QUADTREE BACKGROUND

Hanan Samet

Computer Science Department and
Center for Automation Research and
Institute for Advanced Computer Studies
University of Maryland
College Park, Maryland 20742
e-mail: hjs@umiacs.umd.edu

HISTORY

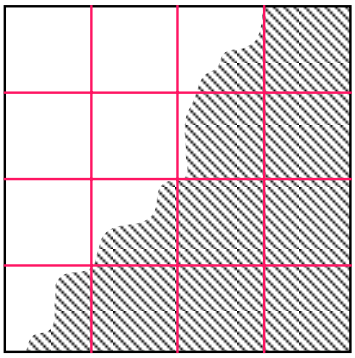
1. Linearization or ordering of higher dimensional space
   - space filling curves — e.g., Peano (1891)
   - spatial index — Morton (1966)
2. Computer graphics
   - sorting objects for display
   - Warnock's algorithm (1968)
     a. vector: hidden-line elimination
     b. raster: hidden-surface elimination
   - animation — Hunter (1978)
   - BSP trees — Fuchs, Kedem, and Naylor (1980)
3. Image processing and pattern recognition
   - Klinger (1971)
   - split-and-merge segmentation methods — Horowitz and Pavlidis (1976)
4. Multidimensional point representation
   - multidimensional binary search trees — Finkel and Bentley (1974)
   - k-d trees — Bentley (1975)
5. Volume data for solid modeling and computer vision
   - bounding boxes — Reddy and Rubin (1978)
   - octrees — Hunter (1978)
6. Finite element mesh generation — Rheinboldt and Mesztenyi (1980)
7. Fast matrix operations — Strassen (1969)
8. Computational complexity
   - dimension reducing device — Hunter (1978)
   - optimal placement — Li, Grosky, and Jain (1981)

SPLIT-AND-MERGE SEGMENTATION

- Subdivide an image until a homogeneity criterion is satisfied — e.g., standard deviation of gray levels is below a particular threshold
- Group adjacent blocks into maximal homogeneous regions
- Ex:

SPLIT-AND-MERGE SEGMENTATION

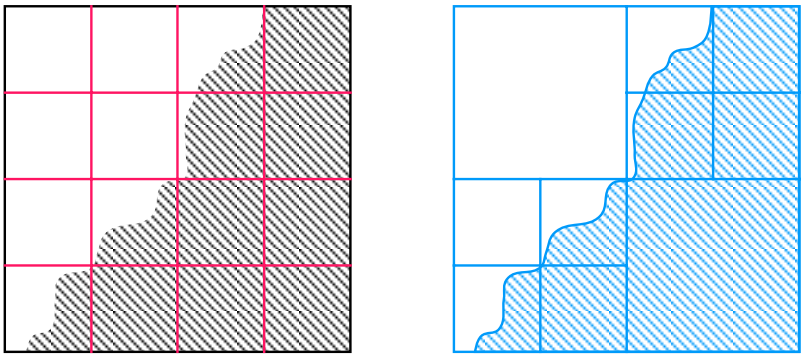
- Subdivide an image until a homogeneity criterion is satisfied — e.g., standard deviation of gray levels is below a particular threshold
- Group adjacent blocks into maximal homogeneous regions
- Ex:



1. initial image decomposition into cells of uniform size

SPLIT-AND-MERGE SEGMENTATION

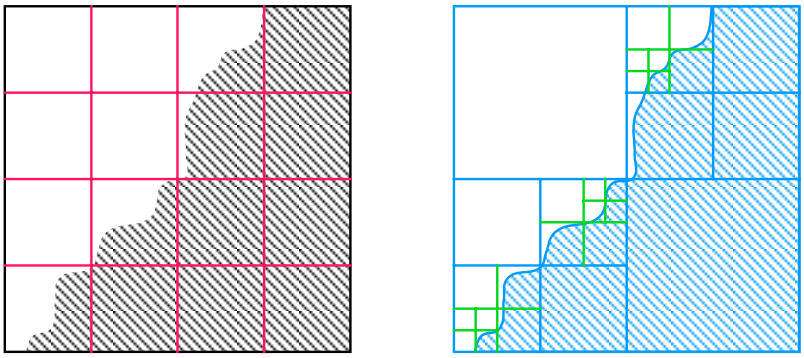
- Subdivide an image until a homogeneity criterion is satisfied — e.g., standard deviation of gray levels is below a particular threshold
- Group adjacent blocks into maximal homogeneous regions
- Ex:



1. initial image decomposition into cells of uniform size
2. merge homogeneous brothers

SPLIT-AND-MERGE SEGMENTATION

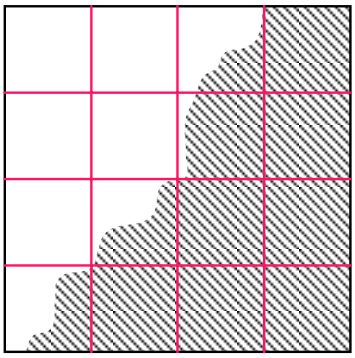
- Subdivide an image until a homogeneity criterion is satisfied — e.g., standard deviation of gray levels is below a particular threshold
- Group adjacent blocks into maximal homogeneous regions
- Ex:

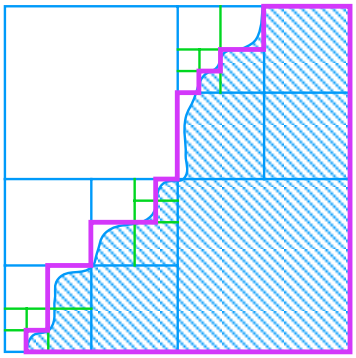


1. initial image decomposition into cells of uniform size
2. merge homogeneous brothers
3. split blocks that are not homogeneous

SPLIT-AND-MERGE SEGMENTATION

- Subdivide an image until a homogeneity criterion is satisfied — e.g., standard deviation of gray levels is below a particular threshold
- Group adjacent blocks into maximal homogeneous regions
- Ex:

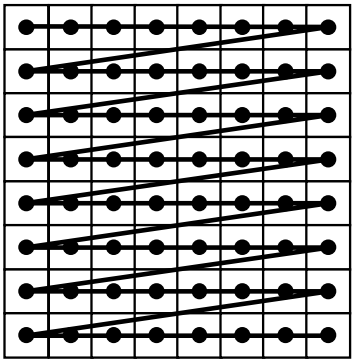




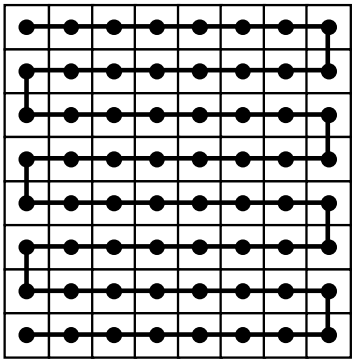
1. initial image decomposition into cells of uniform size
2. merge homogeneous brothers
3. split blocks that are not homogeneous
4. group identical blocks into regions

SPACE ORDERING METHODS

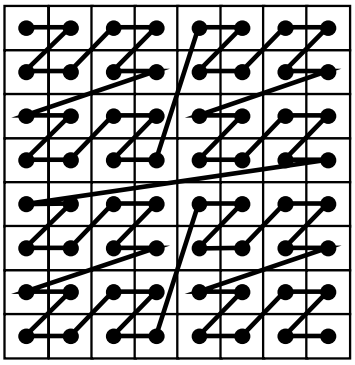
- Linearization of higher dimensional spaces
- Space-filling curves
- Examples:



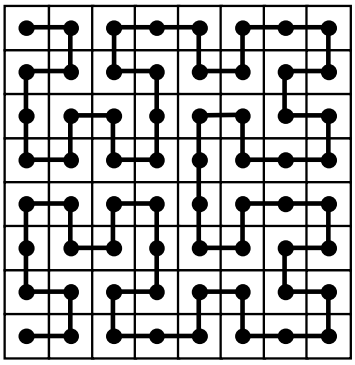
Row order



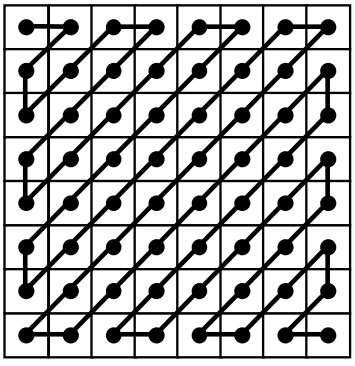
Row-prime order



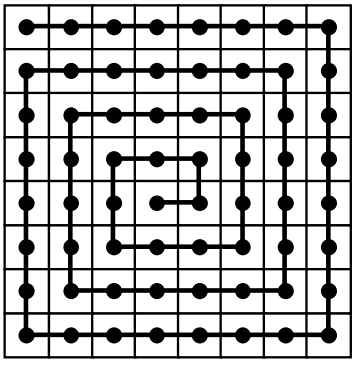
Morton order



Peano-Hilbert order



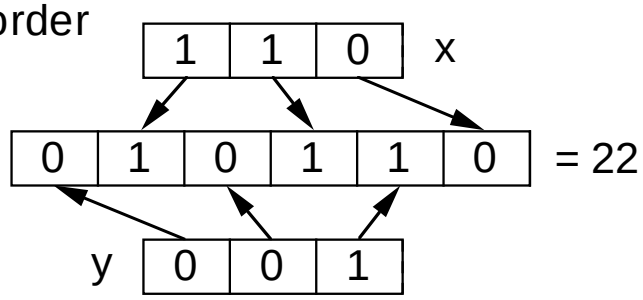
Cantor order



Spiral order

CONVERTING BETWEEN POINTS AND CURVES

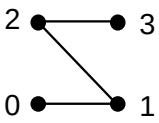
- Need to know size of image for all but the Morton order
- Relatively easy for all but the Peano-Hilbert order which is difficult (although possible) to decode and encode to obtain the corresponding *x* and *y* coordinate values
- Morton order
  1. use bit interleaving of binary representation of the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

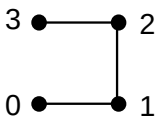


STABILITY OF SPACE ORDERING METHODS

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

Morton: Peano-Hilbert:

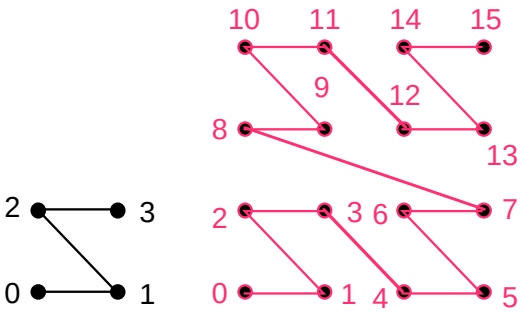




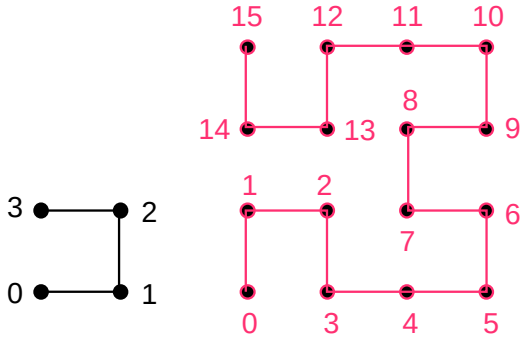
STABILITY OF SPACE ORDERING METHODS

- 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:

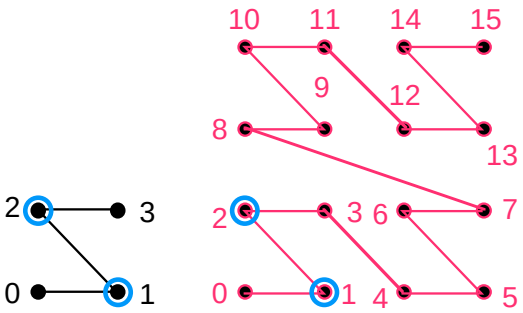


- Result of doubling the resolution (i.e., the coverage)

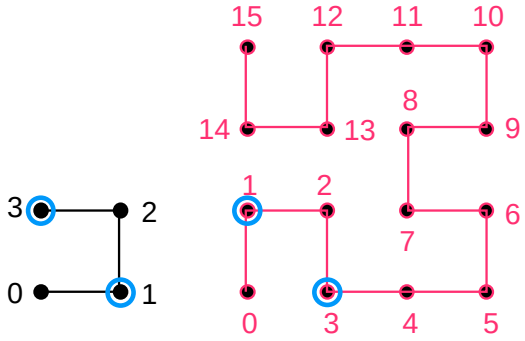
STABILITY OF SPACE ORDERING METHODS

- 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

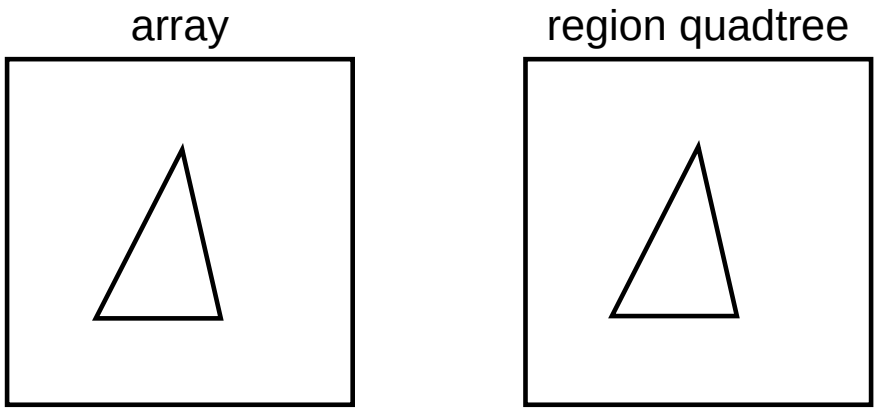
1. Pass through each point in the space once and only once
2. Two points that are neighbors in space are neighbors along the curve and vice versa
   - impossible to satisfy for all points at all resolutions
3. Easy to retrieve neighbors of a point
4. Curve should be stable as the space grows and contracts 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 and Morton orders are admissible
   - Peano-Hilbert order is not admissible
   - row-prime, Cantor, and spiral orders are admissible if permit the direction of the horizontal and vertical neighbors to vary from point to point
6. Easy to convert between two-dimensional data and the curve and vice-versa
   - easy for Morton order
   - difficult for Peano-Hilbert order
   - relatively easy for row, row-prime, Cantor, and spiral orders

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)

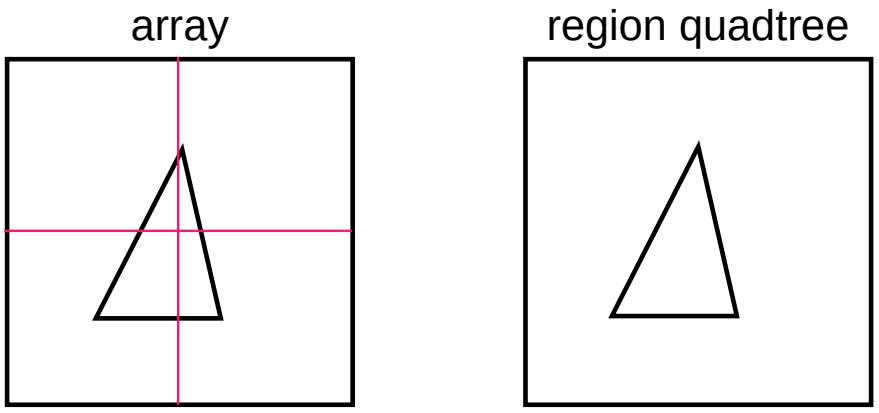
DIMENSION REDUCTION

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

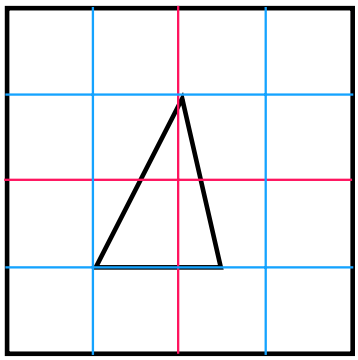
1. Number of blocks necessary to store a simple polygon as a region quadtree is proportional to its perimeter (Hunter)
   - implies that many quadtree algorithms execute in *O*(perimeter) time as they are tree traversals
   - the region quadtree is a dimension reducing device as perimeter (ignoring fractal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



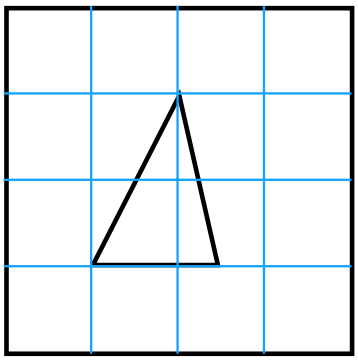
DIMENSION REDUCTION

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.

array

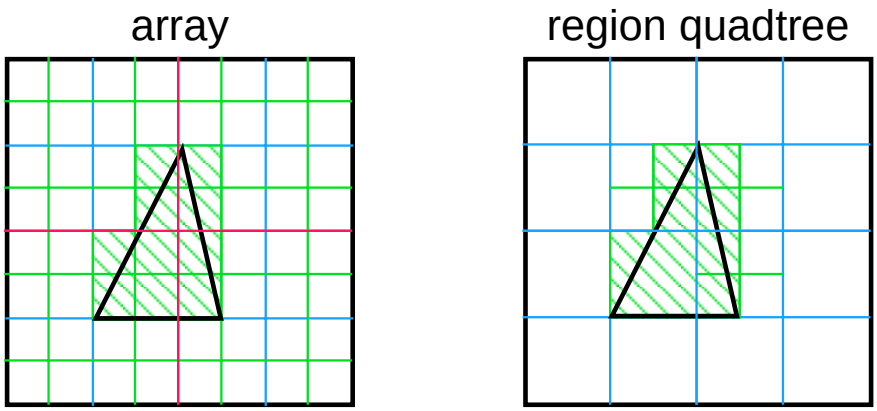


region quadtree



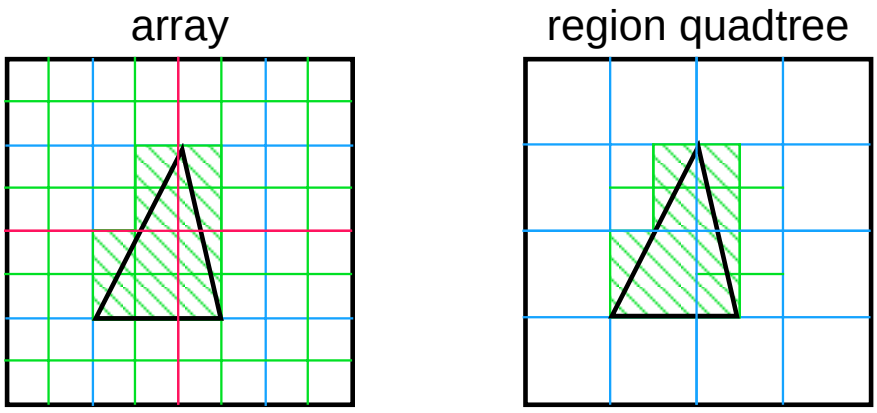
DIMENSION REDUCTION

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.



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

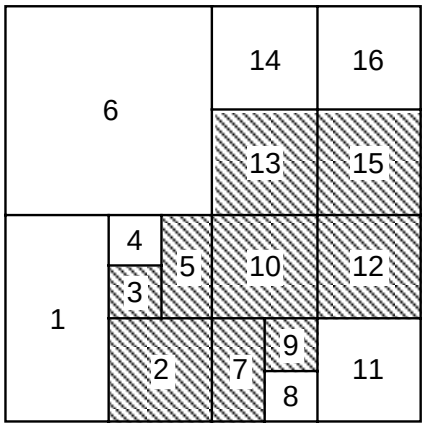
ALTERNATIVE REPRESENTATIONS

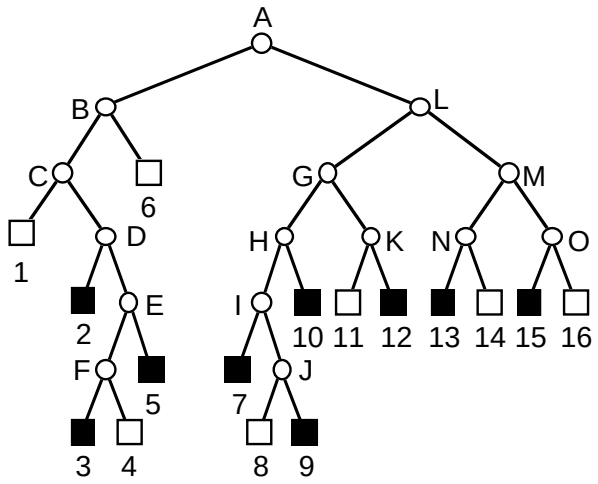
Hanan Samet

Computer Science Department and
Center for Automation Research and
Institute for Advanced Computer Studies
University of Maryland
College Park, Maryland 20742
e-mail: hjs@umiacs.umd.edu

BINTREES (Tamminen, Knowlton)

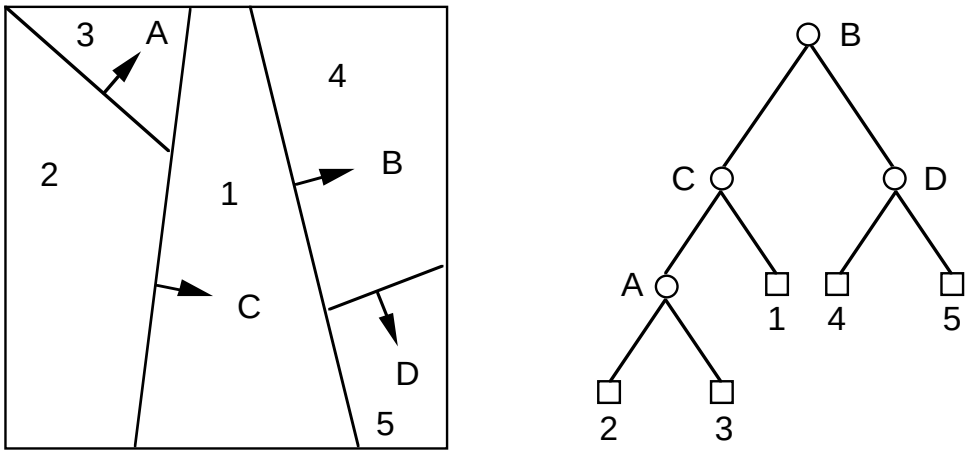
- In higher dimensions (e.g., >3) branching factor of quadtree and octree is too high
- Bintree: regular decomposition binary tree for high dimensional data
  1. at each level split on the basis of another attribute
  2. cycle through the attributes at the different levels
- Ex:





BSP TREES (Fuchs, Kedem, Naylor)

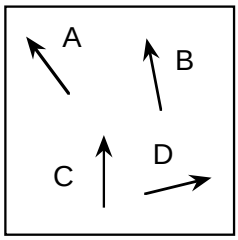
- Like a bintree except that the decomposition lines are at arbitrary orientations (i.e., they need not be parallel or orthogonal)
- For data of arbitrary dimensions
- In 2D (3D), partition along the edges (faces) of a polygon (polyhedron)
- Ex: arrows indicate direction of positive area



- 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

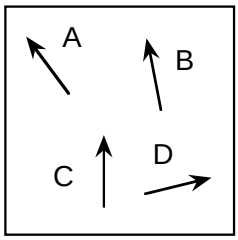
DRAWBACKS OF BSP TREES

- 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

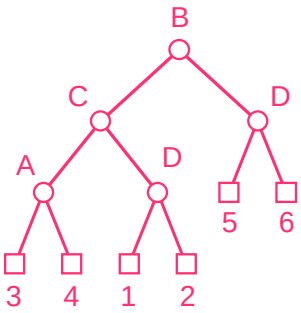


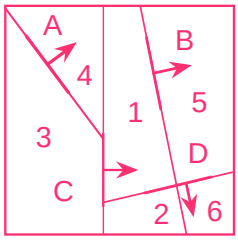
DRAWBACKS OF BSP TREES

- 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



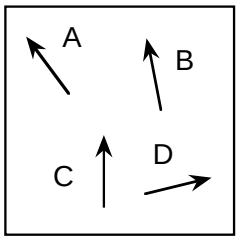
1. partition induced by choosing B as the root



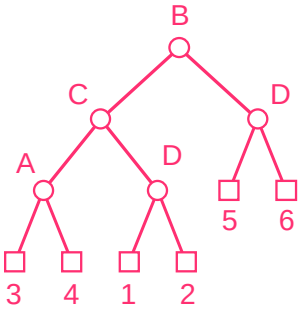


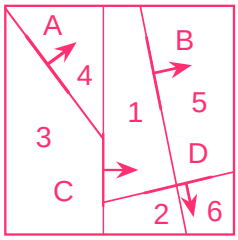
DRAWBACKS OF BSP TREES

- 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

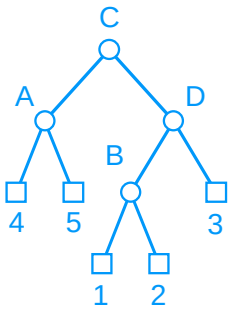


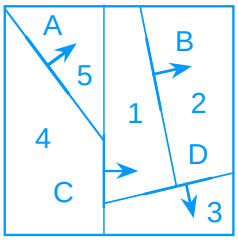
1. partition induced by choosing B as the root





2. partition induced by choosing C as the root

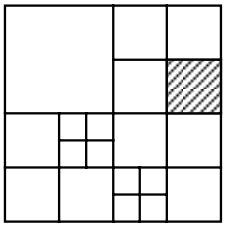




POINTER-LESS QUADTREE REPRESENTATIONS

- Central idea in the quadtree data structure is the recursive decomposition of space into blocks
- The tree is an implementation convenience to enable logarithmic searches for the block associated with a particular point
- Unlike the pyramid, no information is associated with the internal nodes of the quadtree
- Can represent the blocks in a list of numbers where each block has a unique number (termed a location code) formed by concatenating
  1. the sequence of n (assuming a $2^n \times 2^n$ image) two bit codes corresponding to each step in the path from the root of the tree to the block's node
     - let 0, 1, 2, 3 correspond to SW, SE, NW, NE branches, respectively
     - absent steps are encoded with a 0
     - equivalent to interleaving the binary representations of the x and y coordinate values of a particular pixel (e.g., at the lower left corner)
  2. the depth of the block's node
     - necessary to distinguish between paths having trailing digits whose value is 0

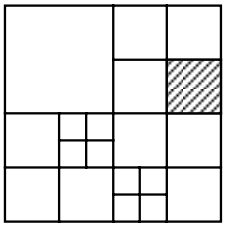
     Ex:



POINTER-LESS QUADTREE REPRESENTATIONS

- Central idea in the quadtree data structure is the recursive decomposition of space into blocks
- The tree is an implementation convenience to enable logarithmic searches for the block associated with a particular point
- Unlike the pyramid, no information is associated with the internal nodes of the quadtree
- Can represent the blocks in a list of numbers where each block has a unique number (termed a location code) formed by concatenating
  1. the sequence of n (assuming a $2^n \times 2^n$ image) two bit codes corresponding to each step in the path from the root of the tree to the block's node
     - let 0, 1, 2, 3 correspond to SW, SE, NW, NE branches, respectively
     - absent steps are encoded with a 0
     - equivalent to interleaving the binary representations of the x and y coordinate values of a particular pixel (e.g., at the lower left corner)
  2. the depth of the block's node
     - necessary to distinguish between paths having trailing digits whose value is 0

     Ex:



path from root = NE, SW

locational code = 310,2

PROPERTIES OF LOCATIONAL CODES

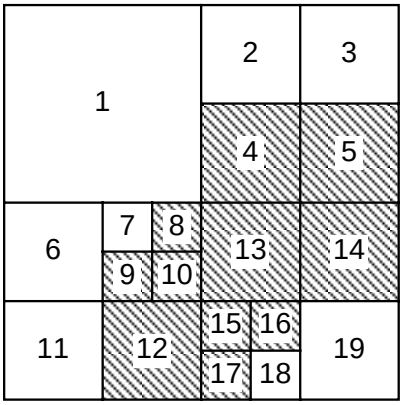
- Locational codes can be stored in a B-tree
- Locational codes are equivalent to a hashing function and are the basis of techniques known as *order preserving linear hashing*
- Sorting locational codes in increasing order has the effect of a space-filling curve and is equivalent to traversing the leaf nodes of the tree in SW, SE, NW, NE order
- Neighbor finding is easy at pixel level but cumbersome at other levels although feasible
- Many alternative locational code implementations exist
  1. variable length locational codes where the depth is omitted and a don't care code (e.g., 4) is used as a sentinel
  2. fixed length locational codes with a don't care symbol to indicate that no further decomposition takes place

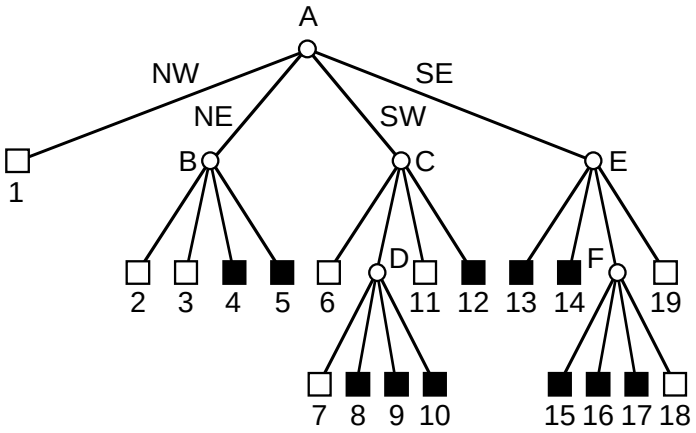
TRAVERSAL-BASED QUADTREE REPRESENTATIONS

- Preorder traversal of the nodes in the quadtree
- Result is a string over the alphabet (DF-expression):

  G = GRAY node B = BLACK node W = WHITE node

- Ex: NW, NE, SW, SE traversal order





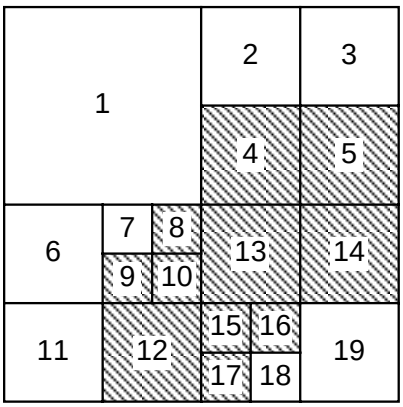
- Drawback: random access is impossible (e.g., for neighbor finding — must always start at the first element in the list and visit all elements prior to the one being searched for
- Useful whenever have to process entire set of nodes in preorder (e.g., NW, NE, SW, SE)
  1. centroid computation
  2. set-theoretic operations
  3. image transformations involving translation, rotation, scaling

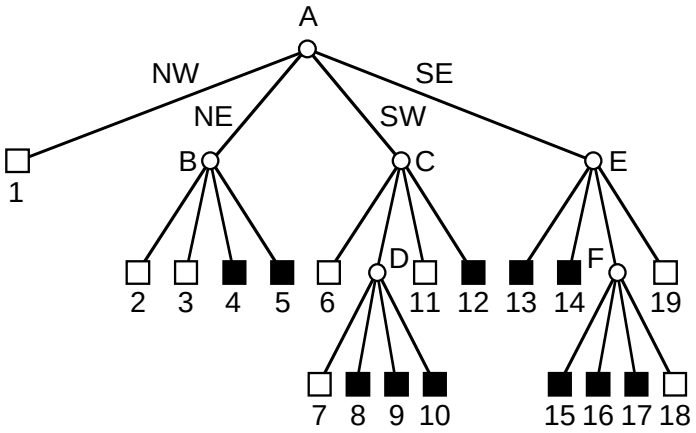
TRAVERSAL-BASED QUADTREE REPRESENTATIONS

- Preorder traversal of the nodes in the quadtree
- Result is a string over the alphabet (DF-expression):

  G = GRAY node B = BLACK node W = WHITE node

- Ex: NW, NE, SW, SE traversal order





G WG WWBB G WG WBBB WB G BBG BBBW W

- Drawback: random access is impossible (e.g., for neighbor finding — must always start at the first element in the list and visit all elements prior to the one being searched for
- Useful whenever have to process entire set of nodes in preorder (e.g., NW, NE, SW, SE)
  1. centroid computation
  2. set-theoretic operations
  3. image transformations involving translation, rotation, scaling

tl0

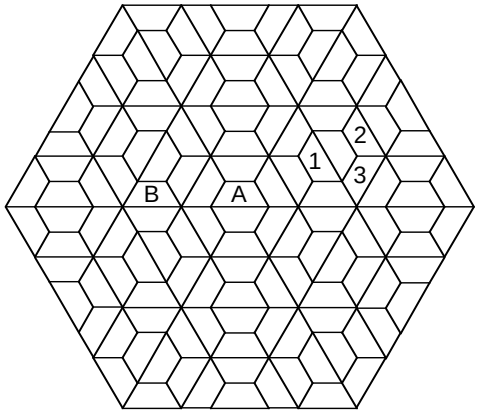
TESSELLATIONS

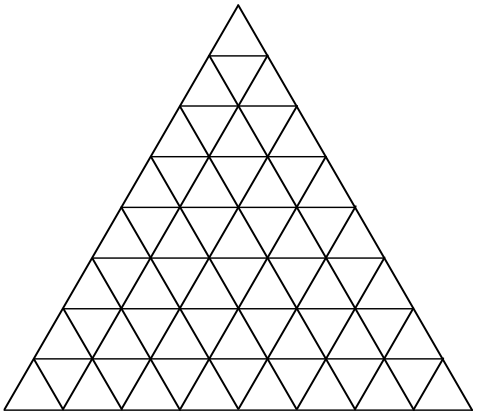
Hanan Samet

Computer Science Department and
Center for Automation Research and
Institute for Advanced Computer Studies
University of Maryland
College Park, Maryland 20742
e-mail: hjs@umiacs.umd.edu

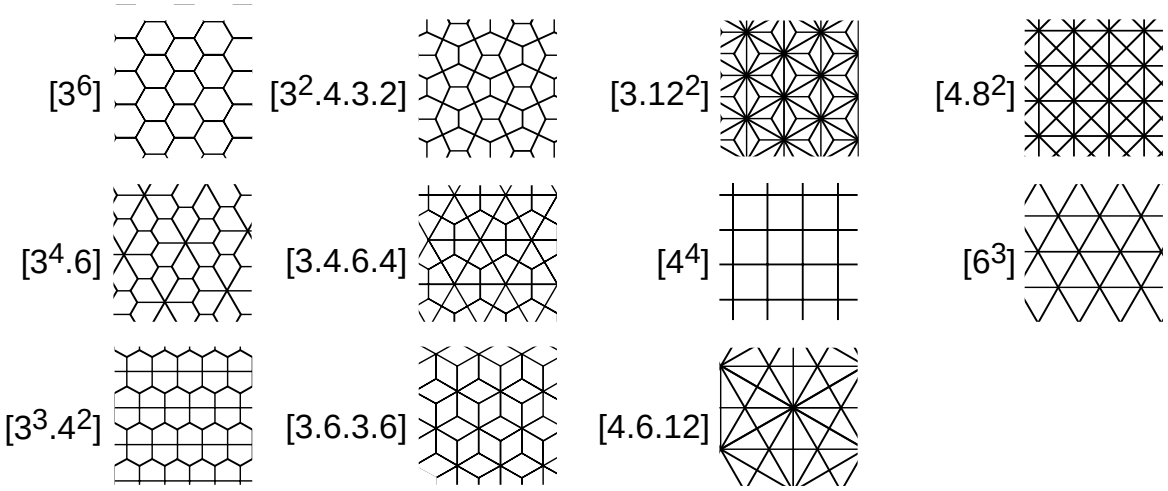
ALTERNATIVE DECOMPOSITION METHODS

- 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)





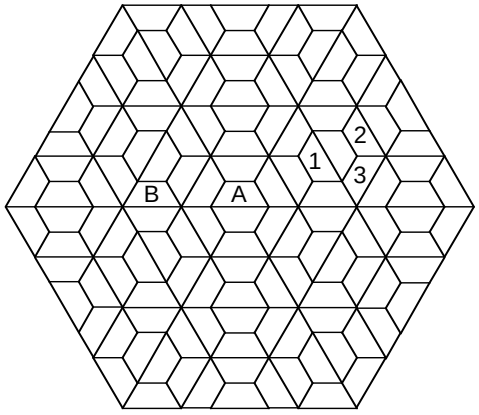
  2. regular — each tile is a regular polygon
- There are 81 types if classify by their symmetry groups
- Only 11 types if classify by their adjacency structure

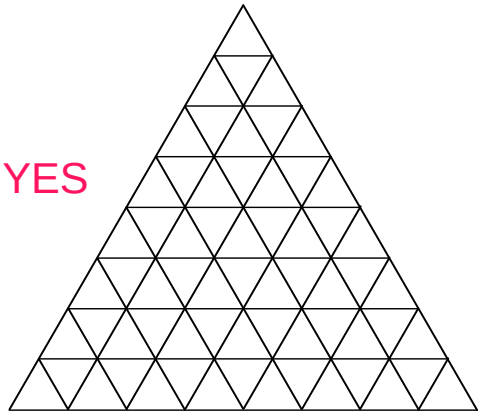


- [3.12^2] means 3 edges at the first vertex of the polygonal tile followed by 12 edges at the next two vertices

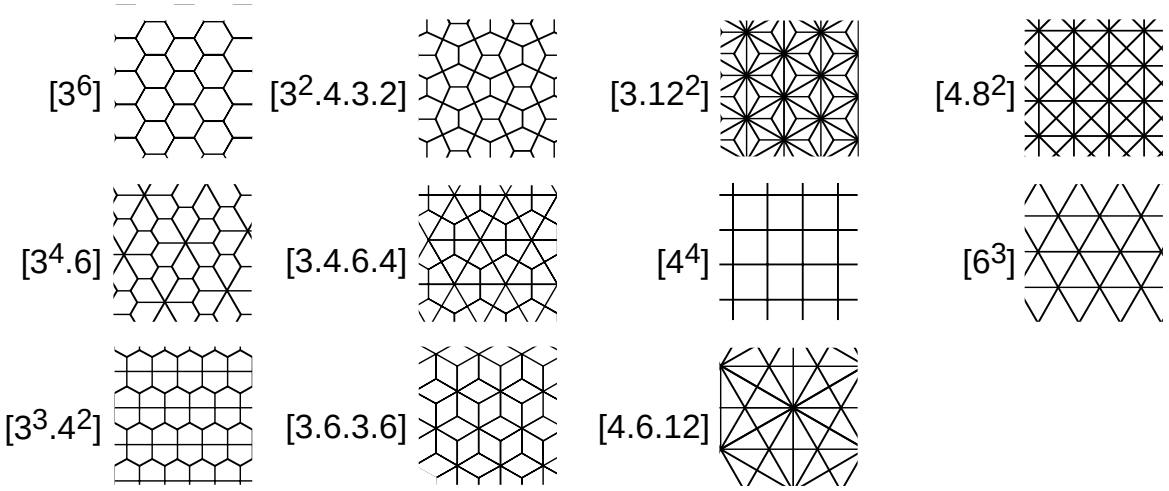
ALTERNATIVE DECOMPOSITION METHODS

- 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)

NO 

YES 

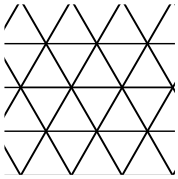
  2. regular — each tile is a regular polygon
- There are 81 types if classify by their symmetry groups
- Only 11 types if classify by their adjacency structure

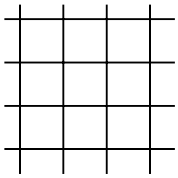


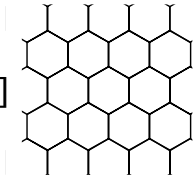
- [3.12^2] means 3 edges at the first vertex of the polygonal tile followed by 12 edges at the next two vertices

PROPERTIES OF TILINGS — SIMILARITY

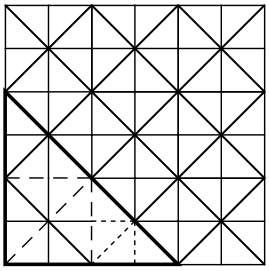
- Similarity — a tile at level *k* has the same shape as a tile at level 0 (basic tile shape)

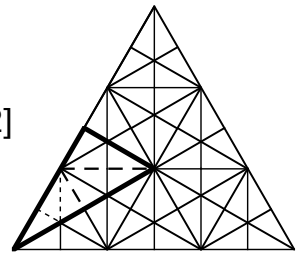
[6^3] 

[4^4] 

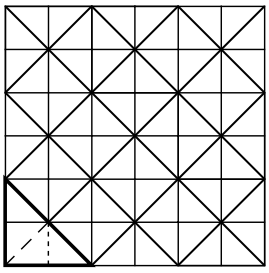
[3^6] 

- 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]

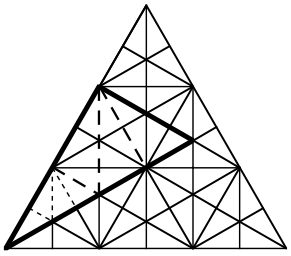
[4.8^2] 

[4.6.12] 

- Two additional hierarchies:



rotation of 135° between levels

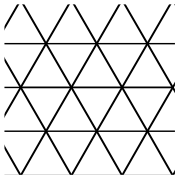


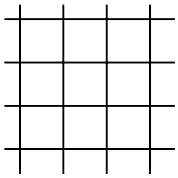
reflection between levels

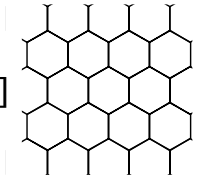
Note: [4.8^2] and [4.6.12] are not regular

PROPERTIES OF TILINGS — SIMILARITY

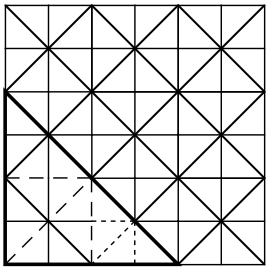
- Similarity — a tile at level *k* has the same shape as a tile at level 0 (basic tile shape)

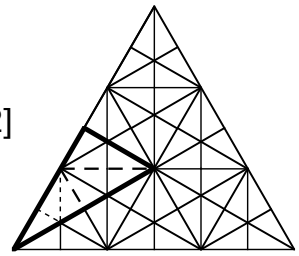
[6^3]  YES

[4^4]  YES

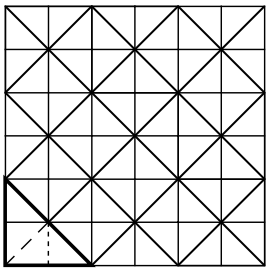
[3^6]  NO

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

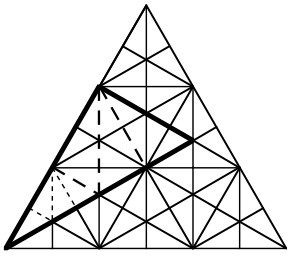
[4.8^2] 

[4.6.12] 

- Two additional hierarchies:



rotation of 135° between levels

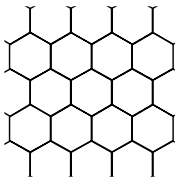


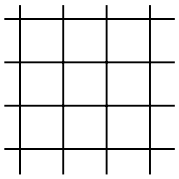
reflection between levels

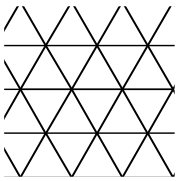
Note: [4.8^2] and [4.6.12] are not regular

PROPERTIES OF TILINGS — ADJACENCY

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

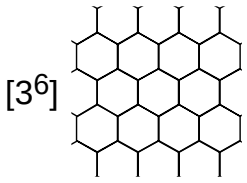
$[3^6]$ 

$[4^4]$ 

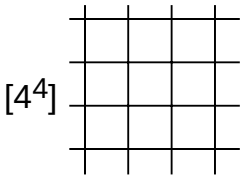
$[6^3]$ 

PROPERTIES OF TILINGS — ADJACENCY

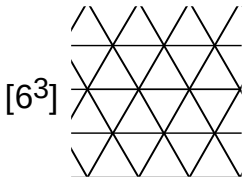
- Adjacency — two tiles are neighbors if they are adjacent along an edge or at a vertex
- Uniform adjacency ≡ distances between the centroid of one tile and the centroids of all its neighbors are the same
- Adjacency number of a tiling (A) ≡ number of different adjacency distances



A=1



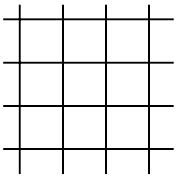
A=2

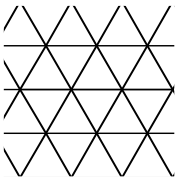


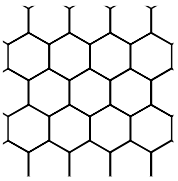
A=3

PROPERTIES OF TILINGS — UNIFORM ORIENTATION

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

$[4^4]$ 

$[6^3]$ 

$[3^6]$ 

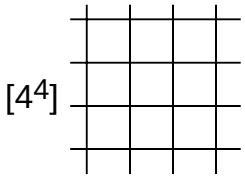
Conclusion:

- $[4^4]$ 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

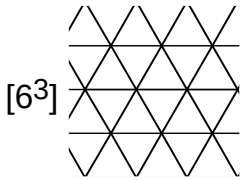
  Use $[4^4]$!

PROPERTIES OF TILINGS — UNIFORM ORIENTATION

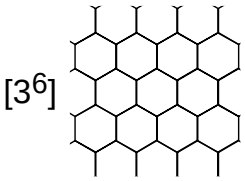
- Uniform orientation
- All tiles with the same orientation can be mapped into each other by translations of the plane which do not involve rotation for reflection



YES



NO



YES

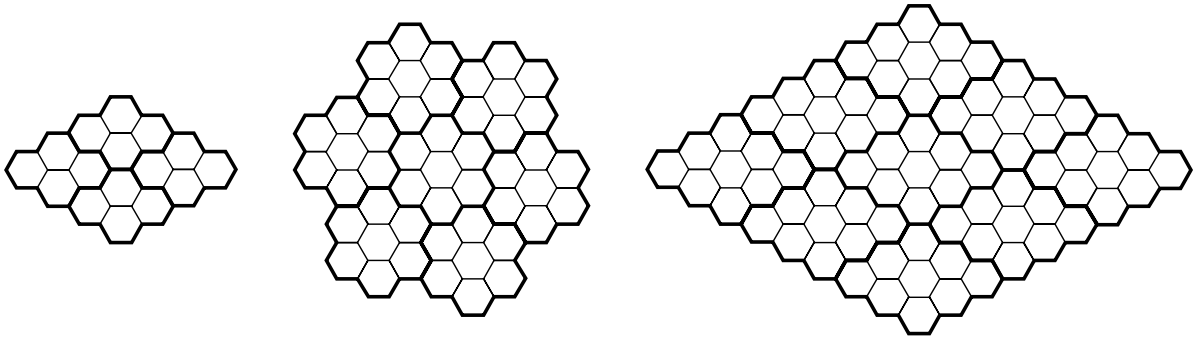
Conclusion:

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

  Use $[4^4]$!

HEXAGONAL TESSELLATIONS [3^6]

1. Still of interest
   - regular
   - uniform orientation
   - uniform adjacency



4-shape 7-shape 9-shape

2. Several tiling hierarchies (*n*-shapes) NOT UNIQUE!
   - $n \equiv$ number of atomic tiles in the first level molecular tile
   - 4-shape and 9-shape have unusual adjacency behavior
     a. contact with two of the neighboring molecular tiles is along only one edge of a molecular tile while contact with the remaining four tiles is nearly along 1/4 of the perimeter
     b. molecular tile has the shape of a rhombus
   - 7-shape
     a. uniform contact with all six neighboring molecular tiles
     b. the shape of the molecular tile is more like a hexagon ($\equiv$ rosette and termed a septree)

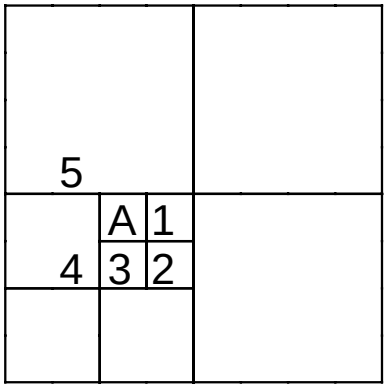
NEIGHBOR FINDING METHODS IN QUADTREES

Hanan Samet

Computer Science Department and
Center for Automation Research and
Institute for Advanced Computer Studies
University of Maryland
College Park, Maryland 20742
e-mail: hjs@umiacs.umd.edu

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

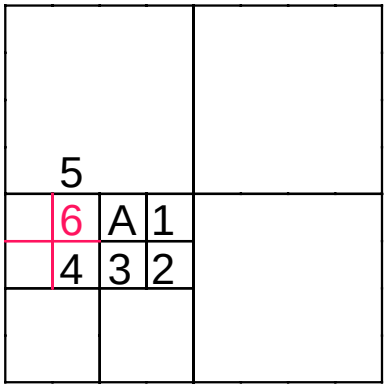


A has 5 neighbors

- 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

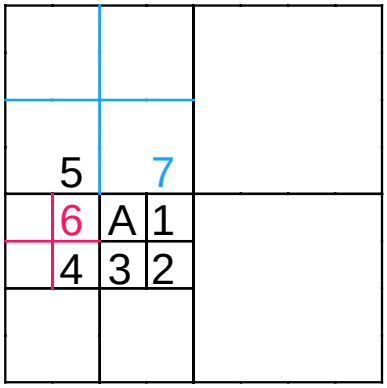


A has ~~5~~ 6 neighbors

- 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

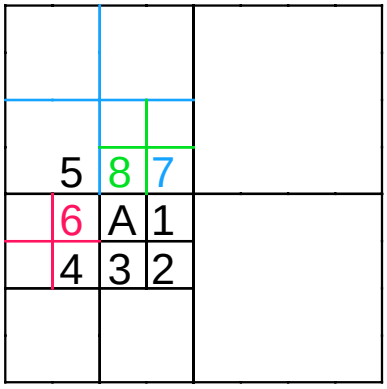


A has ~~5~~ ~~6~~ 7 neighbors

- 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

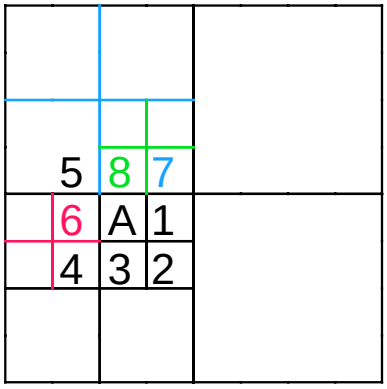


A has ~~5~~ ~~6~~ ~~7~~ 8 neighbors

- 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

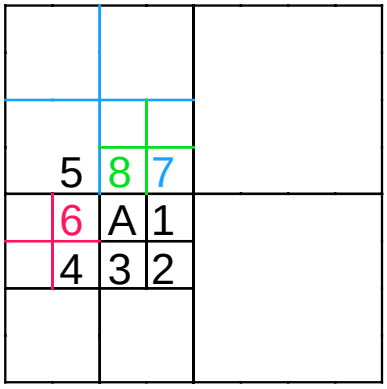
- Desirable to be able to locate neighbors in a manner that
  1. is position-independent
  2. is size-independent
  3. makes no use of additional links to adjacent nodes (e.g., ropes and nets a la Hunter)
  4. just uses the structure of the tree or configuration of the blocks
- Some block configurations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

     A
  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

     A

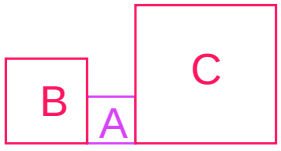
NEIGHBOR FINDING OPERATIONS USING QUADTREES

- 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

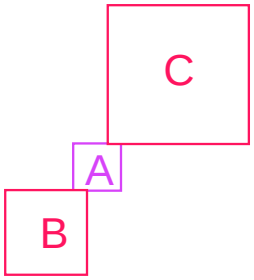


A has ~~5~~ ~~6~~ ~~7~~ 8 neighbors

- 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

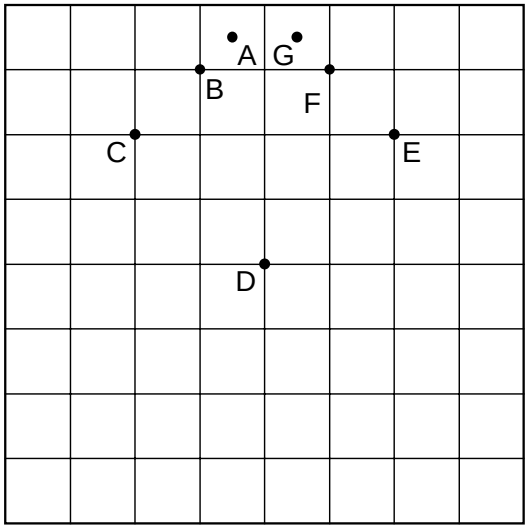


FINDING LATERAL NEIGHBORS OF EQUAL SIZE

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)

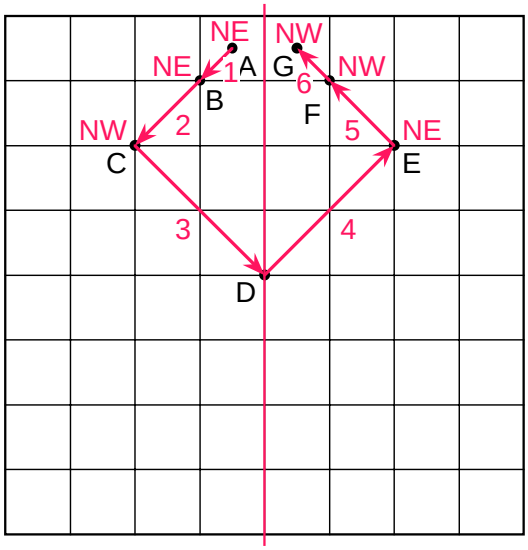


FINDING LATERAL NEIGHBORS OF EQUAL SIZE

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)

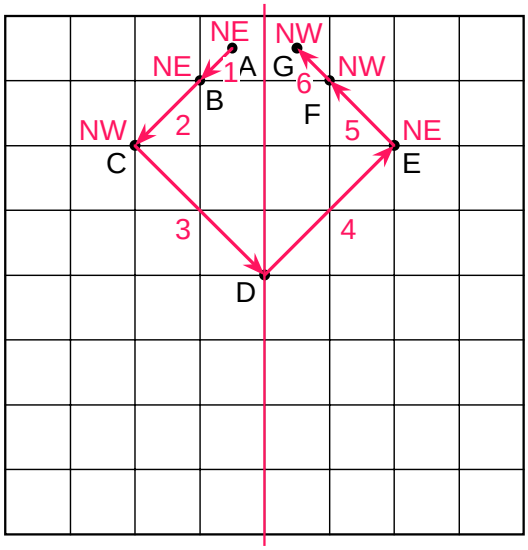


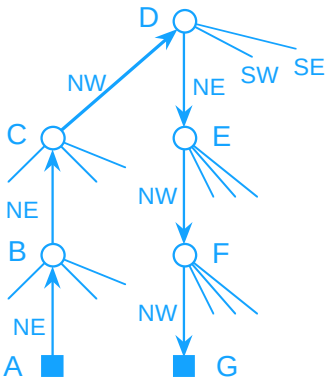
FINDING LATERAL NEIGHBORS OF EQUAL SIZE

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)



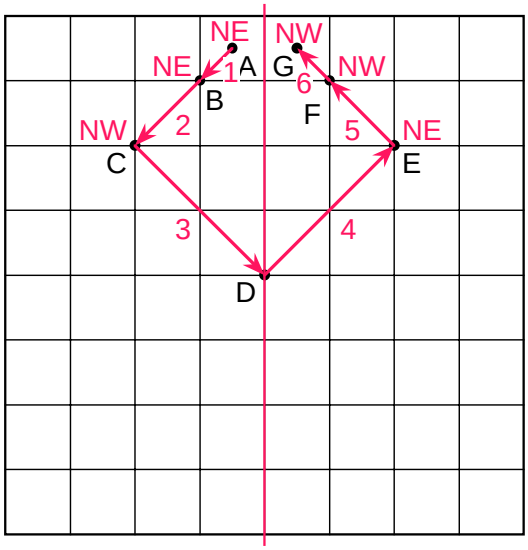


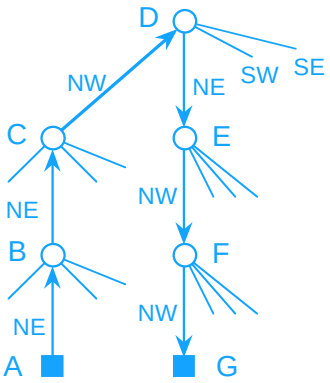
FINDING LATERAL NEIGHBORS OF EQUAL SIZE

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)





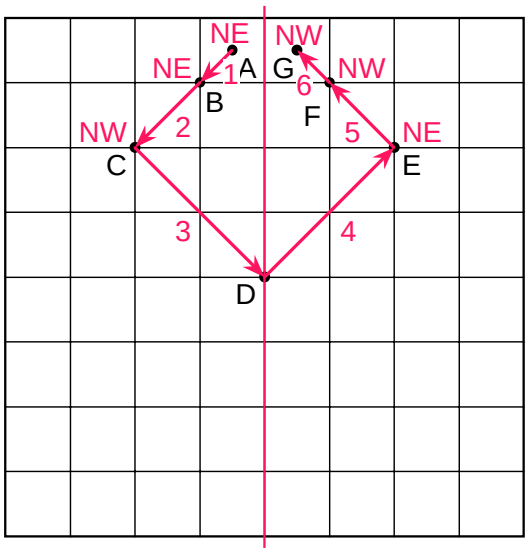

```
node procedure EQUAL_LATERAL_NEIGHBOR(P,D);
/* Find = size neighbor of P in direction D */
begin
  value pointer node P;
  value direction D;
  return(SON(if ADJ(D,SONTYPE(P)) then
               EQUAL_LATERAL_NEIGHBOR(FATHER(P),D)
             else FATHER(P),
             REFLECT(D,SONTYPE(P))));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

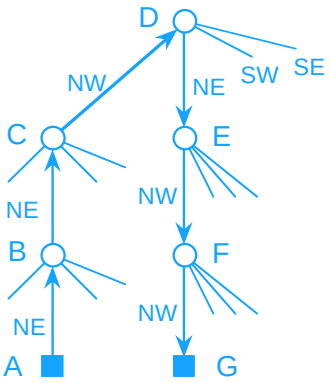
FINDING LATERAL NEIGHBORS OF EQUAL SIZE

Algorithm: based on finding the nearest common ancestor

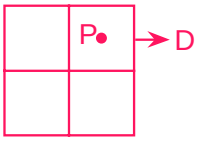
1. Ascend the tree if the node is a son of the same type as the direction of the neighbor (ADJ)
2. Otherwise, the father F is the nearest common ancestor and retrace the path starting at F making mirror image moves about the edge shared by the neighboring blocks

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)

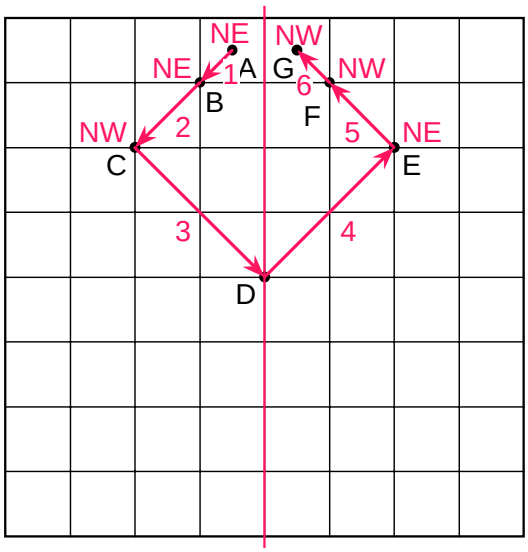
A \ B	NW	NE	SW	SE
N	T	T	F	F
E	F	T	F	T
S	F	F	T	T
W	T	F	T	F

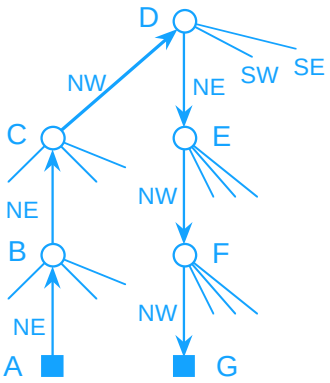
FINDING LATERAL NEIGHBORS OF EQUAL SIZE

Algorithm: based on finding the nearest common ancestor

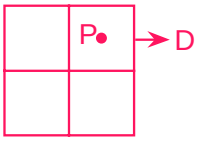
1. Ascend the tree if the node is a son of the same type as the direction of the neighbor (ADJ)
2. Otherwise, the father F is the nearest common ancestor and retrace the path starting at F making mirror image moves about the edge shared by the neighboring blocks

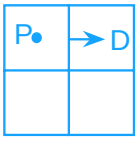
Ex: E neighbor of A (i.e., G)






```
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;
```





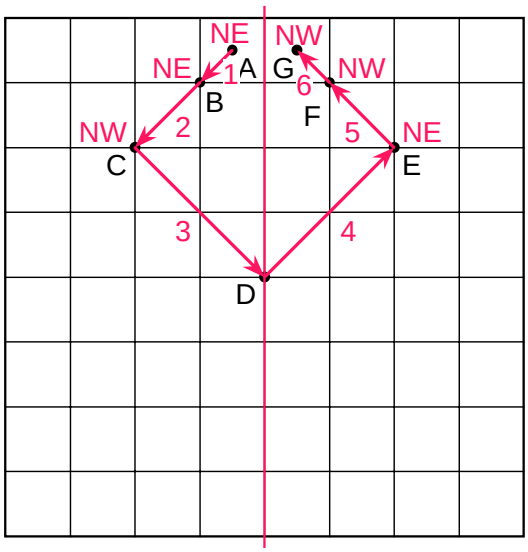
ADJ(A,B)	NW	NE	SW	SE
N	T	T	F	F
E	F	T	F	T
S	F	F	T	T
W	T	F	T	F

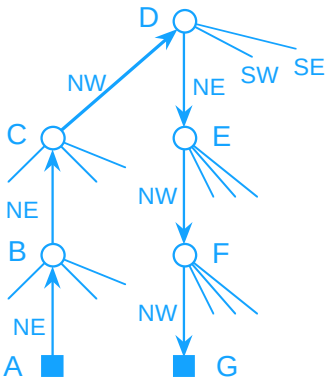
FINDING LATERAL NEIGHBORS OF EQUAL SIZE

Algorithm: based on finding the nearest common ancestor

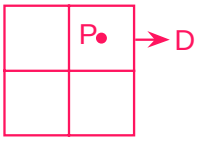
1. Ascend the tree if the node is a son of the same type as the direction of the neighbor (ADJ)
2. Otherwise, the father F is the nearest common ancestor and retrace the path starting at F making mirror image moves about the edge shared by the neighboring blocks

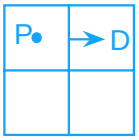
Ex: E neighbor of A (i.e., G)






```
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;
```





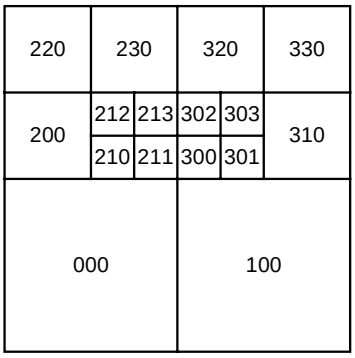
ADJ(A,B)

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

FINDING NEIGHBORS IN A POINTERLESS REPRESENTATION



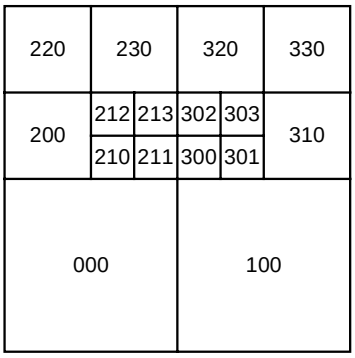
- Assume a bit-interleaved representation that also indicates the depth of each block
- Ex: *y* bit is more significant than *x* bit — i.e., $y_n\ x_n \ldots y_1\ x_1\ y_0\ x_0$

1. Mimic tree algorithm in the sense that to find a lateral neighbor (say in the *x* direction), search for the first *x* bit position that is different while complementing the bits encountered in the process

   Ex: find pixel-sized eastern neighbor of `213 = 10 01 11`

FINDING NEIGHBORS IN A POINTERLESS REPRESENTATION

- Assume a bit-interleaved representation that also indicates the depth of each block
- Ex: *y* bit is more significant than *x* bit — i.e., $y_n\ x_n \ldots y_1\ x_1\ y_0\ x_0$



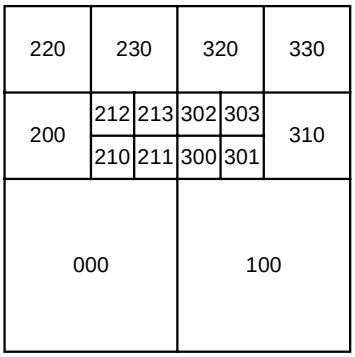
1. Mimic tree algorithm in the sense that to find a lateral neighbor (say in the *x* direction), search for the first *x* bit position that is different while complementing the bits encountered in the process

   Ex: find pixel-sized eastern neighbor of `213 = 10 01 11`

   - search for an *x* bit with a value of 0

FINDING NEIGHBORS IN A POINTERLESS REPRESENTATION

- Assume a bit-interleaved representation that also indicates the depth of each block
- Ex: *y* bit is more significant than *x* bit — i.e., $y_n\ x_n \ldots y_1\ x_1\ y_0\ x_0$



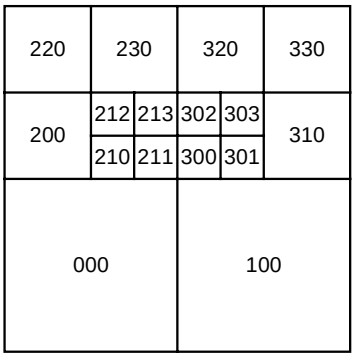
1. Mimic tree algorithm in the sense that to find a lateral neighbor (say in the *x* direction), search for the first *x* bit position that is different while complementing the bits encountered in the process

   Ex: find pixel-sized eastern neighbor of `213 = 10 01 11`

   - search for an *x* bit with a value of 0
   - result is `302 = 11 00 10`

FINDING NEIGHBORS IN A POINTERLESS REPRESENTATION

- Assume a bit-interleaved representation that also indicates the depth of each block
- Ex: *y* bit is more significant than *x* bit — i.e., $y_n\ x_n \ldots y_1\ x_1\ y_0\ x_0$



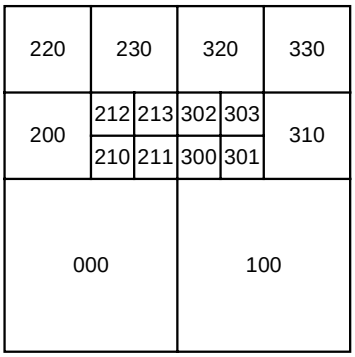
1. Mimic tree algorithm in the sense that to find a lateral neighbor (say in the *x* direction), search for the first *x* bit position that is different while complementing the bits encountered in the process

   Ex: find pixel-sized eastern neighbor of `213 = 10 01 11`

   - search for an *x* bit with a value of 0
   - result is `302 = 11 00 10`
   - cumbersome as many bit operations are required

FINDING NEIGHBORS IN A POINTERLESS REPRESENTATION

- Assume a bit-interleaved representation that also indicates the depth of each block
- Ex: *y* bit is more significant than *x* bit — i.e., $y_n\ x_n \ldots y_1\ x_1\ y_0\ x_0$



1. Mimic tree algorithm in the sense that to find a lateral neighbor (say in the *x* direction), search for the first *x* bit position that is different while complementing the bits encountered in the process

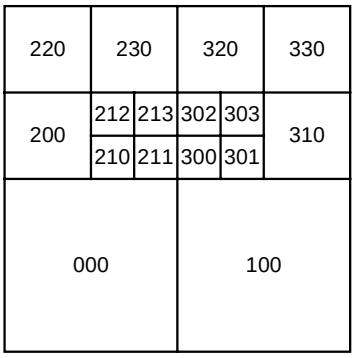
   Ex: find pixel-sized eastern neighbor of `213 = 10 01 11`

   - search for an *x* bit with a value of 0
   - result is `302 = 11 00 10`
   - cumbersome as many bit operations are required

2. Use arithmetic by adding or subtracting from the appropriate *x* (*y*) bit and skip the positions corresponding to the *y* (or *x*) bits
   - save the contents of the *y* (*x*) bit positions
   - load a 1 in every *y* (*x*) bit position of the addend — enables propagation of the carry (if one is present) to the next *x* (*y*) bit position

Ex: find pixel-sized eastern neighbor of `213 = 10 01 11`
the addend 00 00 01 becomes `+ 10 10 11`

FINDING NEIGHBORS IN A POINTERLESS REPRESENTATION



- Assume a bit-interleaved representation that also indicates the depth of each block
- Ex: *y* bit is more significant than *x* bit — i.e., $y_n\ x_n \ldots y_1\ x_1\ y_0\ x_0$

1. Mimic tree algorithm in the sense that to find a lateral neighbor (say in the *x* direction), search for the first *x* bit position that is different while complementing the bits encountered in the process

   Ex: find pixel-sized eastern neighbor of `213 = 10 01 11`

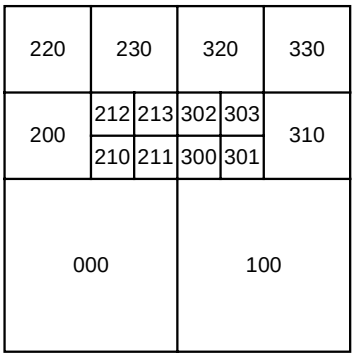
   - search for an *x* bit with a value of 0
   - result is `302 = 11 00 10`
   - cumbersome as many bit operations are required

2. Use arithmetic by adding or subtracting from the appropriate *x* (*y*) bit and skip the positions corresponding to the *y* (or *x*) bits
   - save the contents of the *y* (*x*) bit positions
   - load a 1 in every *y* (*x*) bit position of the addend — enables propagation of the carry (if one is present) to the next *x* (*y*) bit position
   - perform the addition or subtraction

Ex: find pixel-sized eastern neighbor of `213 = 10 01 11`
the addend 00 00 01 becomes `+ 10 10 11`
result `01 00 10`

FINDING NEIGHBORS IN A POINTERLESS REPRESENTATION

- Assume a bit-interleaved representation that also indicates the depth of each block
- Ex: *y* bit is more significant than *x* bit — i.e., $y_n\ x_n \ldots y_1\ x_1\ y_0\ x_0$



1. Mimic tree algorithm in the sense that to find a lateral neighbor (say in the *x* direction), search for the first *x* bit position that is different while complementing the bits encountered in the process

   Ex: find pixel-sized eastern neighbor of `213 = 10 01 11`

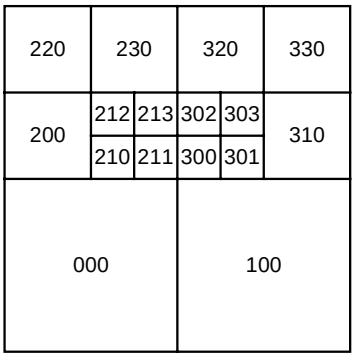
   - search for an *x* bit with a value of 0
   - result is `302 = 11 00 10`
   - cumbersome as many bit operations are required

2. Use arithmetic by adding or subtracting from the appropriate *x* (*y*) bit and skip the positions corresponding to the *y* (or *x*) bits
   - save the contents of the *y* (*x*) bit positions
   - load a 1 in every *y* (*x*) bit position of the addend — enables propagation of the carry (if one is present) to the next *x* (*y*) bit position
   - perform the addition or subtraction
   - reset the *y* (*x*) bit positions to their previous value

Ex: find pixel-sized eastern neighbor of `213 = 10 01 11`
the addend 00 00 01 becomes `+ 10 10 11`
result `01 00 10`
restoring *y* bit positions to previous `11 00 10 = 302`

FINDING NEIGHBORS IN A POINTERLESS REPRESENTATION

- Assume a bit-interleaved representation that also indicates the depth of each block
- Ex: *y* bit is more significant than *x* bit — i.e., $y_n\ x_n \ldots y_1\ x_1\ y_0\ x_0$



1. Mimic tree algorithm in the sense that to find a lateral neighbor (say in the *x* direction), search for the first *x* bit position that is different while complementing the bits encountered in the process

   Ex: find pixel-sized eastern neighbor of `213 = 10 01 11`
   - search for an *x* bit with a value of 0
   - result is `302 = 11 00 10`
   - cumbersome as many bit operations are required

2. Use arithmetic by adding or subtracting from the appropriate *x* (*y*) bit and skip the positions corresponding to the *y* (or *x*) bits
   - save the contents of the *y* (*x*) bit positions
   - load a 1 in every *y* (*x*) bit position of the addend — enables propagation of the carry (if one is present) to the next *x* (*y*) bit position
   - perform the addition or subtraction
   - reset the *y* (*x*) bit positions to their previous value

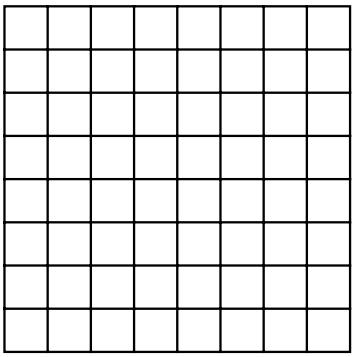
Ex: find pixel-sized eastern neighbor of `213 = 10 01 11`
the addend 00 00 01 becomes `+ 10 10 11`
result `01 00 10`
restoring *y* bit positions to previous `11 00 10 = 302`

- only requires 4 instructions regardless of resolution of image and bit position of nearest common ancestor

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

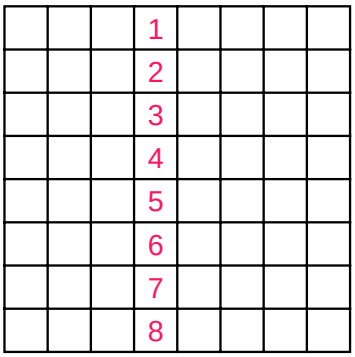
ANALYSIS OF FINDING LATERAL NEIGHBORS



$2^3 \cdot (2^3 - 1)$ 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

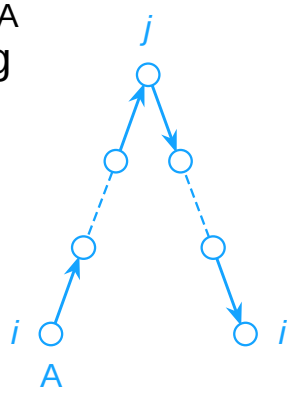
  $2^{n-i} \cdot 2^0$ have NCA at level n

  $2^{n-i} \cdot 2^1$ 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

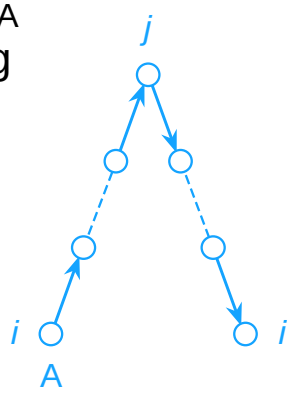
  $2^{n-i} \cdot 2^0$ have NCA at level n

  $2^{n-i} \cdot 2^1$ 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



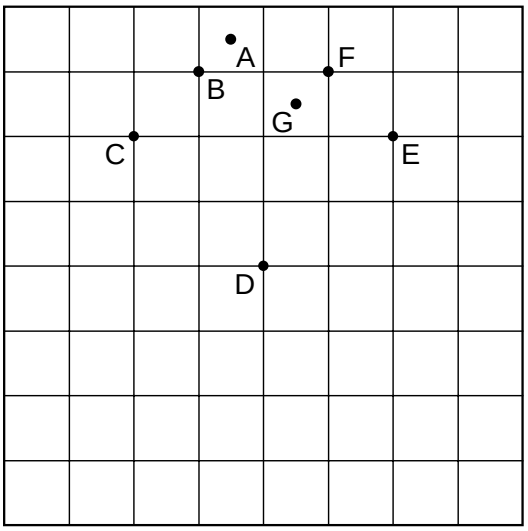
$$\frac{\sum_{i=0}^{n-1} \sum_{j=i+1}^{n} 2^{n-i} \cdot 2^{n-j} \cdot 2 \cdot (j-i)}{\sum_{i=0}^{n-1} 2^{n-i} \cdot (2^{n-i} - 1)}$$

nodes are visited on the average ≤ 4

FINDING DIAGONAL 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. If the father and an ancestor A of the desired neighbor are adjacent along an edge (COMMON_EDGE), then calculate the desired neighbor with EQUAL_LATERAL_NEIGHBOR and apply the retracing step in 3
3. Otherwise, the father F is the nearest common ancestor and now retrace the path starting at F making diagonally opposite moves

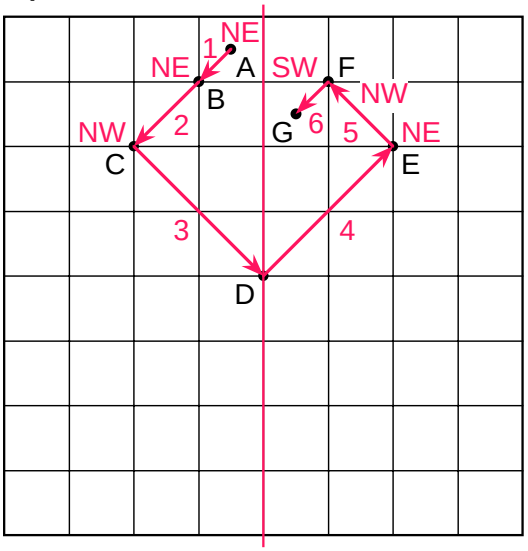


Ex: SE neighbor of A (i.e., G)

FINDING DIAGONAL 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. If the father and an ancestor A of the desired neighbor are adjacent along an edge (COMMON_EDGE), then calculate the desired neighbor with EQUAL_LATERAL_NEIGHBOR and apply the retracing step in 3
3. Otherwise, the father F is the nearest common ancestor and now retrace the path starting at F making diagonally opposite moves



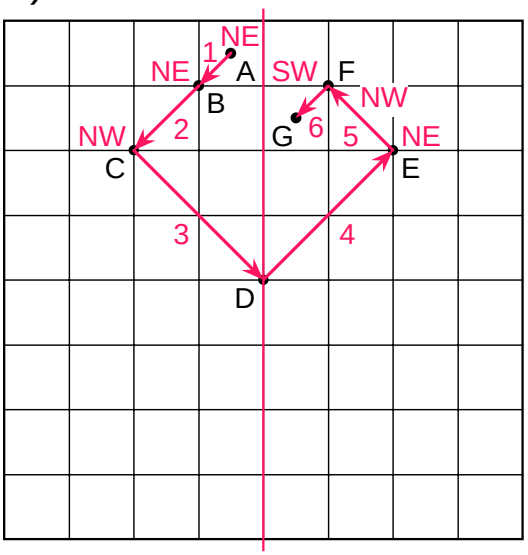
Ex: SE neighbor of A (i.e., G)

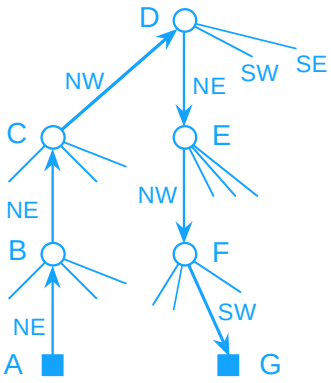
FINDING DIAGONAL NEIGHBORS OF EQUAL SIZE

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. If the father and an ancestor A of the desired neighbor are adjacent along an edge (COMMON_EDGE), then calculate the desired neighbor with EQUAL_LATERAL_NEIGHBOR and apply the retracing step in 3
3. Otherwise, the father F is the nearest common ancestor and now retrace the path starting at F making diagonally opposite moves

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

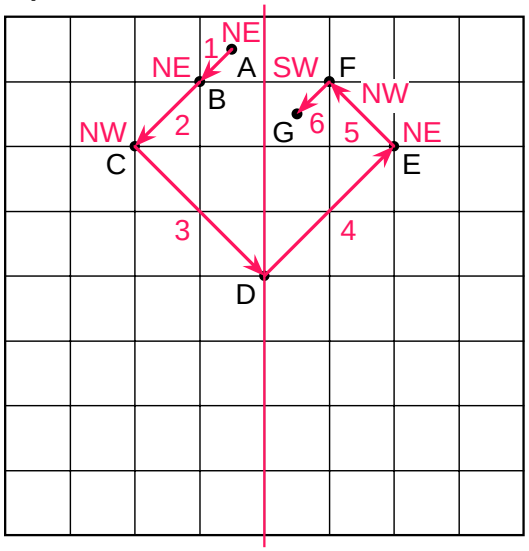




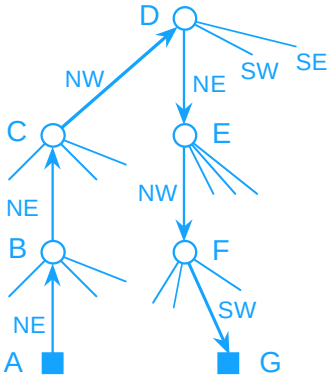
FINDING DIAGONAL NEIGHBORS OF EQUAL SIZE

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. If the father and an ancestor A of the desired neighbor are adjacent along an edge (COMMON_EDGE), then calculate the desired neighbor with EQUAL_LATERAL_NEIGHBOR and apply the retracing step in 3
3. Otherwise, the father F is the nearest common ancestor and now retrace the path starting at F making diagonally opposite moves



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

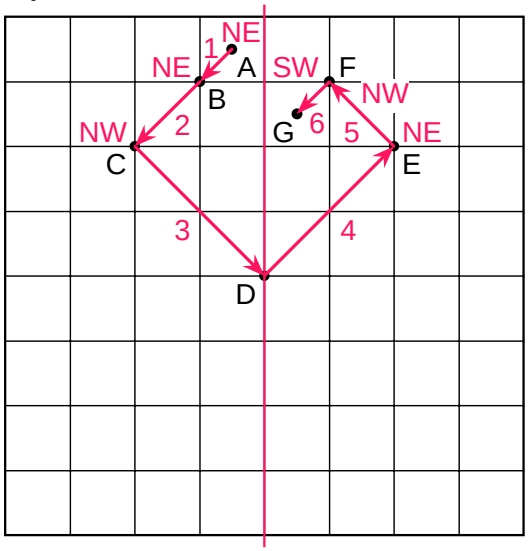



```
node procedure EQUAL_DIAGONAL_NEIGHBOR(P,C);
/* Find = size neighbor of P towards quadrant C */
begin
  value pointer node P;
  value quadrant C;
  return(SON(if ADJ(C,SONTYPE(P)) then
               EQUAL_DIAGONAL_NEIGHBOR(FATHER(P),C)
             else if COMMON_EDGE(C,SONTYPE(P))≠Ω then
               EQUAL_LATERAL_NEIGHBOR(FATHER(P),
               COMMON_EDGE(C,SONTYPE(P)))
             else FATHER(P),
             OPQUAD(SONTYPE(P))));
end;
```

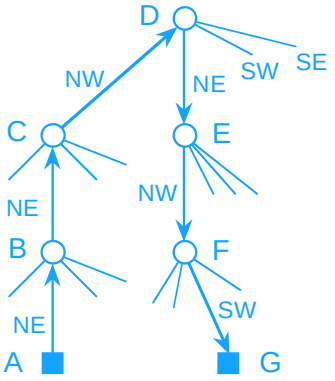
FINDING DIAGONAL NEIGHBORS OF EQUAL SIZE

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. If the father and an ancestor A of the desired neighbor are adjacent along an edge (COMMON_EDGE), then calculate the desired neighbor with EQUAL_LATERAL_NEIGHBOR and apply the retracing step in 3
3. Otherwise, the father F is the nearest common ancestor and now retrace the path starting at F making diagonally opposite moves

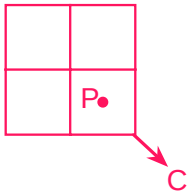


Ex: SE neighbor of A (i.e., G)



A \ B	NW	NE	SW	SE
NW	T	F	F	F
NE	F	T	F	F
SW	F	F	T	F
SE	F	F	F	T

ADJ(A,B)

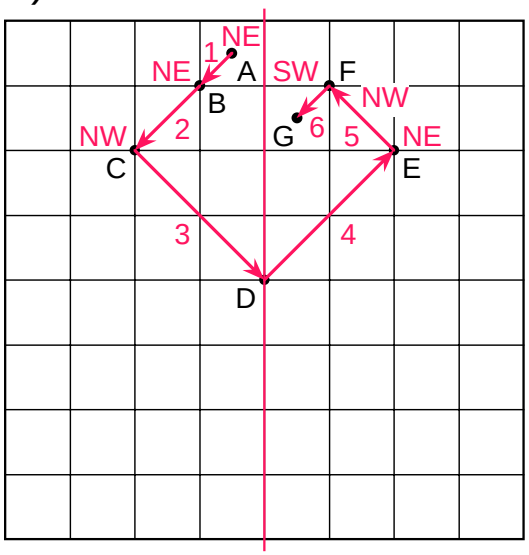



```
node procedure EQUAL_DIAGONAL_NEIGHBOR(P,C);
/* Find = size neighbor of P towards quadrant C */
begin
  value pointer node P;
  value quadrant C;
  return(SON(if ADJ(C,SONTYPE(P)) then
                EQUAL_DIAGONAL_NEIGHBOR(FATHER(P),C)
              else if COMMON_EDGE(C,SONTYPE(P))≠Ω then
                EQUAL_LATERAL_NEIGHBOR(FATHER(P),
                COMMON_EDGE(C,SONTYPE(P)))
              else FATHER(P),
              OPQUAD(SONTYPE(P))));
end;
```

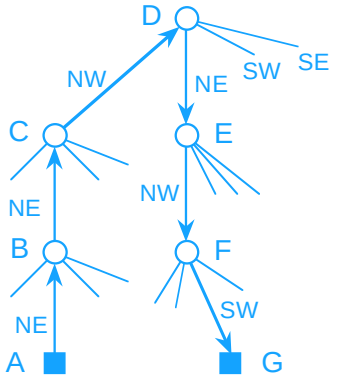
FINDING DIAGONAL NEIGHBORS OF EQUAL SIZE

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. If the father and an ancestor A of the desired neighbor are adjacent along an edge (COMMON_EDGE), then calculate the desired neighbor with EQUAL_LATERAL_NEIGHBOR and apply the retracing step in 3
3. Otherwise, the father F is the nearest common ancestor and now retrace the path starting at F making diagonally opposite moves



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



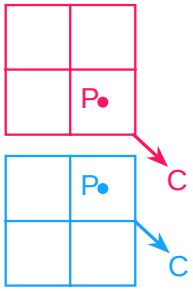
A \ B	NW	NE	SW	SE
NW	T	F	F	F
NE	F	T	F	F
SW	F	F	T	F
SE	F	F	F	T

ADJ(A,B)

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

COMMON_EDGE(A,B)

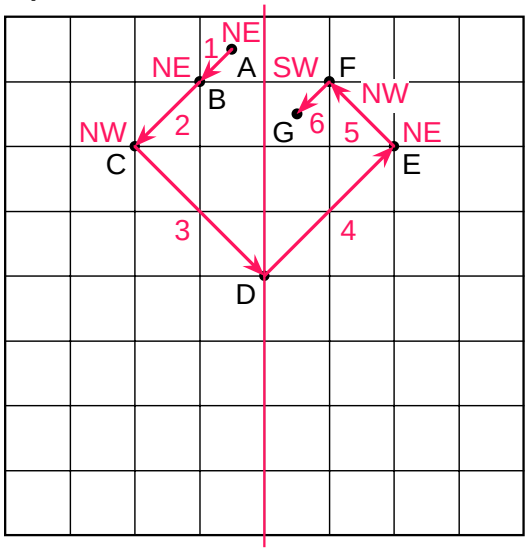
```
node procedure EQUAL_DIAGONAL_NEIGHBOR(P,C);
/* Find = size neighbor of P towards quadrant C */
begin
  value pointer node P;
  value quadrant C;
  return(SON(if ADJ(C,SONTYPE(P)) then
                EQUAL_DIAGONAL_NEIGHBOR(FATHER(P),C)
              else if COMMON_EDGE(C,SONTYPE(P))≠Ω then
                EQUAL_LATERAL_NEIGHBOR(FATHER(P),
                COMMON_EDGE(C,SONTYPE(P)))
              else FATHER(P),
              OPQUAD(SONTYPE(P))));
end;
```



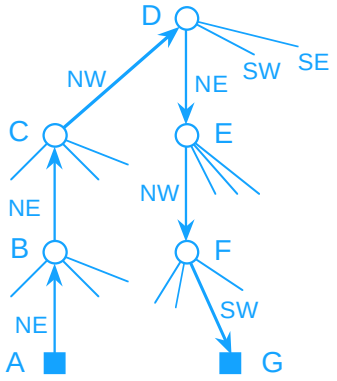
FINDING DIAGONAL NEIGHBORS OF EQUAL SIZE

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. If the father and an ancestor A of the desired neighbor are adjacent along an edge (COMMON_EDGE), then calculate the desired neighbor with EQUAL_LATERAL_NEIGHBOR and apply the retracing step in 3
3. Otherwise, the father F is the nearest common ancestor and now retrace the path starting at F making diagonally opposite moves



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



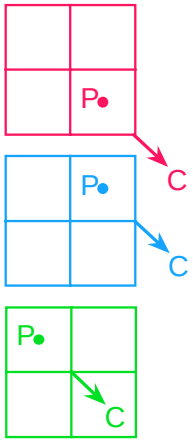
ADJ(A,B)

A \ B	NW	NE	SW	SE
NW	T	F	F	F
NE	F	T	F	F
SW	F	F	T	F
SE	F	F	F	T

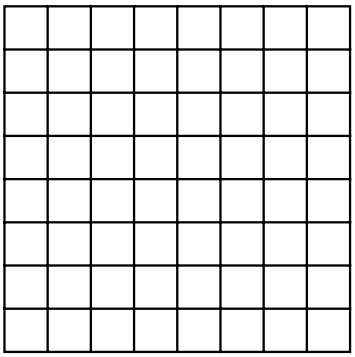
COMMON_EDGE(A,B)

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

```
node procedure EQUAL_DIAGONAL_NEIGHBOR(P,C);
/* Find = size neighbor of P towards quadrant C */
begin
  value pointer node P;
  value quadrant C;
  return(SON(if ADJ(C,SONTYPE(P)) then
                EQUAL_DIAGONAL_NEIGHBOR(FATHER(P),C)
              else if COMMON_EDGE(C,SONTYPE(P))≠Ω then
                EQUAL_LATERAL_NEIGHBOR(FATHER(P),
                COMMON_EDGE(C,SONTYPE(P)))
              else FATHER(P),
              OPQUAD(SONTYPE(P))));
end;
```



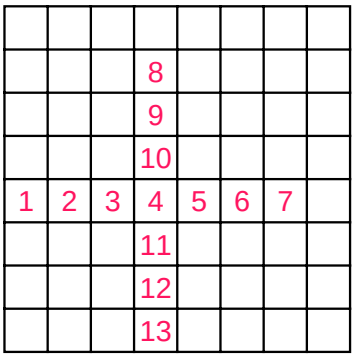
ANALYSIS OF FINDING DIAGONAL NEIGHBORS



$(2^3–1)^2$ neighbor pairs of equal sized nodes in direction NE

NCA = nearest common ancestor

ANALYSIS OF FINDING DIAGONAL NEIGHBORS



$(2^3–1)^2$ neighbor pairs of equal sized nodes in direction NE

NCA = nearest common ancestor

1–13 have NCA at level 3

ANALYSIS OF FINDING DIAGONAL NEIGHBORS

	17		8		22		
14	15	16	9	19	20	21	
	18		10		23		
1	2	3	4	5	6	7	
	27		11		32		
24	25	26	12	29	30	31	
	28		13		33		

$(2^3–1)^2$ neighbor pairs of equal sized nodes in direction NE

NCA = nearest common ancestor

1–13 have NCA at level 3

14–33 have NCA at level 2

ANALYSIS OF FINDING DIAGONAL NEIGHBORS

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

$(2^3–1)^2$ neighbor pairs of equal sized nodes in direction NE

NCA = nearest common ancestor

1–13 have NCA at level 3

14–33 have NCA at level 2

34–49 have NCA at level 1

ANALYSIS OF FINDING DIAGONAL NEIGHBORS

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

$(2^3–1)^2$ neighbor pairs of equal sized nodes in direction NE

NCA = nearest common ancestor

1–13 have NCA at level 3

14–33 have NCA at level 2

34–49 have NCA at level 1

Theorem: average number of nodes visited by `EQUAL_DIAGONAL_NEIGHBOR` is ≤ 16/3

Proof:

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

  $4^0 \cdot (2 \cdot (2^{n-i}–1)–1)$ have NCA at level n

  $4^1 \cdot (2 \cdot (2^{n-i-1}–1)–1)$ have NCA at level $n–1$

  …

  $4^{n-i-1} \cdot (2 \cdot (2^{n-i-(n-i-1)}–1)–1)$ have NCA at level $i+1$

ANALYSIS OF FINDING DIAGONAL NEIGHBORS

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

$(2^3–1)^2$ neighbor pairs of equal sized nodes in direction NE

NCA = nearest common ancestor

1–13 have NCA at level 3

14–33 have NCA at level 2

34–49 have NCA at level 1

Theorem: average number of nodes visited by `EQUAL_DIAGONAL_NEIGHBOR` is ≤ 16/3

Proof:

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

  $4^0 \cdot (2 \cdot (2^{n-i} – 1) – 1)$ have NCA at level n

  $4^1 \cdot (2 \cdot (2^{n-i-1} – 1) – 1)$ have NCA at level $n–1$

  …

  $4^{n-i-1} \cdot (2 \cdot (2^{n-i-(n-i-1)} – 1) – 1)$ 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 DIAGONAL NEIGHBORS

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

$(2^3–1)^2$ neighbor pairs of equal sized nodes in direction NE

NCA = nearest common ancestor

1–13 have NCA at level 3

14–33 have NCA at level 2

34–49 have NCA at level 1

Theorem: average number of nodes visited by `EQUAL_DIAGONAL_NEIGHBOR` is ≤ 16/3

Proof:

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

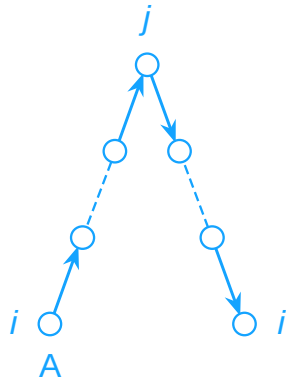
  $4^0 \cdot (2 \cdot (2^{n-i}–1)–1)$ have NCA at level n

  $4^1 \cdot (2 \cdot (2^{n-i-1}–1)–1)$ have NCA at level $n–1$

  …

  $4^{n-i-1} \cdot (2 \cdot (2^{n-i-(n-i-1)}–1)–1)$ 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 DIAGONAL NEIGHBORS

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

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

NCA = nearest common ancestor

1–13 have NCA at level 3

14–33 have NCA at level 2

34–49 have NCA at level 1

Theorem: average number of nodes visited by `EQUAL_DIAGONAL_NEIGHBOR` is ≤ 16/3

Proof:

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

  $4^0 \cdot (2 \cdot (2^{n-i}-1)-1)$ have NCA at level n

  $4^1 \cdot (2 \cdot (2^{n-i-1}-1)-1)$ have NCA at level $n-1$

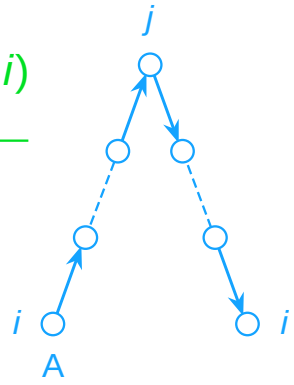
  …

  $4^{n-i-1} \cdot (2 \cdot (2^{n-i-(n-i-1)}-1)-1)$ 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

$$\frac{\sum_{i=0}^{n-1} \sum_{j=i+1}^{n} 4^{n-j} \cdot (2 \cdot (2^{n-i-(n-j)} - 1) - 1) \cdot 2 \cdot (j-i)}{\sum_{i=0}^{n-1} (2^{n-i} - 1)^2}$$

nodes are visited on the average ≤ 16/3



FINDING NEIGHBORS IN HIGHER DIMENSIONS

1. Three dimensions
   - need direction of a vertex
   - direction of an edge = direction of a vertex in two dimensions
   - direction of a face = direction of an edge in two dimensions
2. Arbitrary dimensions (*d*): neighbor of node *N*
   - use induction and routines for adjacencies along 1, 2, ... *d* - 1 dimensions
   - add one new routine for a *d*-dimensional adjacency (e.g., vertex in three dimensions)

   a. ascend the tree if the node is a son of the same type as the direction of the neighbor (ADJ)

   b. for *i* = *d* - 1 step -1 until 1
      - determine if the father of *N* and the ancestor of the desired neighbor have an *i*-dimensional adjacency in which case apply the algorithm for obtaining such a neighbor in *d* - 1 dimensions
      - exit loop upon success

   c. father of *N* is the desired nearest common ancestor
      - retrace path making directly opposite moves about the vertex shared by node *N* and its neighbor

PERFORMANCE FOR TWO-DIMENSIONAL DATA

- Assume 512x512 images
- Results correlate well with the model
- Average cost of neighbor finding for neighbors of greater than or equal size using the position model

	Flood	Topo	Land	Pebble	Average	Predicted
lateral neighbor	3.50	3.60	3.59	3.56	3.57	3.46
diagonal neighbor	4.47	4.68	4.63	4.60	4.60	4.44

- Average cost of stage 1 of neighbor finding (i.e., just locating the nearest common ancestor

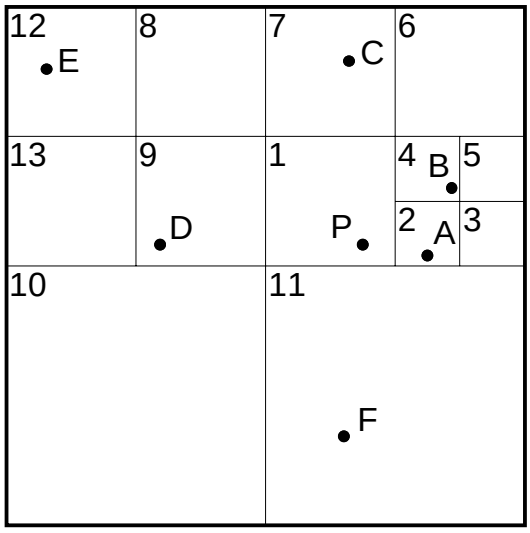
	Flood	Topo	Land	Pebble	Average	Predicted
lateral neighbor	2.01	2.00	2.00	1.99	2.00	1.98
diagonal neighbor	2.69	2.67	2.66	2.65	2.67	2.62

- Average cost of stage 2 of neighbor finding (i.e., descending the tree once the nearest common ancestor has been located)

	Flood	Topo	Land	Pebble	Average	Predicted
lateral neighbor	1.49	1.60	1.59	1.57	1.57	1.48
diagonal neighbor	1.79	2.00	1.97	1.95	1.94	1.82

FINDING THE NEAREST OBJECT

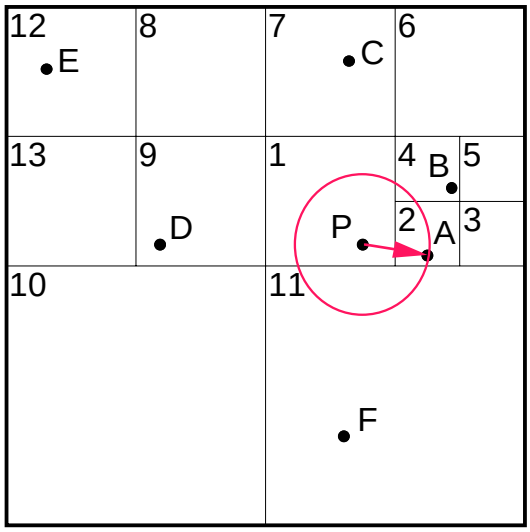
- Ex: find the nearest object to P



- 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

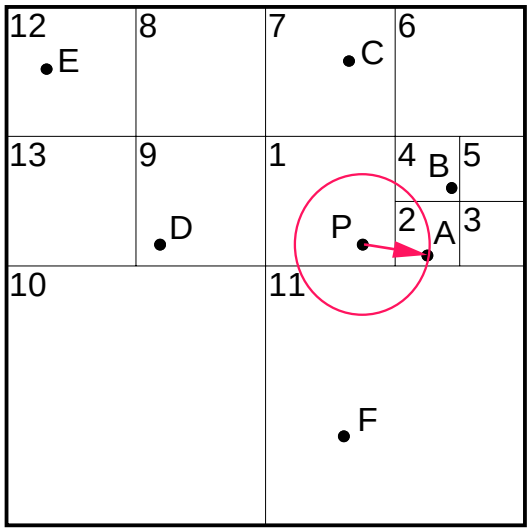


- Assume PR quadtree for points (i.e., at most one point per block)
- Search neighbors of block 1 in counterclockwise order
- Points are sorted with respect to the space they occupy which enables pruning the search space
- Algorithm:

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

FINDING THE NEAREST OBJECT

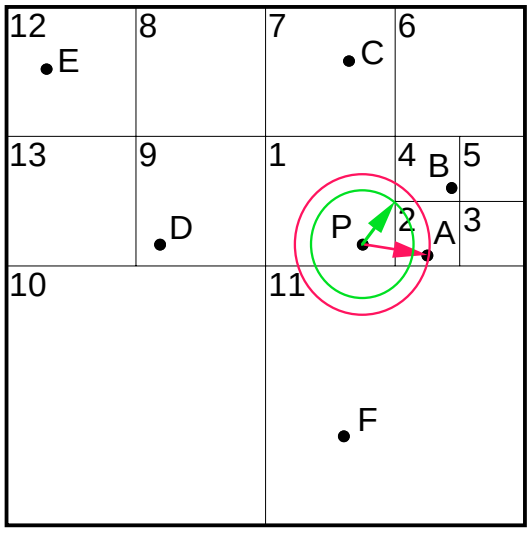
- Ex: find the nearest object to P



- 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

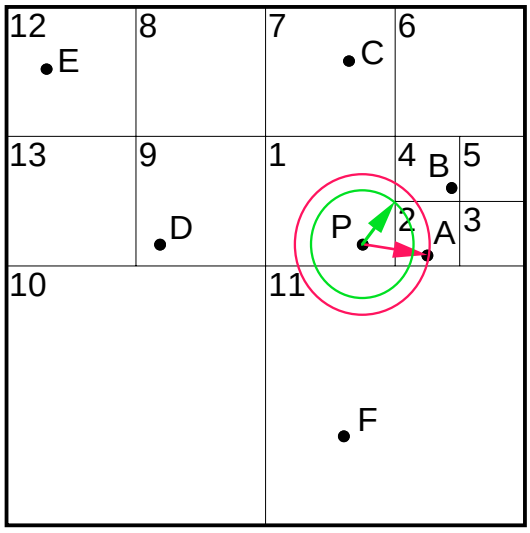
- Ex: find the nearest object to P



- 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

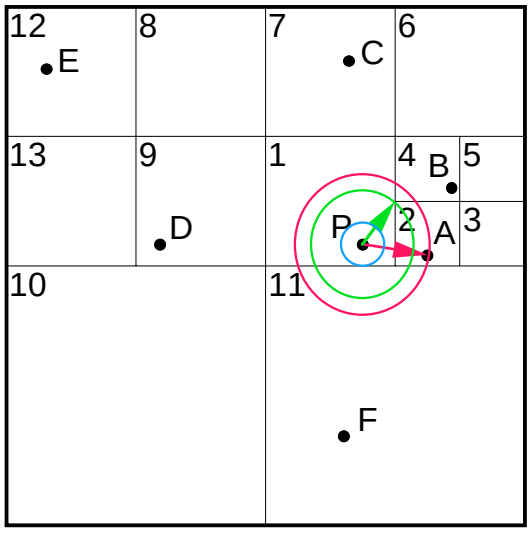
- Ex: find the nearest object to P



- 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

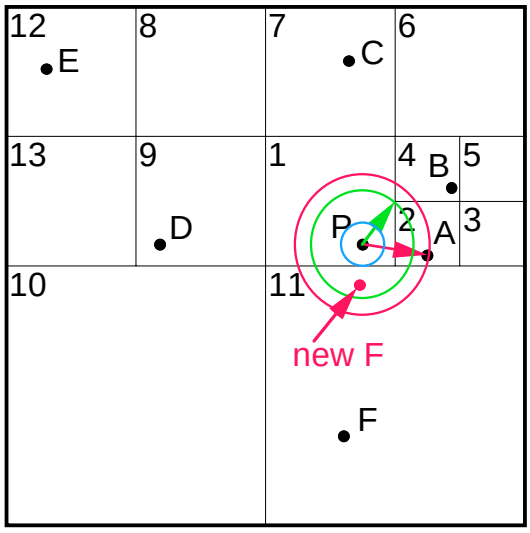
- Ex: find the nearest object to P



- 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

USE OF NEIGHBOR FINDING IN RAY TRACING

- Goal: sort the faces of the objects to reduce the number of necessary ray-object intersection tests
- Trace each ray by checking the blocks through which it passes
- Ex: two-dimensional object