider an employee record that has attributes corresponding to the employee's name, address, gender, age, height, weight, and social security number (i.e., identity number). Such records arise in database management systems and can be treated as points in, for this example, a seven-dimensional space (i.e., there is one dimension for each attribute), although the different dimensions have different type units (i.e., name and address are strings of characters; gender is binary; while age, height, weight, and social security number are numbers some of which have are associated with different units). Note that the address attribute could also be interpreted in a locational sense using positioning coordinates such as latitude and longitude readings although the stringlike symbolic representation is far more common.

*This work was supported in part by the National Science Foundation under Grants EIA-00-91474, CCF-05-15241, and IIS-07-13501, Microsoft Research, NVIDIA, and the University of Maryland General Research Board.

When multidimensional data corresponds to locational data, we have the additional property that all of the attributes usually have the same unit (possibly with the aid of scaling transformations), which is distance in space. In this case, we can combine the distance-denominated attributes and pose queries that involve proximity. For example, we may wish to find the closest city to Chicago within the two-dimensional space from which the locations of the cities are drawn. Another query seeks to find all cities within 50 miles of Chicago. In contrast, such queries are not very meaningful when the attributes do not have the same type. Nevertheless, other queries such as range queries that seek, for example, all individuals born between 1940 and 1960 whose weight ranges between 150 and 200 pounds are quite common and can be posed regardless of the nature of the attributes.

When the range of multidimensional data spans a continuous physical space (i.e., an infinite collection of locations), the issues become more interesting. In particular, we are no longer just interested in the locations of objects, but, in addition, we are also interested in the space that they occupy (i.e., their extent). Some example objects with extent include lines (e.g., roads, rivers), intervals (which can correspond to time as well as space), regions of varying shape and dimensionality (e.g., lakes, counties, buildings, crop maps, polygons, polyhedra), and surfaces. The objects (when they are not points) may be disjoint or could even overlap.

The fact that the objects have extent has a direct effect on the type of indexes that we need. This can be best understood by examining the nature of the queries that we wish to support. For example, consider a database of objects. There are three types of queries that can be posed to such a database. The first is the set of queries about the objects themselves such as finding all objects that contain a given point or set of points, have a non-empty intersection with a given object, have a partial boundary in common, have a boundary in common, have any points in common, contain a given object, included in a given object, etc. The second consists of proximity queries such as the nearest object to a given point or object, and all objects within a given distance of a point or object (also known as a range or window query). The third consists of queries involving non-spatial attributes of objects such as given a point or object, finding the nearest object of a particular type, the minimum enclosing object of a particular type, or all the objects of a particular type whose boundary passes through it.

Being able to support the different types of queries described above has a direct effect on the type of indexes that are useful for such data. In particular, recall our earlier observation that a record in a conventional database may be considered as a point in a multidimensional space. For example, a straight line segment object having endpoints (x_1,y_1) and (x_2,y_2) can be transformed (i.e., represented) as the point (x_1,y_1,x_2,y_2) in a 4-d space (termed a *corner transformation* [50])[1]. This representation is good for queries about the line segments (the first type), while it is not good for proximity queries (i.e., the second and third type) since points outside the object are not mapped into the higher dimensional space. In particular, the 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), thereby leading to large search regions. This is especially true if there is a great difference in the relative size of the two objects (e.g., a short line in proximity to a long line as in Figure 1). On the other hand, when the objects are small (e.g., their extent is small), then the method works reasonably well as the objects are basically point objects. The 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). Proponents of the transformation method argue that this problem can be overcome by projecting back to original space and indexing on the projection (e.g., [54]). However, at this point, it is not unreasonable to ask why we bother to make the transformation in the first place.

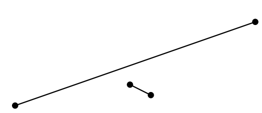


Figure 1: Example of two objects that are close to each other in the original space but are not clustered in the same region of the transformed space when using a transformation such as the corner transformation.

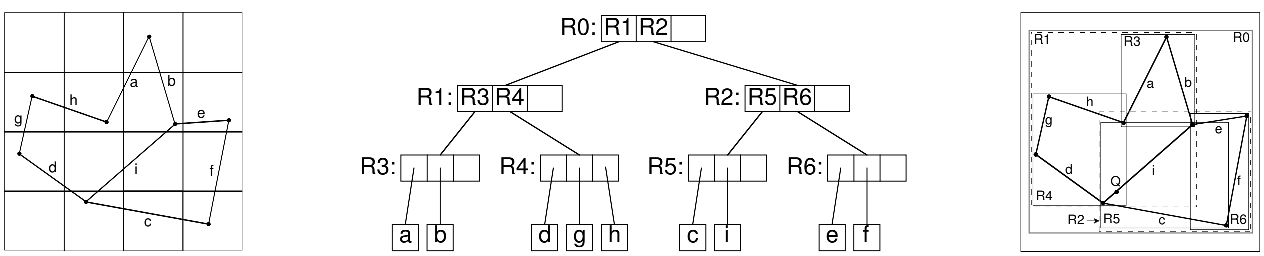
[1] Although for ease of visualization, our discussion and examples are in terms of line segment and rectangle objects, it is applicable to data of arbitrary dimension such as polyhedra and hyperrectangles.

It is important to observe that our notion of *sorting* spatial objects is more one of differentiating between the objects which is different from the conventional one which is intimately tied to the notion of providing an ordering. As we know, such an ordering implies a linearization which restricts the underlying data to one dimension, and such an ordering usually does not exist in dimensions d higher than one save for a dominance relationship (e.g., [39]) where point $a = \{a_i | 1 \leq i \leq d\}$ is said to dominate point $b = \{b_i | 1 \leq i \leq d\}$ if $b_i \leq a_i, 1 \leq i \leq d$. On the other hand, it is clear that the rationale for our discussion is that the data in which we are interested is of dimension greater than one. This leads to the conclusion that what is needed is an index that sorts (i.e., differentiates) between objects on the basis of spatial occupancy (i.e., their spatial extent). In other words, it sorts the objects relative to the space that they occupy, and this is the focus of the rest of this paper.

Before choosing a particular index we should also make sure that the following requirements are satisfied. First of all, the index should be compatible with the type of data (i.e., spatial objects) that is being stored. In other words, it should enable users to distinguish between different objects as well as render the search efficient in terms of pruning irrelevant objects from further consideration. Second, we must have an appropriate zero or reference point. In the case of spatial occupancy, this is usually some easily identified point or object (e.g., the origin of the multidimensional space from which the objects are drawn). Most importantly, given our observation about the absence of an ordering, it is best to have an implicit rather than an explicit index.

In particular, an implicit index is needed because it is impossible to foresee all possible queries in advance. For example, in the case of spatial relationships such as left, right, up, down, etc. it is impractical to have a data structure which has an attribute for every possible spatial relationship. In other words, the index should support the ability to derive the spatial relationships between the objects. It should be clear that an implicit index is superior to an explicit index, which, for example in the case of two-dimensional data such as the locations of cities, sorts the cities on the basis of their distance from a given point. The problem is that this sorting order is inapplicable to other reference points. In other words, having sorted all of the cities in the US with respect to their distance from Chicago, the result is useless if we want to find the closest city to New Orleans that satisfies a particular condition like having a population greater than 50,000 inhabitants. Therefore, having an implicit index means that we don't have to resort the data for queries other than updates.

2 Methods Based on Spatial Occupancy



(a) (b) (c)

Figure 2: (a) Example collection of straight line segments embedded in a 4×4 grid, (b) the object hierarchy for the R-tree corresponding to the objects in (a), and (c) the spatial extent of the minimum bounding rectangles corresponding to the object hierarchy in (b). Notice that the leaf nodes in (b) also store bounding rectangles although this is only shown for the nonleaf nodes.

The indexing methods that are based on sorting the spatial objects by spatial occupancy essentially decompose the underlying space from which the data is drawn into regions called *buckets* in the spirit of classical hashing methods, with the difference that the spatial indexing methods preserve order. In other words, objects in close proximity should be placed in the same bucket or at least in buckets that are close to each other in the sense of the order in which they would be accessed (i.e., retrieved from secondary storage in case of a false hit, etc.).

There are two principal methods of representing spatial data. The first is to use an object hierarchy that initially aggregates objects into groups, preferably based on their spatial proximity, and then uses proximity to further aggregate the groups thereby forming a hierarchy, where the number of objects that are aggregated in each node of the hierarchy is permitted to range between parameters $m \leq \lceil M/2 \rceil$ and M. The rationale for choosing this type of a range is for the hierarchy to mimic the behavior of a B-tree (e.g., [15]), where each element of the hierarchy acts like a disk page and thus is guaranteed to be half full, provided that $m = \lceil M/2 \rceil$.

Note that the object hierarchy is not unique as it depends on the manner in which the objects were aggregated to form the hierarchy (e.g., minimizing overlap between objects or coverage of the underlying space). Queries are facilitated by also associating a minimum bounding box with each object and group of objects as this enables a quick way to test if a point can possibly lie within the area spanned by the object or group of objects. A negative answer means that no further processing is required for the object or group while a positive answer means that further tests must be performed. Thus the minimum bounding box serves to avoid wasting work. Equivalently, it serves to differentiate (i.e., "sort") between occupied and unoccupied space. Data structures that make use of axis-aligned bounding boxes (AABB) such as the R-tree [23] and the R*-tree [10] illustrate the use of this method, as well as the more general oriented bounding box (OBB) where the sides are orthogonal, while no longer having to be parallel to the coordinate axes (e.g., [22, 40]). In addition, some data structures use other shapes for the bounding boxes such as spheres (e.g., SS-tree [35, 61]), combinations of hyperrectangles and hyperspheres (e.g.,SR-tree [30]), truncated tetrahedra (e.g., prism tree [38]), as well as triangular pyramids which are 5-sided objects with two parallel triangular faces and three rectangular faces forming a three-dimensional pie slice (e.g., BOXTREE [9]). These data structures differ primarily in the properties of the bounding boxes, and their interrelationships, that they use to determine how to aggregate the bounding boxes, and, of course, the objects. Aggregation is an issue when the data structure is used in a dynamic environment, where objects are inserted and removed from the hierarchy thereby leading to elements that are full or sparse vis-a-vis the values of m and M.

As an example of an R-tree, consider the collection of straight line segment objects given in Figure 2(a) shown embedded in a 4×4 grid. Figure 2(b) is an example of the object hierarchy induced by an R-tree for this collection, with $m = 2$ and $M = 3$. Figure 2(c) shows the spatial extent of the bounding rectangles of the nodes in Figure 2(a), with heavy lines denoting the bounding rectangles corresponding to the leaf nodes, and broken lines denoting the bounding rectangles corresponding to the subtrees rooted at the nonleaf nodes.

The drawback of the object hierarchy approach is that from the perspective of a space decomposition method, the resulting hierarchy of bounding boxes often leads to a non-disjoint decomposition of the underlying space. This means that if a search fails to find an object in one path starting at the root, then it is not necessarily the case that the object will not be found in another path starting at the root. This is the case in Figure 2(c) when we search for the line segment object that contains Q. In particular, we first visit nodes R1 and R4 unsuccessfully, and thus need to visit nodes R2 and R5 in order to find the correct line segment object i.

The second method is based on a decomposition (usually recursive) of the underlying space into disjoint blocks so that a subset of the objects is associated with each block. There are several ways to proceed. The first is to simply redefine the decomposition and aggregation associated with the object hierarchy method so that the minimum bounding boxes are decomposed into disjoint boxes, thereby also implicitly partitioning the underlying objects that they bound. In this case, the partition of the underlying space is heavily dependent on the data and is said to be at arbitrary positions. The k-d-B-tree [42] and the R$^+$-tree [51] are examples of such an approach, with the difference being that in the k-d-B-tree, the entire space which contains the objects is decomposed into subspaces and it is these subspaces that are aggregated, while in the R$^+$-tree, it is the bounding boxes that are decomposed and subsequently aggregated.

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

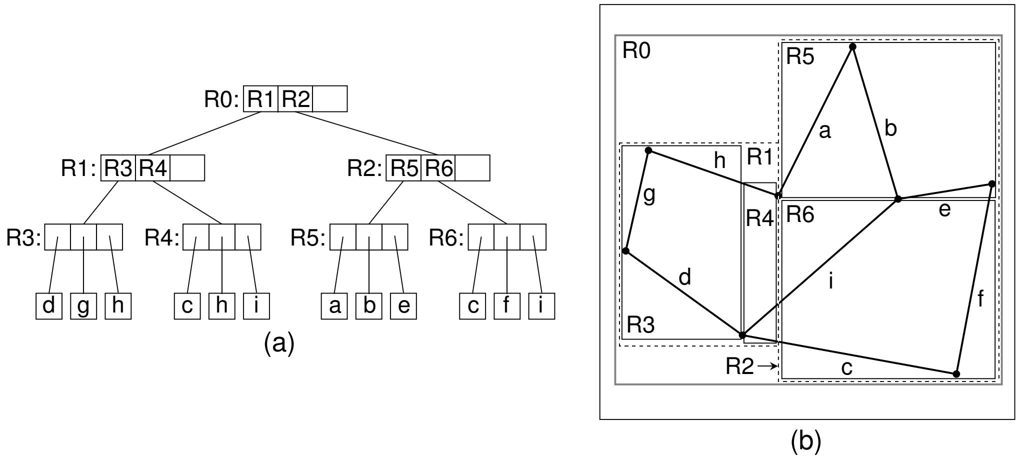


Figure 3: (a) R^+-tree for the collection of line segments in Figure 2(a) with m=2 and M=3, and (b) the spatial extents of the bounding rectangles. Notice that the leaf nodes in the index also store bounding rectangles although this is only shown for the nonleaf nodes.

The second way is to partition the underlying space into cells (i.e., blocks) at fixed positions so that all resulting cells are of uniform size, which is the case when using the uniform grid (e.g., [11, 33, 43]), also the standard indexing method for maps. Figure 2(a) is an example of a 4×4 uniform grid in which a collection of straight line segments has been embedded. One drawback of the uniform grid is the possibility of a large number of empty or sparsely-filled cells when the objects are not uniformly distributed, as well as the possibility that most of the objects will lie in a small subset of the cells. This is resolved by making use of a variable resolution representation such as one of the quadtree variants (e.g., [47]) where the subset of the objects that are associated with the cells is defined by placing an upper bound on the number of objects that can be associated with each cell. The cells that comprise the underlying space are recursively decomposed into congruent sibling cells whenever this upper bound is exceeded. Therefore, the upper bound serves as a *stopping condition* for the recursive decomposition process. An alternative, as exemplified by the PK-tree [46, 58], makes use of a lower bound on the number of objects that can be associated with each cell (termed an *instantiation* or *aggregation threshold*). Depending on the underlying representation that is used, the result can also be viewed as a hierarchy of congruent cells (see, e.g., the pyramid structure [55] which is a family of representations that make use of multiple resolution which can be characterized as image hierarchies [47]).

The PR quadtree [36, 45] is one example of a variable resolution representation for point objects where the underlying space in which a set of point objects lie is recursively decomposed into four equal-sized square-shaped cells until each cell is empty or contains just one object. For example, Figure 4 is the PR quadtree for the set of point objects A–F and P. The PR quadtree represents the underlying decomposition as a tree although our figure only illustrates the resulting decomposition of the underlying space into cells (i.e., the leaf nodes/blocks of the PR quadtree).

Turning to more complex such objects such as line segments, which have extent, we consider the PM_1 quadtree [49]. It is an example of a variable resolution representation for a collection of straight line segment objects such as the polygonal subdivision given in Figure 2(a). In this case, the stopping condition of its decomposition rule stipulates that partitioning occurs as long as a cell contains more than one line segment unless the line segments are all incident at the same vertex, which is also in the same cell (e.g., Figure 5(a)), The PM_1 quadtree and its variants are ideal for representing polygonal meshes as they provide an access structure to enable the quick determination of the polygon that contains a given point (i.e., a point location operation). In particular, the PM_2 quadtree [49], which differs from the PM_1 quadtree by permitting a cell c to contain several line segments as long as they are incident at the same vertex v regardless of whether or not v is in c (e.g., Figure 5(b)), is particularly suitable for representing triangular meshes [16]. A similar representation to the PM_1 quadtree has been devised for collections of three-dimensional objects such as

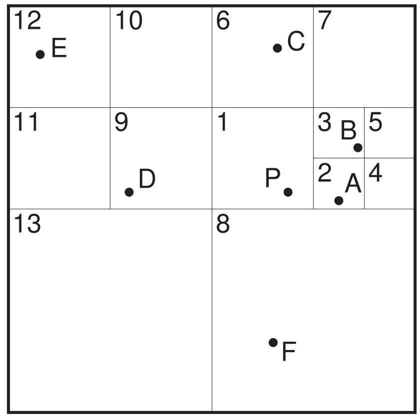


Figure 4: Block decomposition induced by the PR quadtree for the point objects A–F and P.

polyhedra images (e.g., [8] and the references cited in [47]). The decomposition criteria are such that no cell contains more than one face, edge, or vertex unless the faces all meet at the same vertex or are adjacent to the same edge.

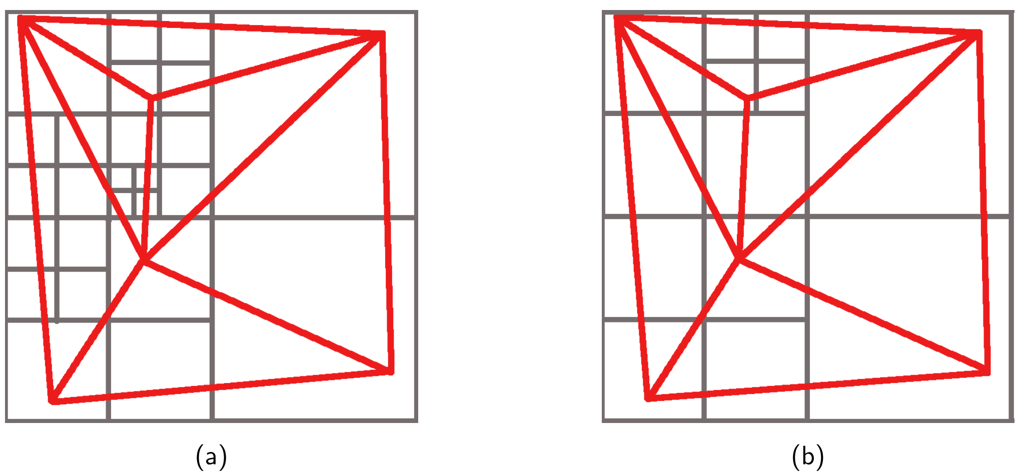


Figure 5: (a) PM_1 quadtree and (b) PM_2 quadtree for a collection of straight line segment objects that form a triangulation.

The above variants of the PM quadtree and PM octree represent an object by its boundary. The region quadtree [32] and region octree [27, 34] are variable resolution representations of objects by their interiors. In particular, the environment containing the objects is recursively decomposed into four or eight, respectively, rectangular congruent blocks until each block is either completely occupied by an object or is empty. For example, Figure 6(b) is the block decomposition for the region quadtree corresponding to the result of embedding the two-dimensional object in Figure 6(a) in an 8×8 grid, while Figure 7(b) is the block decomposition for the region octree corresponding to the three-dimensional staircaselike object in Figure 7(a).

Region octrees are also known as volumetric or voxel representations and are useful for medical applications. They are to be contrasted with procedural representations such as constructive solid geometry (CSG) [41] where primitive instances of objects are combined to form more complex objects by use of geometric transformations and regularized Boolean set operations (e.g., union, intersection). A disadvantage of the CSG representation is that it is not unique. In particular, there are frequently several ways of constructing an object (e.g., from different primitive elements). In addition, there is no overall notion of geometry except of the primitives that form each of the objects and thus there is no easy correlation between the objects and

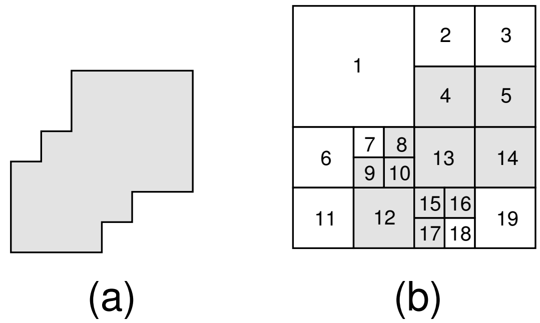


Figure 6: (a) Sample object, and (b) its region quadtree block decomposition with the blocks of the object being shaded, assuming that it is embedded in an 8×8 grid.

the space in which they are embedded unless techniques such as the PM-CSG tree [62] are used.

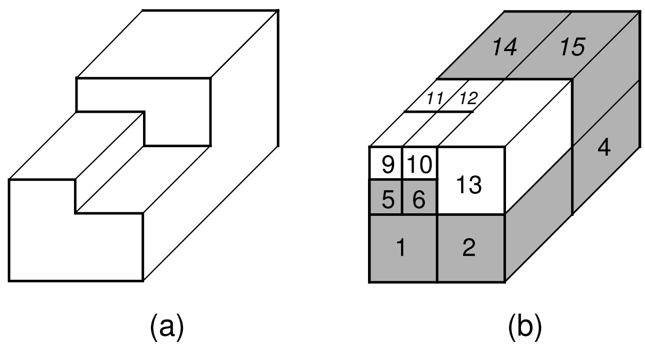


Figure 7: (a) Example three-dimensional object, and (b) its region octree block decomposition.

The principal drawback of the disjoint method is that when the objects have extent (e.g., line segments, rectangles, and any other non-point objects), then an object is associated with more than one cell when the object has been decomposed. This means that queries such as those that seek the length of all objects in a particular spatial region will have to remove duplicate objects before reporting the total length. Nevertheless, methods have been developed that avoid these duplicates by making use of the geometry of the type of the data that is being represented (e.g., [4, 5, 17]). Note that the result of constraining the positions of the partitions means that there is a limit on the possible sizes of the resulting cells (e.g., a power of 2 in the case of a quadtree variant). However, the result is that the underlying representation is good for operations between two different data sets as their representations are in registration (i.e., it is easy to correlate occupied and unoccupied space in the two data sets, which is not easy when the positions of the partitions are not constrained as is the case with methods rooted in representations based an object hierarchy even though the resulting decomposition of the underlying space is disjoint).

The PR, PM, and region quadtrees make use of a space hierarchy of where each level of the hierarchy contains congruent cells. The difference is that in the PR quadtree, each object is associated with just one cell, while in the PM and region quadtrees, the extent of the objects causes them to be decomposed into subobjects and thereby possibly be associated with more than one cell, although the cells are disjoint. At times, we want to use a space decomposition method that makes use of a hierarchy of congruent cells while still not decomposing the objects. In this case, we relax the disjointness requirement by stipulating that only the cells at a given level (i.e., depth) of the hierarchy must be disjoint. In particular, we recursively decompose the cells that comprise the underlying space into congruent sibling cells so that each object is associated with just one cell, and this is the smallest possible congruent cell that contains the object in its entirety. Assuming a top-down subdivision process that decomposes each cell into four square cells (i.e., a quadtree) at each level of decomposition, the result is that each object is associated with its minimum enclosing quadtree cell.

Subdivision ceases whenever a cell contains no objects. Alternatively, subdivision can also cease once a cell is smaller than a predetermined threshold size. This threshold is often chosen to be equal to the expected size of the objects. We use the term *MX-CIF quadtree* [1, 31] (see also the multilayer grid file [53], R-file [28], filter tree [52], and SQ-histogram [3]) to describe such a decomposition method.

In order to simplify our presentation, we assume that the objects stored in the MX-CIF quadtree are rectangles, although the MX-CIF quadtree is applicable to arbitrary objects in arbitrary dimensions in which case it keeps track of their minimum bounding boxes. For example, Figure 8b is the tree representation of the MX-CIF quadtree for a collection of rectangle objects given in Figure 8a. Note that objects can be associated with both terminal and non-terminal nodes of the tree.

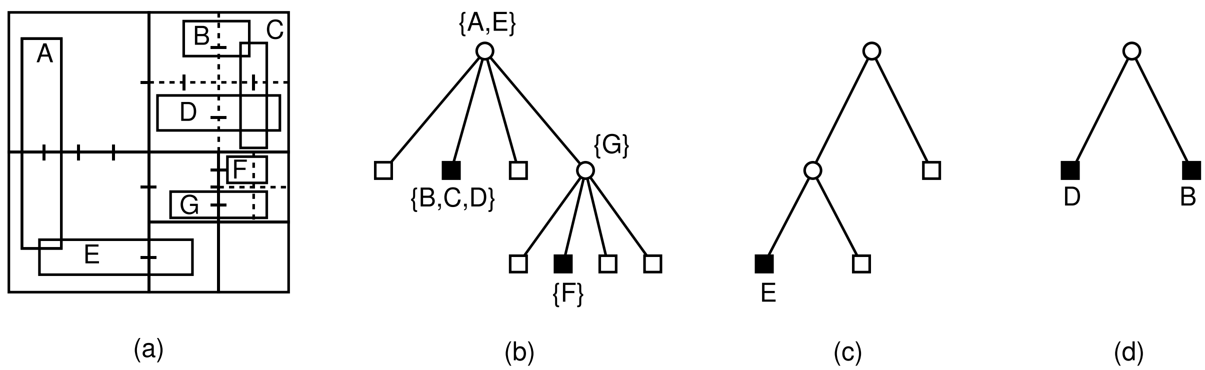


Figure 8: (a) Collection of rectangle objects and the cell decomposition induced by the MX-CIF quadtree; (b) the tree representation of (a); the binary trees for the y axes passing through the root of the tree in (b), and through (d) the NE son of the root of the tree in (b).

Since there is no limit on the number of objects that are associated with a particular cell, an additional decomposition rule is sometimes provided to distinguish between these objects. For example, in the case of the MX-CIF quadtree, a one-dimensional analog of the two-dimensional decomposition rule is used. In particular, all objects that are associated with a given cell b are partitioned into two sets: those that intersect (or whose sides are collinear) with the vertical axis passing through the center of b, and those that intersect (or whose sides are collinear) with the horizontal axis passing through the center of b. Objects that intersect with the center of b are associated with the horizontal axis. Associated with each axis is a one-dimensional MX-CIF quadtree (i.e., a binary tree), where each object o is associated with the node that corresponds to o's minimum enclosing interval. For example, Figure 8c and Figure 8d illustrate the binary trees associated with the y axes passing through the root and the NE son of the root, respectively, of the MX-CIF quadtree of Figure 8b. Thus we see that the two-dimensional MX-CIF quadtree acts like a hashing function with the one-dimensional MX-CIF quadtree playing the role of a collision resolution technique.

The MX-CIF quadtree can be interpreted as an object hierarchy where the objects appear at different levels of the hierarchy and the congruent cells play the same role as the minimum bounding boxes. The difference is that the set of possible minimum bounding boxes is constrained to the set of possible congruent cells. Thus, we can view the MX-CIF quadtree as a variable resolution R-tree. An alternative interpretation is that the MX-CIF quadtree provides a variable number of grids, each one being at half the resolution of its immediate successor, where an object is associated with the grid whose cells have the tightest fit. In fact, this interpretation forms the basis of the *filter tree* [52] and the *multilayer grid file* [53] where the only difference from the MX-CIF quadtree is the nature of the access structure for the cells (i.e., a hierarchy of grids based on a regular decomposition for the filter tree and based on a grid file for the multilayer grid file, and a tree structure for the MX-CIF quadtree).

One of the main drawbacks of the MX-CIF quadtree is that the size (i.e., width w) of the cell c corresponding to the minimum enclosing quadtree cell of object o's minimum enclosing bounding box b is not a function of the size of b or o. Instead, it is dependent on the position of o. In fact, c is often considerably larger than b thereby causing inefficiency in search operations due to a reduction in the ability to prune objects from further consideration. This situation arises whenever b overlaps the axes lines that pass through the center of c, and thus w can be as large as the width of the entire underlying space.

There are several ways of overcoming this drawback. One easy way is to introduce redundancy (i.e., representing the object several times thereby replicating the number of references to it) by decomposing the quadtree cell c into smaller quadtree cells, each of which minimally encloses some portion of o (or, alternatively, some portion of o's minimum enclosing bounding box b) and contains a reference to o. The expanded MX-CIF quadtree [2] is a simple example of such an approach where c is decomposed once into four subblocks c_i, which are then decomposed further until obtaining the minimum enclosing quadtree cell s_i for the portion of o, if any, that is covered by c_i. A more general approach. used in spatial join algorithms [29], sets a bound on the number of replications, (termed a *size bound* [37] and used in the GESS method [18]) or on the size of the covering quadtree cells resulting from the decomposition of c that contain the replicated references (termed an *error bound* [37]).

Replicating the number of references to the objects is reminiscent of the manner in which the non-disjointness of the decomposition of the underlying space resulting from the use of an object hierarchy was overcome, and thus has the same shortcoming of possibly requiring the application of a duplicate object removal step prior to reporting the answer to some queries. The *cover fieldtree* [19, 20], and the equivalent *loose quadtree* (*loose octree* in three dimensions) [57], adopt a different approach at overcoming the independence of the sizes of c and b drawback. In particular, they do not replicate the objects. Instead, they expand the size of the space that is spanned by each quadtree cell c of width w by a cell expansion factor p ($p > 0$) so that the expanded cell is of width $(1+p)\cdot w$. In this case, an object is associated with its minimum enclosing expanded quadtree cell. It has been shown that given a quadtree cell c of width w and cell expansion factor p, the radius r of the minimum bounding box b of the smallest object o that could possibly be associated with c must be greater than $pw/4$ [57]. However, the utility of the loose quadtree is best evaluated in terms of the inverse of this relation (i.e., the maximum possible width w of c given an object o with minimum bounding box b of radius r) as reducing w is the primary motivation for the development of the loose quadtree as an alternative to the MX-CIF quadtree.

It has been shown [48] that the maximum possible width w of c given an object o with minimum bounding box b of radius r is just a function of r and p and is independent of the position of o. More precisely, taking the ratio of cell to bounding box width $w/(2r)$, we have [48]:

$$1/(1+p) \leq w/(2r) \leq 1/p.$$

In particular, the range of possible ratios of width $w/(2r)$ as a function of p for $p \geq 1$ takes on at most two values, and usually just one value [48].

The ideal value for p is 1 [57]. The rationale is that using cell expansion factors much smaller than 1 increases the likelihood that the minimum enclosing expanded quadtree cell is large (as is the case for the MX-CIF quadtree, where $p = 0$), and that letting p be much larger than 1 results in the areas spanned by the expanded quadtree cells being too large, thereby having much overlap. For example, letting $p = 1$, Figure 9 is the loose quadtree corresponding to the collection of objects in Figure 8(a) and its MX-CIF quadtree in Figure 8(b). In this example, there are only two differences between the loose and MX-CIF quadtrees:

1. Rectangle object E is associated with the SW child of the root of the loose quadtree instead of with the root of the MX-CIF quadtree.

2. Rectangle object B is associated with the NW child of the NE child of the root of the loose quadtree instead of with the NE child of the root of the MX-CIF quadtree.

Note that the loose quadtree (cover fieldtree) is not the only approach at overcoming the drawback of the MX-CIF quadtree. In particular, the partition fieldtree [19, 20] is an alternative method of overcoming the drawback of the MX-CIF quadtree. The partition fieldtree proceeds by shifting the positions of the centroids of cells at successive levels of subdivision by one-half the width of the cell that is being subdivided. Figure 10 shows an example of such a subdivision. This subdivision rule guarantees that the width w of the minimum enclosing quadtree cell for the minimum bounding box b for object o is bounded by eight times the maximum extent r of b [20, 47]. The same ratio is obtained for the cover fieldtree when $p = 1/4$, and thus the partition fieldtree is superior to the cover fieldtree when $p < 1/4$ [47].

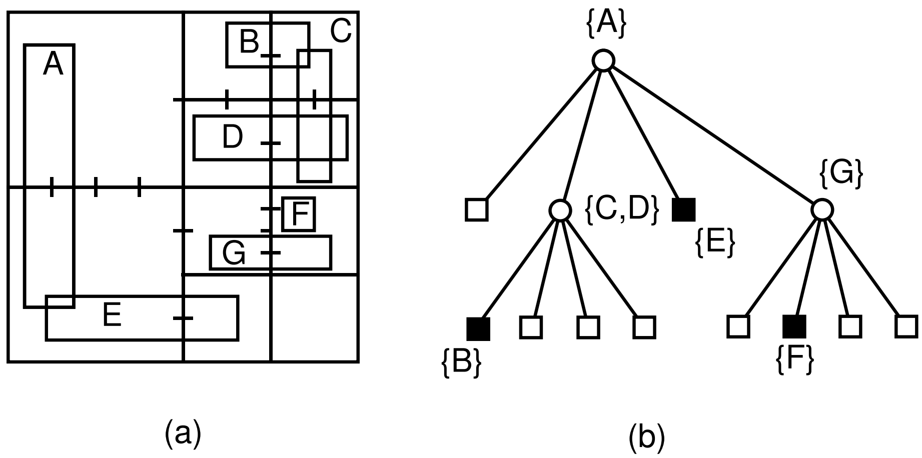


Figure 9: (a) Cell decomposition induced by the loose quadtree for a collection of rectangle objects identical to those in Figure 8(a), and (b) its tree representation.

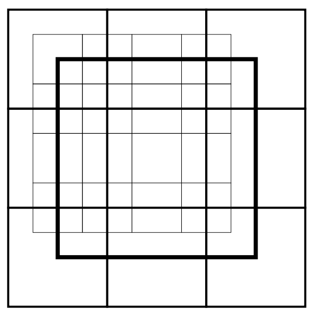


Figure 10: Example of the subdivision induced by a partition fieldtree.

3 Examples of the Utility of Sorting

As an example of the utility of sorting spatial data suppose that we want to determine the nearest object to a given point (i.e., a "pick" operation in computer graphics). In order to see how the search is facilitated by sorting the underlying data, consider the set of point objects A–F in Figure 4 which are stored in a PR quadtree [36, 45], and let us find the nearest neighbor of P. The search must first determine the leaf that contains the location/object whose nearest neighboring object is sought (i.e., P). Assuming a tree-based index, this is achieved by a top-down recursive algorithm. Initially, at each level of the recursion, we explore the subtree that contains P. Once the leaf node containing P has been found (i.e., 1), the distance from P to the nearest object in the leaf node is calculated (empty leaf nodes have a value of infinity). Next, we unwind the recursion so that at each level, we search the subtrees that represent regions overlapping a circle centered at P whose radius is the distance to the closest object that has been found so far. When more than one subtree must be searched, the subtrees representing regions nearer to P are searched before the subtrees that are farther away (since it is possible that an object in them might make it unnecessary to search the subtrees that are farther away).

In our example, the order in which the nodes are visited is given by their labels. We visit the brothers of the node 1 containing the query point P (and all remaining nodes at each level) in the order of the minimum distance from P to their borders (i.e., SE, NW, and NE for node 1). Therefore, as we unwind for the first time, we visit the eastern brother of node 1 and its subtrees (nodes 2 and 3 followed by nodes 4 and 5), node 6, and node 7. Note that once we have visited node 2, there is no need to visit node 4 since node 2 contains A. However, we must still visit node 3 containing point B (closer than A), but now there is no need to visit node 5. Similarly, there is no need to visit nodes 6 and 7 as they are too far away from P given our knowledge of A. Unwinding one more level reveals that due to the distance between P and A, we must visit node 8 as it could contain a point that is closer to P than A; however, there is no need to visit nodes 9, 10, 11, 12, and 13.

The algorithm that we described can also be adapted to find the k nearest neighbors in which case the pruning of objects that cannot serve as the k nearest neighbors is achieved by making use of the distance to the kth nearest object that has been found so far. Having retrieved the k closest objects, should we be interested in retrieving an additional object (i.e., the $k+1$th nearest object), then we have to reinvoke the algorithm again to find the $k+1$ nearest objects. An alternative approach is incremental and makes use of a priority queue [24, 25, 26] so that there is no need to look again for the neighboring objects that have been reported so far.

There are many other applications where the sorting of objects is useful, and below we review a few that arise in computer graphics. For example, sorting forms the basis of all operations on z buffers, visibility calculations (e.g., BSP trees [21]), as well as back-to-front and front-to-back display algorithms. It also forms the basis of Warnock's hidden-line [59] and hidden-surface [60] algorithms that repeatedly subdivide the picture area into successively smaller blocks while simultaneously searching it for areas that are sufficiently simple to be displayed. It is also used to accelerate ray tracing by finding ray-object intersections (e.g., [7]).

4 Concluding Remarks

An overview has been given of the rationale for sorting spatial objects in order to be able to index them thereby facilitating a number of operations involving search in the multidimensional domain. A distinction has been made between spatial objects that could be represented by traditional methods that have been applied to point data and those that have extent thereby rendering the traditional methods inapplicable.

Sorting is also used as the basis of an index in an environment where the data is drawn from a metric space rather than a vector space. In this case, the only information that we have is a distance function d (often a matrix) that indicates the degree of similarity (or dissimilarity) between all pairs of objects, given a set of N objects. Usually, it is required that d obey the triangle inequality, be nonnegative, and be symmetric, in which case it is known as a *metric* and also referred to as a *distance metric*. Indexes in such an environment are based on either picking one distinguished object p and a value r, and then recursively subdividing the remaining objects into two classes depending on a comparison of their distance from p with r, or by choosing two distinguished objects p_1 and p_2 and recursively subdividing the remaining objects into two classes depending on which of p_1 or p_2 is closer (e.g., [47, 56]). The difference between these methods and those for data that lies in a vector space is that the subdivision lines in the embedding space from which the objects are drawn are explicit for the vector space while they are implicit for the metric space (see [47] for more details).

The functioning of these various spatial sorting methods can be experienced by trying VASCO [12, 13, 14], a system for Visualizing and Animating Spatial Constructs and Operations. VASCO consists of a set of spatial index JAVA™ (e.g., [6]) applets that enable users on the worldwide web to experiment with a number of hierarchical representations (e.g., [44, 45, 47]) for different spatial data types, and see animations of how they support a number of search queries (e.g., nearest neighbor and range queries). The VASCO system can be found at `http://cs.umd.edu/~hjs/quadtree/`.

Acknowledgments

I am deeply grateful to Jagan Sankaranarayanan for help with the figures and preparation for publication.

References

[1] D. J. Abel and J. L. Smith. A data structure and algorithm based on a linear key for a rectangle retrieval problem. *Computer Vision, Graphics, and Image Processing*, 24(1):1–13, October 1983.

[2] D. J. Abel and J. L. Smith. A data structure and query algorithm for a database of areal entities. *Australian Computer Journal*, 16(4):147–154, November 1984.

[3] A. Aboulnaga and J. F. Naughton. Accurate estimation of the cost of spatial selections. In *Proceedings of the 16th IEEE International Conference on Data Engineering*, pages 123–134, San Diego, CA, February 2000.

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

[5] W. G. Aref and H. Samet. Hashing by proximity to process duplicates in spatial databases. In *Proceedings of the 3rd International Conference on Information and Knowledge Management (CIKM)*, pages 347–354, Gaithersburg, MD, December 1994.

[6] K. Arnold and J. Gosling. *The JAVA™ Programming Language*. Addison-Wesley, Reading, MA, 1996.

[7] J. Arvo and D. Kirk. A survey of ray tracing acceleration techniques. In *An Introduction to Ray Tracing*, A. S. Glassner, ed., chapter 6, pages 201–262. Academic Press, New York, 1989.

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

[9] G. Barequet, B. Chazelle, L. J. Guibas, J. S. B. Mitchell, and A. Tal. BOXTREE: a hierarchical representation for surfaces in 3D. In *Proceedings of the EUROGRAPHICS'96 Conference*, J. Rossignac and F. X. Sillion, eds., pages 387–396, 484, Poitiers, France, August 1996. Also in *Computer Graphics Forum*, 15(3):387–396, 484, August 1996.

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

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

[12] F. Brabec and H. Samet. The VASCO R-tree JAVA™ applet. In *Visual Database Systems (VDB4). Proceedings of the IFIP TC2//WG2.6 Fourth Working Conference on Visual Database Systems*, pages 147–153, Chapman and Hall, L'Aquila, Italy, May 1998.

[13] F. Brabec and H. Samet. Visualizing and animating R-trees and spatial operations in spatial databases on the worldwide web. In *Visual Database Systems (VDB4). Proceedings of the IFIP TC2//WG2.6 Fourth Working Conference on Visual Database Systems*, pages 123–140, Chapman and Hall, L'Aquila, Italy, May 1998.

[14] F. Brabec and H. Samet. Visualizing and animating search operations on quadtrees on the worldwide web. In *Proceedings of the 16th European Workshop on Computational Geometry*, pages 70–76, Eilat, Israel, March 2000.

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

[16] L. De Floriani, M. Facinoli, P. Magillo, and D. Dimitri. A hierarchical spatial index for triangulated surfaces. In *Proceedings of the Third International Conference on Computer Graphics Theory and Applications (GRAPP 2008)*, J. Braz, N. Jardim Nunes, and J. Madeiras Pereira, eds., pages 86–91, Funchal, Madeira, Portugal, January 2008.

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

[18] J.-P. Dittrich and B. Seeger. GESS: a scalable similarity-join algorithm for mining large data sets in high dimensional spaces. In *Proceedings of the 7th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 47–56, San Francisco, August 2001.

[19] A. Frank. Problems of realizing LIS: storage methods for space related data: the fieldtree. Technical Report 71, Institute for Geodesy and Photogrammetry, ETH, Zurich, Switzerland, June 1983.

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

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

[22] S. Gottschalk, M. C. Lin, and D. Manocha. OBBTree: a hierarchical structure for rapid interference detection. In *Proceedings of the SIGGRAPH'96 Conference*, pages 171–180, New Orleans, LA, August 1996.

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

[24] A. Henrich. A distance-scan algorithm for spatial access structures. In *Proceedings of the 2nd ACM Workshop on Geographic Information Systems*, N. Pissinou and K. Makki, eds., pages 136–143, Gaithersburg, MD, December 1994.

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

[26] G. R. Hjaltason and H. Samet. Distance browsing in spatial databases. *ACM Transactions on Database Systems*, 24(2):265–318, June 1999. Also University of Maryland Computer Science Technical Report TR–3919, July 1998.

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

[28] A. Hutflesz, H.-W. Six, and P. Widmayer. The R-file: an efficient access structure for proximity queries. In *Proceedings of the 6th IEEE International Conference on Data Engineering*, pages 372–379, Los Angeles, February 1990.

[29] E. Jacox and H. Samet. Spatial join techniques. *ACM Transactions on Database Systems*, 32(1):7, March 2007. Also an expanded version in University of Maryland Computer Science Technical Report TR–4730, June 2005.

[30] N. Katayama and S. Satoh. The SR-tree: an index structure for high-dimensional nearest neighbor queries. In *Proceedings of the ACM SIGMOD Conference*, J. Peckham, ed., pages 369–380, Tucson, AZ, May 1997.

[31] G. Kedem. The quad-CIF tree: a data structure for hierarchical on-line algorithms. In *Proceedings of the 19th Design Automation Conference*, pages 352–357, Las Vegas, NV, June 1982. Also University of Rochester Computer Science Technical Report TR–91, September 1981.

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

[33] D. E. Knuth. *The Art of Computer Programming: Sorting and Searching*, vol. 3. Addison-Wesley, Reading, MA, second edition, 1998.

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

[35] S. M. Omohundro. Five balltree construction algorithms. Technical Report TR–89–063, International Computer Science Institute, Berkeley, CA, December 1989.

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

[37] J. A. Orenstein. Redundancy in spatial databases. In *Proceedings of the ACM SIGMOD Conference*, pages 294–305, Portland, OR, June 1989.

[38] J. Ponce and O. Faugeras. An object centered hierarchical representation for 3d objects: the prism tree. *Computer Vision, Graphics, and Image Processing*, 38(1):1–28, April 1987.

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

[40] D. R. Reddy and S. Rubin. Representation of three-dimensional objects. Computer Science Technical Report CMU–CS–78–113, Carnegie-Mellon University, Pittsburgh, PA, April 1978.

[41] A. A. G. Requicha and H. B. Voelcker. Solid modeling: a historical summary and contemporary assessment. *IEEE Computer Graphics and Applications*, 2(2):9–24, March 1982.

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

[43] J. B. Rothnie Jr. and T. Lozano. Attribute based file organization in a paged memory environment. *Communications of the ACM*, 17(2):63–69, February 1974.

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

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

[46] H. Samet. Decoupling partitioning and grouping: overcoming shortcomings of spatial indexing with bucketing. *ACM Transactions on Database Systems*, 29(4):789–830, December 2004.

[47] H. Samet. *Foundations of Multidimensional and Metric Data Structures*. Morgan-Kaufmann, San Francisco, 2006.

[48] H. Samet and J. Sankaranarayanan. Maximum containing cell sizes in cover fieldtrees and loose quadtrees and octrees. Computer Science Technical Report TR–4900, University of Maryland, College Park, MD, October 2007.

[49] H. Samet and R. E. Webber. Storing a collection of polygons using quadtrees. *ACM Transactions on Graphics*, 4(3):182–222, July 1985.

[50] B. Seeger and H.-P. Kriegel. Techniques for design and implementation of efficient spatial access methods. In *Proceedings of the 14th International Conference on Very Large Databases (VLDB)*, F. Bachillon and D. J. DeWitt, eds., pages 360–371, Los Angeles, August 1988.

[51] T. Sellis, N. Roussooulos, and C. Faloutsos. The R^+-tree: a dynamic index for multi-dimensional objects. In *Proceedings of the 13th International Conference on Very Large Databases (VLDB)*, pages 71–79, Brighton, United Kingdom, September 1987.

[52] K. Sevcik and N. Koudas. Filter trees for managing spatial data over a range of size granularities. In *Proceedings of the 22nd International Conference on Very Large Data Bases (VLDB)*, T. M. Vijayaraman, A. P. Buchmann, C. Mohan, and N. L. Sarda, eds., pages 16–27, Mumbai (Bombay), India, September 1996.

[53] H.-W. Six and P. Widmayer. Spatial searching in geometric databases. In *Proceedings of the 4th IEEE International Conference on Data Engineering*, pages 496–503, Los Angeles, February 1988.

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

[55] S. L. Tanimoto and T. Pavlidis. A hierarchical data structure for picture processing. *Computer Graphics and Image Processing*, 4(2):104–119, June 1975.

[56] J. K. Uhlmann. Satisfying general proximity/similarity queries with metric trees. *Information Processing Letters*, 40(4):175–179, November 1991.

[57] T. Ulrich. Loose octrees. In *Game Programming Gems*, M. A. DeLoura, ed., pages 444–453. Charles River Media, Rockland, MA, 2000.

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

[59] J. E. Warnock. A hidden line algorithm for halftone picture representation. Computer Science Technical Report TR 4–5, University of Utah, Salt Lake City, UT, May 1968.

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

[61] D. A. White and R. Jain. Similarity indexing with the SS-tree. In *Proceedings of the 12th IEEE International Conference on Data Engineering*, S. Y. W. Su, ed., pages 516–523, New Orleans, LA, February 1996.

[62] G. Wyvill and T. L. Kunii. A functional model for constructive solid geometry. *Visual Computer*, 1(1):3–14, July 1985.

Sorting in Space: Multidimensional, Spatial, and Metric Data Structures for Applications in Spatial Databases, Geographic Information Systems (GIS), and Location-Based Services

Hanan Samet
Center for Automation Research, Institute for Advanced Computer Studies
Department of Computer Science, University of Maryland
College Park, MD 20742 USA
hjs@cs.umd.edu

***Abstract*— Techniques for representing multidimensional, spatial, and metric data for applications in spatial databases, geographic information systems (GIS), and location-based services are reviewed. This includes both geometric and textual representations of spatial data.**

I. Introduction

The representation of multidimensional, spatial, and metric data is an important issue in applications of spatial database, geographic information systems (GIS), and location-based services. This is in part a direct result of the increasing popularity of web-based services such as Microsoft Bing Maps and Google Maps and Earth, as well as their deployment on gesturing-based devices such as smartphones and tablets which have also brought Apple into the picture [68]. This popularity has led to an increase in the awareness of the importance of location as an attribute in a database. The existence of the database means that the data stored therein must be retrieved and this involves searching. The efficiency of searching is dependent on the extent to which the underlying data is sorted. The conventional definition of the term *sort* is that it is a verb meaning: (1) To put in a certain place or rank according to kind, class, or nature. (2) To arrange according to characteristics. The sorting is encapsulated by the data structure used to represent the spatial data thereby making it more accessible. In fact, the term *access structure* or *index* is often used as an alternative to the term *data structure* in order to emphasize the importance of the connection to sorting.

Notwithstanding the above definition, sorting usually implies the existence of an ordering. Orderings are fine for one-dimensional data. For example, in the case of individuals we can sort them by their weight, and given an individual such as Bill, we can use the ordering to find the person closest in weight to Bill. Similarly, we can use the same ordering to also find the person closest in weight to John. Unfortunately, in two dimensions and higher, such a solution does not always work. In particular, suppose we sort all of the cities in the US by their distance from Chicago. This is fine for finding the closest city to Chicago, say with population greater than 200,000. However, we cannot use the same ordering to find the closest city to New York, say with population greater than 200,000, without resorting the cities.

The problem is that for two dimensions and higher, the notion of an ordering does not exist unless a dominance relation holds (e.g., [44])—that is, a point $a = \{a_i | 1 \leq i \leq d\}$ is said to dominate a point $b = \{b_i | 1 \leq i \leq d\}$ if $a_i \leq b_i, 1 \leq i \leq d$. Thus the only way to ensure that an ordering exists is to linearize the data as can be done, for example, using a space-filling curve (e.g., [47], [64]). The problem with such an approach is that the ordering is explicit. Instead, what is needed is an implicit ordering so that we do not need to resort the data when, for example in our sample query, the reference point for the query changes (e.g., from Chicago to New York). Such an ordering is a natural byproduct when we sort objects by spatial occupancy, and is the subject of this paper.

II. Methods Based on Spatial Occupancy

The indexing methods that are based on sorting the spatial objects by spatial occupancy essentially decompose the underlying space from which the data is drawn into regions called *buckets* in the spirit of classical hashing methods. The difference is that the spatial indexing methods preserve order. In other words, objects in close proximity should be in the same bucket or at least in buckets that are close to each other in the sense of the order in which they would be accessed (i.e., retrieved from secondary storage in case of a false hit, etc.).

There are two principal methods of representing spatial data. The first is to use an object hierarchy that initially aggregates objects into groups based on their spatial proximity and then uses proximity to further aggregate the groups thereby forming a hierarchy. Note that the object hierarchy is not unique as it depends on the manner in which the objects were aggregated to form the hierarchy. Queries are facilitated by also associating a minimum bounding box with each object and group of objects as this enables a quick way to test if a point can possibly lie within the area spanned by the object or group of objects. A negative answer means that no further processing is required for the object or group, while a positive answer means that further tests must be performed. Thus the minimum bounding box serves to avoid wasting work. Data structures such as the R-tree [16] and the R*-tree [6] illustrate the use of this method.

The drawback of the object hierarchy approach is that from the perspective of a space decomposition method, the resulting hierarchy of bounding boxes leads to a non-disjoint decomposition of the underlying space. This means that if a search fails to find an object in one path starting at the root, then it is not necessarily the case that the object will not be found in another path starting at the root.

The second method is based on a recursive decomposition of the underlying space into disjoint blocks so that a subset of the objects are associated with each block. There are several ways to proceed. The first is to simply redefine the decomposition and aggregation associated with the object hierarchy method so that the minimum bounding rectangles are decomposed into disjoint rectangles, thereby also implicitly partitioning the underlying objects that they bound. In this case, the partition of the underlying space is heavily dependent on the data and is said to be at arbitrary positions. The k-d-B-tree [46] and the R^+-tree [88] are examples of such an approach.

The second way is to partition the underlying space at fixed positions so that all resulting cells are of uniform size, which is the case when using the uniform grid (e.g., [29]), also the standard indexing method for maps. The drawback of the uniform grid is the possibility of a large number of empty or sparsely-filled cells when the objects are not uniformly distributed. This is resolved by making use of a variable resolution representation such as one of the quadtree variants (e.g., [64]) where the subset of the objects that are associated with the blocks are defined by placing an upper bound on the number of objects that can be associated with each block (termed a *stopping condition* for the recursive decomposition process) and also often referred to as a *bucket capacity*. In this case we can say that the objects are sorted into cells which act like bins (i.e., buckets). The PR quadtree [43], [62] and its bucket variants are examples of such a structure for points, while the PM quadtree family [21], [37], [72], [79] (see also the related PMR quadtree [19], [40], [41]) is an example of a variable resolution representation for collections of straight line segment objects such as those found in polygonal subdivisions as well as higher dimensions (e.g., faces of three-dimensional objects as in the PM octree [5]). An alternative, as exemplified by the PK-tree [63], [97], makes use of a lower bound on the number of objects that can be associated with each block (termed an *instantiation* or *aggregation* threshold).

Quadtrees [24], [28] and their three-dimensional octree analogs [23], [39]. have also been used widely for representing and operating on region data in two and three dimensions, respectively (e.g., [59]). In particular, algorithms have been devised for converting between them and numerous representations such as binary arrays [48], boundary codes [14], [49], [78], rasters [50], [56], [89], medial axis transforms [55], [57], terrain models [91], boundary models [92], constructive solid geometry (CSG) [73], as well as operations such as connected component labeling [52], [75], [76], perimeters [51], [74], distance [53], image dilation [1], computing Euler numbers [13], and ray tracing [60]. Many of these operations are implemented by traversing the actual quadtrees/octrees and performing the operation on each node and its neighbors [31], [54], [58], [60], [71]. Quadtrees and their variants are to be distinguished from pyramids (e.g., [93]) which are multiresolution data structures useful in spatial data mining [2].

The principal drawback of the disjoint method is that when the objects have extent (e.g., line segments, rectangles, and any other non-point objects), then an object may be associated with more than one block. This means that queries such as those that seek the length of all objects in a particular spatial region will have to remove duplicate objects before reporting the total length. Nevertheless, methods have been developed that avoid these duplicates by making use of the geometry of the type of the data that is being represented (e.g., [3], [4], [12]). Note that the result of constraining the positions of the partitions means that there is a limit on the possible sizes of the resulting cells (e.g., a power of 2 in the case of a quadtree variant). However, this means that the underlying representation is good for operations between two different data sets (e.g., a spatial join [22], [25], [26]) as their representations are in registration (i.e., it is easy to correlate occupied and unoccupied space in the two data sets, which is not easy when the positions of the partitions are not constrained as is the case with methods rooted in representations based an object hierarchy even though the resulting decomposition of the underlying space is disjoint). For an empirical comparison of these representations with respect to multidimensional point data, see [27].

III. Future Trends

In this paper, the discussion has been in the context of the traditional explicit specification geometric representation of spatial data (e.g., as latitude-longitude pairs of numbers). This is often cumbersome as users don't always think of a location in this way, and often don't know it in this way or have easy access to it, and, more importantly, are not accustomed to communicate it to others in this way. Instead, they are accustomed to specify a location textually (including verbally). A textual specification has a number of advantages. The first is that it is easy to communicate especially on smartphone devices where a textual (also increasingly verbal via speech recognition such as Siri on the Apple platform) input capability is always present. Another important advantage is that the text acts like a polymorphic type in the sense that one size fits all. In particular, depending on the application which makes use of this information, a term such as "Washington" can be interpreted both as a point or as an area, and the user need not be concerned with this question. The drawback of the textual specification of location data is that it is ambiguous. In particular, there are many possible locations named "Washington" and they must be resolved (i.e., "toponym resolution") [33], [35], [45]. Moreover, in some cases we are not even sure that the term "Washington" denotes a location as it could be a reference to the name of a person (i.e., "toponym recognition") [32]. This can be the case when processing documents such as newspaper articles [34], [67], [77], [96], tweets [86], blogs, etc. Being able to handle such specifications enables the development of map query interfaces to a wide range of spatially-referenced data thereby enabling the development of new applications such as disease tracking [30] as well as the hidden web [36]. Moreover, such interfaces enable the search to make use of spatial synonyms which result in nearest neighbor computation where the results are names of the neighbors rather than their coordinate values.

IV. Concluding Remarks

Sorting spatial and metric data is particularly useful for proximity queries usually where proximity is measured in terms of as "the crow flies" (e.g., [17], [18], [65]). However, these representations can also be used to support proximity in a graph such as a road network (e.g., [70], [80], [81], [82], [83], [84], [85]). They can also be used with different metrics such as a Hausdorff distance [42].

Interestingly, methods analogous to those that we described have also been used in cases where the only information that we have available is a distance function that indicates the degree of similarity (or dis-similarity) between all pairs of the N objects. Usually the distance function d is required to obey the triangle inequality, be non-negative, and be symmetric, in which case it is known as a *metric* and also referred to as a *distance metric*. Given a distance function, we usually partition and index the objects with respect to their distance from a few selected objects. There are two basic partitioning schemes: *ball partitioning* and *generalized hyperplane partitioning* [20]. In ball partitioning, the data set is partitioned based on distances from one distinguished object, into the subset that is inside and the subset that is outside a ball around the object In generalized hyperplane partitioning, two distinguished objects p_1 and p_2 are chosen and the data set is partitioned into two sets based on which of the two distinguished objects is the closest. It is interesting to observe that both schemes achieve a partitioning of the underlying data set into spatial-like zones. However, the difference is that the boundaries of the zones are more well-defined in the case of ball partitioning methods as they can be expressed explicitly using a small number of objects and a known distance value. In contrast, in the case of generalized hyperplane partitioning methods, the boundaries of the zones are usually expressed implicitly in terms of the distinguished objects, instead of explicitly, which may require quite a bit of computation to determine. In fact, very often, the boundaries cannot be expressed explicitly as, for example, in the case of an arbitrary metric space (in contrast to a Euclidean space) where we do not have a direct representation of the 'generalized hyperplane' that separates the two partitions.

The functioning of the various spatial sorting methods can be experienced by trying VASCO [7], [8], [9], [11], a system for Visualizing and Animating Spatial Constructs and Operations. VASCO consists of a set of spatial index JAVA^{TM} applets that enable users on the worldwide web to experiment with a number of hierarchical representations (e.g., [61], [62], [64]) for different spatial data types, and see animations of how they support a number of search queries (e.g., nearest neighbor and range queries). The VASCO system can be found at `http://www.cs.umd.edu/~hjs/quadtree/`. For an example of their use in a spatial database/geographic information system (GIS), see the SAND Spatial Browser [10], [15], [66] and the QUILT system [69], [90]. Such systems find use in many application domains (e.g., digital government [38], point clouds [87] and in peer-to-peer settings [94], [95]).

ACKNOWLEDGMENT

This work was supported in part by the National Science Foundation under Grants IIS-10-18475 and IIS-12-19023, and by Google Research.

REFERENCES

[1] C.-H. Ang, H. Samet, and C. A. Shaffer. A new region expansion for quadtrees. *IEEE TPAMI*, 12(7):682–686, July 1990.

[2] W. G. Aref and H. Samet. Efficient processing of window queries in the pyramid data structure. In *PODS'90*, pp. 265–272, Nashville, TN, Apr. 1990.

[3] W. G. Aref and H. Samet. Uniquely reporting spatial objects: yet another operation for comparing spatial data structures. In *SDH'92*, pp. 178–189, Charleston, SC, Aug. 1992.

[4] W. G. Aref and H. Samet. Hashing by proximity to process duplicates in spatial databases. In *CIKM'94*, pp. 347–354, Gaithersburg, MD, Dec. 1994.

[5] D. Ayala, P. Brunet, R. Juan, and I. Navazo. Object representation by means of nonminimal division quadtrees and octrees. *TODS*, 4(1):41–59, Jan. 1985.

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

[7] F. Brabec and H. Samet. The VASCO R-tree JAVA^{TM} applet. In *Visual Database Systems (VDB4)*, pp. 147–153, L'Aquila, Italy, May 1998.

[8] F. Brabec and H. Samet. Visualizing and animating R-trees and spatial operations in spatial databases on the worldwide web. In *Visual Database Systems (VDB4)*, pp. 123–140, L'Aquila, Italy, May 1998.

[9] F. Brabec and H. Samet. Visualizing and animating search operations on quadtrees on the worldwide web. In *Proc. 16th European Workshop on Computational Geometry*, pp. 70–76, Eilat, Israel, Mar. 2000.

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

[11] F. Brabec, H. Samet, and C. Yilmaz. VASCO: visualizing and animating spatial constructs and operations. In *Proc. 19th Annual Symposium on Computational Geometry*, pp. 374–375, San Diego, CA, June 2003.

[12] J.-P. Dittrich and B. Seeger. Data redundancy and duplicate detection in spatial join processing. In *ICDE*, pp. 535–546, San Diego, CA, Feb. 2000.

[13] C. R. Dyer. Computing the Euler number of an image from its quadtree. *CGIP*, 13(3):270–276, July 1980.

[14] C. R. Dyer, A. Rosenfeld, and H. Samet. Region representation: boundary codes from quadtrees. *CACM*, 23(3):171–179, Mar. 1980.

[15] C. Esperança and H. Samet. Experience with SAND/Tcl: a scripting tool for spatial databases. *JVLC*, 13(2):229–255, Apr. 2002.

[16] A. Guttman. R-trees: a dynamic index structure for spatial searching. In *SIGMOD*, pp. 47–57, Boston, June 1984.

[17] A. Henrich. A distance-scan algorithm for spatial access structures. In *GIS'94*, pp. 136–143, Gaithersburg, MD, Dec. 1994.

[18] G. R. Hjaltason and H. Samet. Distance browsing in spatial databases. *TODS*, 24(2):265–318, June 1999.

[19] G. R. Hjaltason and H. Samet. Speeding up construction of PMR quadtree-based spatial indexes. *VLDBJ*, 11(2):109–137, Oct. 2002.

[20] G. R. Hjaltason and H. Samet. Index-driven similarity search in metric spaces. *TODS*, 28(4):517–580, Dec. 2003.

[21] E. G. Hoel and H. Samet. Efficient processing of spatial queries in line segment databases. In *SSD'91*, pp. 237–256, Zurich, Aug. 1991.

[22] E. G. Hoel and H. Samet. Benchmarking spatial join operations with spatial output. In *VLDB*, pp. 606–618, Zurich, Sept. 1995.

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

[24] G. M. Hunter and K. Steiglitz. Operations on images using quad trees. *IEEE TPAMI*, 1(2):145–153, Apr. 1979.

[25] E. Jacox and H. Samet. Iterative spatial join. *TODS*, 28(3):268–294, Sept. 2003.

[26] E. Jacox and H. Samet. Spatial join techniques. *TODS*, 32(1):7, Mar. 2007.

[27] Y. J. Kim and J. M. Patel. Rethinking choices for multi-dimensional point indexing: making the case for the often ignored quadtree. In *CIDR 2007*, pp. 281–291, Asilomar, CA, Jan. 2007.

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

[29] D. E. Knuth. *The Art of Computer Programming: Sorting and Searching*, volume 3. Addison-Wesley, Reading, MA, second edition, 1998.

[30] R. Lan, M. D. Lieberman, and H. Samet. The picture of health: map-based, collaborative spatio-temporal disease tracking. In *Proc. 1st ACM SIGSPATIAL International Workshop on the Use of GIS in Public Health (HealthGIS 2012)*, Redondo Beach, CA, Nov. 2012.

[31] M. Lee, L. De Floriani, and H. Samet. Constant-time neighbor finding in hierarchical tetrahedral meshes. In *SMI'01*, pp. 286–295, Genova, Italy, May 2001.

[32] M. D. Lieberman and H. Samet. Multifaceted toponym recognition for streaming news. In *SIGIR'11)*, pp. 843–852, Beijing, July 2011.

[33] M. D. Lieberman and H. Samet. Adaptive context features for toponym resolution in streaming news. In *SIGIR'12*, pp. 731–740, Portland, OR, Aug. 2012.

[34] M. D. Lieberman and H. Samet. Supporting rapid processing and interactive map-based exploration of streaming news. In *GIS'12*, Redondo Beach, CA, Nov. 2012.

[35] M. D. Lieberman, H. Samet, and J. Sankaranarayanan. Geotagging with local lexicons to build indexes for textually-specified spatial data. In *ICDE*, pp. 201–212, Long Beach, CA, Mar. 2010.

[36] M. D. Lieberman, H. Samet, J. Sankaranarayanan, and J. Sperling. STEWARD: architecture of a spatio-textual search engine. In *GIS'07*, pp. 186–193, Seattle, WA, Nov. 2007.

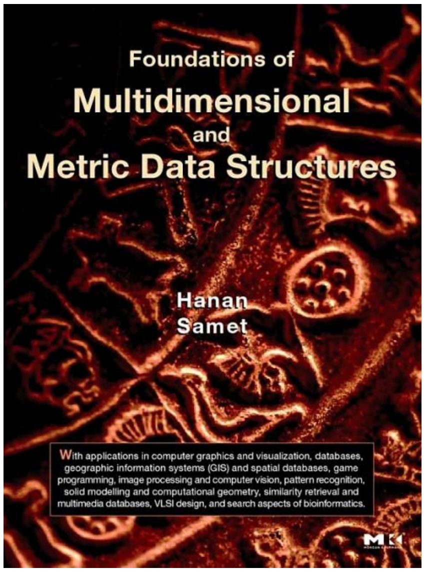
[37] M. Lindenbaum, H. Samet, and G. R. Hjaltason. A probabilistic analysis of trie-based sorting of large collections of line segments in spatial databases. *SIAM J. Comp.*, 35(1):22–58, Sep. 2005.
[38] G. Marchionini, H. Samet, and L. Brandt. Introduction to the digital government special issue. *CACM*, 46(1):24–27, Jan. 2003.
[39] D. Meagher. Geometric modeling using octree encoding. *CGIP*, 19(2):129–147, June 1982.
[40] R. C. Nelson and H. Samet. A consistent hierarchical representation for vector data. *Computer Graphics*, 20(4):197–206, Aug. 1986. Also in *SIGGRAPH*, Dallas, TX, Aug. 1986.
[41] R. C. Nelson and H. Samet. A population analysis for hierarchical data structures. In *SIGMOD*, pp. 270–277, San Francisco, May 1987.
[42] S. Nutanong, E. H. Jacox, and H. Samet. An incremental Hausdorff distance calculation algorithm. *PVLDB*, 4(8):506–517, Aug. 2011.
[43] J. A. Orenstein. Multidimensional tries used for associative searching. *INFOPL*, 14(4):150–157, June 1982.
[44] F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction*. Springer-Verlag, New York, 1985.
[45] G. Quercini, H. Samet, J. Sankaranarayanan, and M. D. Lieberman. Determining the spatial reader scopes of news sources using local lexicons. In *GIS'10*, pp. 43–52, San Jose, CA, Nov. 2010.
[46] J. T. Robinson. The K-D-B-tree: a search structure for large multidimensional dynamic indexes. In *SIGMOD*, pp. 10–18, Ann Arbor, MI, Apr. 1981.
[47] H. Sagan. *Space-Filling Curves*. Springer-Verlag, New York, 1994.
[48] H. Samet. Region representation: quadtrees from binary arrays. *CGIP*, 13(1):88–93, May 1980.
[49] H. Samet. Region representation: quadtrees from boundary codes. *CACM*, 23(3):163–170, Mar. 1980.
[50] H. Samet. An algorithm for converting rasters to quadtrees. *IEEE TPAMI*, 3(1):93–95, Jan. 1981.
[51] H. Samet. Computing perimeters of images represented by quadtrees. *IEEE TPAMI*, 3(6):683–687, Nov. 1981.
[52] H. Samet. Connected component labeling using quadtrees. *JACM*, 28(3):487–501, July 1981.
[53] H. Samet. Distance transform for images represented by quadtrees. *IEEE TPAMI*, 4(3):298–303, May 1982.
[54] H. Samet. Neighbor finding techniques for images represented by quadtrees. *CGIP*, 18(1):37–57, Jan. 1982.
[55] H. Samet. A quadtree medial axis transform. *CACM*, 26(9):680–693, Sept. 1983. Also see CORRIGENDUM, *CACM*, 27(2):151, Feb. 1984.
[56] H. Samet. Algorithms for the conversion of quadtrees to rasters. *CVGIP*, 26(1):1–16, Apr. 1984.
[57] H. Samet. Reconstruction of quadtrees from quadtree medial axis transforms. *CVGIP*, 29(3):311–328, Mar. 1985.
[58] H. Samet. A top-down quadtree traversal algorithm. *IEEE TPAMI*, 7(1):94–98, Jan. 1985.
[59] H. Samet. An overview of quadtrees, octrees, and related hierarchical data structures. In R. A. Earnshaw, editor, *Theoretical Foundations of Computer Graphics and CAD*, pp. 51–68. Springer-Verlag, Berlin, West Germany, 1988.
[60] H. Samet. Implementing ray tracing with octrees and neighbor finding. *Computers & Graphics*, 13(4):445–460, 1989.
[61] H. Samet. *Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS*. Addison-Wesley, Reading, MA, 1990.
[62] H. Samet. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, Reading, MA, 1990.
[63] H. Samet. Decoupling partitioning and grouping: overcoming shortcomings of spatial indexing with bucketing. *TODS*, 29(4):789–830, Dec. 2004.
[64] H. Samet. *Foundations of Multidimensional and Metric Data Structures*. Morgan-Kaufmann, San Francisco, 2006.
[65] H. Samet. K-nearest neighbor finding using MaxNearestDist. *IEEE TPAMI*, 30(2):243–252, Feb. 2008.
[66] H. Samet, H. Alborzi, F. Brabec, C. Esperança, G. R. Hjaltason, F. Morgan, and E. Tanin. Use of the SAND spatial browser for digital government applications. *CACM*, 46(1):63–66, Jan. 2003.
[67] H. Samet, M. D. Adelfio, B. C. Fruin, M. D. Lieberman, and B. E. Teitler. Porting a web-based mapping application to a smartphone app. In *GIS'11*, pp. 525–528, Chicago, November 2011.
[68] H. Samet, B. C. Fruin, and S. Nutanong. Duking it out at the smartphone mobile app mapping API corral: Apple, Google, and the competition. In *Proc. 1st ACM SIGSPATIAL Int. Wrkshp on Mobile Geog. Inf. Sys. (MobiGIS 2012)*, Redondo Beach, CA, Nov. 2012.
[69] H. Samet, A. Rosenfeld, C. A. Shaffer, and R. E. Webber. A geographic information system using quadtrees. *Pattern Recognition*, 17(6):647–656, Nov/Dec 1984.
[70] H. Samet, J. Sankaranarayanan, and H. Alborzi. Scalable network distance browsing in spatial databases. In *SIGMOD*, pp. 43–54, Vancouver, Canada, June 2008.
[71] H. Samet and C. A. Shaffer. A model for the analysis of neighbor finding in pointer-based quadtrees. *IEEE TPAMI*, 7(6):717–720, Nov. 1985.
[72] H. Samet, C. A. Shaffer, and R. E. Webber. Digitizing the plane with cells of non-uniform size. *INFOPL*, 24(6):369–375, Apr. 1987.
[73] H. Samet and M. Tamminen. Bintrees, CSG trees, and time. *Computer Graphics*, 19(3):121–130, July 1985. Also in *SIGGRAPH*, San Francisco, July 1985.
[74] H. Samet and M. Tamminen. Computing geometric properties of images represented by linear quadtrees. *IEEE TPAMI*, 7(2):229–240, Mar. 1985.
[75] H. Samet and M. Tamminen. An improved approach to connected component labeling of images. In *CVPR*, pp. 312–318, Miami Beach, FL, June 1986.
[76] H. Samet and M. Tamminen. Efficient component labeling of images of arbitrary dimension represented by linear bintrees. *IEEE TPAMI*, 10(4):579–586, July 1988.
[77] H. Samet, B. E. Teitler, M. D. Adelfio, and M. D. Lieberman. Adapting a map query interface for a gesturing touch screen interface. In *WWW'11 (Companion Volume)*, pp. 257–260, Hyderabad, India, Mar. 2011.
[78] H. Samet and R. E. Webber. On encoding boundaries with quadtrees. *IEEE TPAMI*, 6(3):365–369, May 1984.
[79] H. Samet and R. E. Webber. Storing a collection of polygons using quadtrees. *TOGS*, 4(3):182–222, July 1985.
[80] J. Sankaranarayanan, H. Alborzi, and H. Samet. Efficient query processing on spatial networks. In *GIS'05*, pp. 200–209, Bremen, Germany, Nov. 2005.
[81] J. Sankaranarayanan, H. Alborzi, and H. Samet. Distance join queries on spatial networks. In *GIS'06*, pp. 211–218, Arlington, VA, Nov. 2006.
[82] J. Sankaranarayanan and H. Samet. Distance oracles for spatial networks. In *ICDE*, pp. 652–663, Shanghai, Apr. 2009.
[83] J. Sankaranarayanan and H. Samet. Query processing using distance oracles for spatial networks. *IEEE TKDE*, 22(8):1158–1175, Aug. 2010.
[84] J. Sankaranarayanan and H. Samet. Roads belong in databases. *IEEE Data Engineering Bulletin*, 33(2):4–11, June 2010.
[85] J. Sankaranarayanan, H. Samet, and H. Alborzi. Path oracles for spatial networks. *PVLDB*, 2(1):1210–1221, Aug. 2009.
[86] J. Sankaranarayanan, H. Samet, B. Teitler, M. D. Lieberman, and J. Sperling. TwitterStand: News in tweets. In *GIS'09*, pp. 42–51, Seattle, WA, Nov. 2009.
[87] J. Sankaranarayanan, H. Samet, and A. Varshney. A fast all nearest neighbor algorithm for applications involving large point-clouds. *Computers & Graphics*, 31(2):157–174, Apr. 2007.
[88] T. Sellis, N. Roussopoulos, and C. Faloutsos. The R^+-tree: a dynamic index for multi-dimensional objects. In *VLDB*, pp. 71–79, Brighton, United Kingdom, Sept. 1987.
[89] C. A. Shaffer and H. Samet. Optimal quadtree construction algorithms. *CVGIP*, 37(3):402–419, Mar. 1987.
[90] C. A. Shaffer, H. Samet, and R. C. Nelson. QUILT: a geographic information system based on quadtrees. *IJGIS*, 4(2):103–131, Apr.–June 1990.
[91] R. Sivan and H. Samet. Algorithms for constructing quadtree surface maps. In *SDH'92*, volume 1, pp. 361–370, Charleston, SC, Aug. 1992.
[92] M. Tamminen and H. Samet. Efficient octree conversion by connectivity labeling. *Computer Graphics*, 18(3):43–51, July 1984. Also in *SIGGRAPH*, Minneapolis, MN, July 1984.
[93] S. L. Tanimoto and T. Pavlidis. A hierarchical data structure for picture processing. *CGIP*, 4(2):104–119, June 1975.
[94] E. Tanin, A. Harwood, and H. Samet. A distributed quadtree index for peer-to-peer settings. In *ICDE*, pp. 254–255, Tokyo, Apr. 2005.
[95] E. Tanin, A. Harwood, and H. Samet. Using a distributed quadtree index in P2P networks. *VLDBJ*, 16(2):165–178, Apr. 2007.
[96] B. Teitler, M. D. Lieberman, D. Panozzo, J. Sankaranarayanan, H. Samet, and J. Sperling. NewsStand: A new view on news. In *GIS'08*, pp. 144–153, Irvine, CA, Nov. 2008.
[97] W. Wang, J. Yang, and R. Muntz. PK-tree: a spatial index structure for high dimensional point data. In *FODO'90*, pp. 27–36, Kobe, Japan, Nov. 1998.

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 ▪ ~~$79.95~~ ▪ ~~£47.99~~ ▪ ~~€57.95~~ ▪ $63.96 ▪ £38.39 ▪ €46.36



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 ~~$79.95~~ • $63.96

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

Order from Morgan Kaufmann Publishers

To receive 20% off, please refer to code 85511

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