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

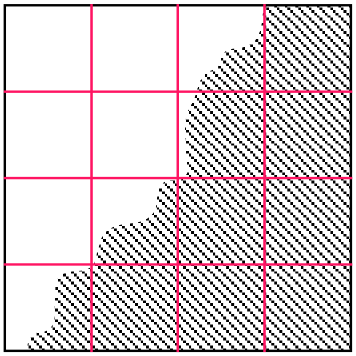
1. Linearization or ordering of higher dimensional space
    - space filling curves — e.g., Peano (1891)
    - spatial index — Morton (1966)
2. Computer graphics
    - sorting objects for display
    - Warnock's algorithm (1968)
        a. vector: hidden-line elimination
        b. raster: hidden-surface 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

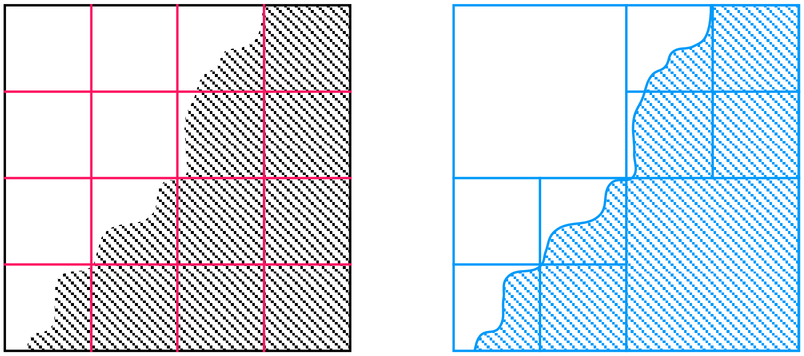
- Subdivide an image until a homogeneity criterion is satisfied — e.g., standard deviation of gray levels is below a particular threshold
- Group adjacent blocks into maximal homogeneous regions
- Ex:



1. initial image decomposition into cells of uniform size

SPLIT-AND-MERGE SEGMENTATION

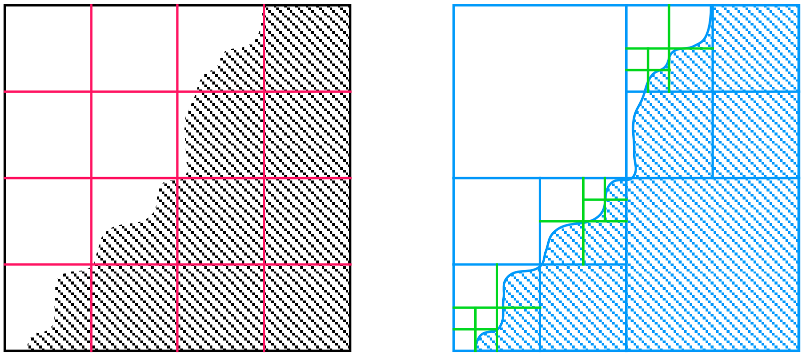
- Subdivide an image until a homogeneity criterion is satisfied — e.g., standard deviation of gray levels is below a particular threshold
- Group adjacent blocks into maximal homogeneous regions
- Ex:



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

SPLIT-AND-MERGE SEGMENTATION

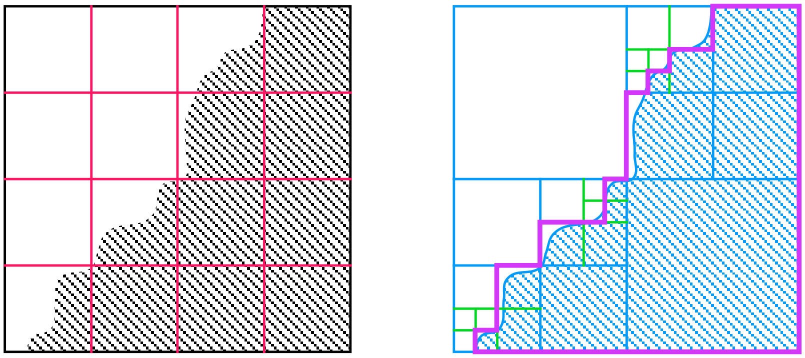
- Subdivide an image until a homogeneity criterion is satisfied — e.g., standard deviation of gray levels is below a particular threshold
- Group adjacent blocks into maximal homogeneous regions
- Ex:



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

SPLIT-AND-MERGE SEGMENTATION

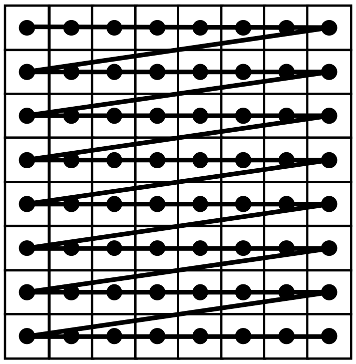
- Subdivide an image until a homogeneity criterion is satisfied — e.g., standard deviation of gray levels is below a particular threshold
- Group adjacent blocks into maximal homogeneous regions
- Ex:

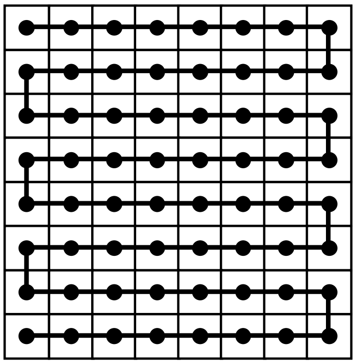


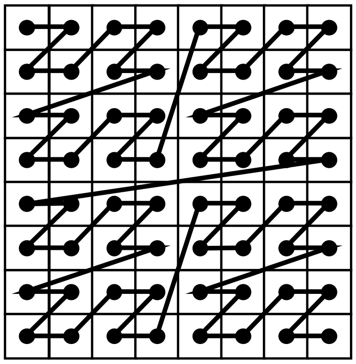
1. initial image decomposition into cells of uniform size
2. merge homogeneous brothers
3. split blocks that are not homogeneous
4. group identical blocks into regions

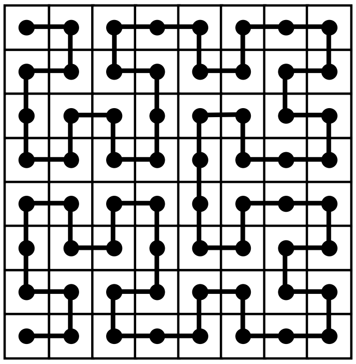
SPACE ORDERING METHODS

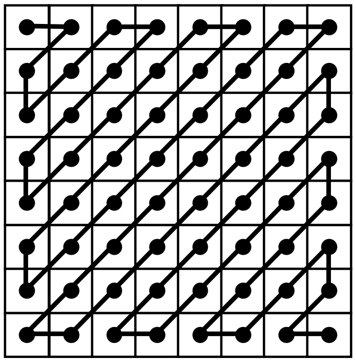
- Linearization of higher dimensional spaces
- Space-filling curves
- Examples:

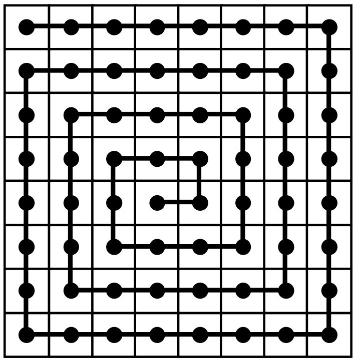

Row order


Row-prime order


Morton order

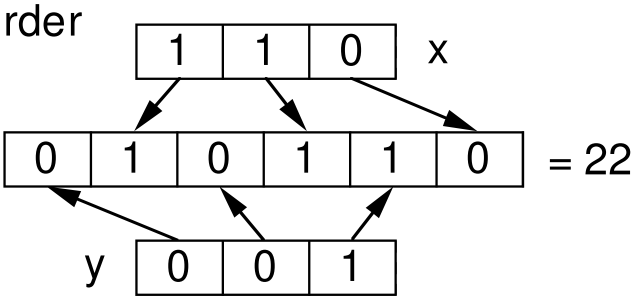

Peano-Hilbert order


Cantor order


Spiral order

CONVERTING BETWEEN POINTS AND CURVES

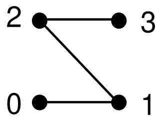
- Need to know size of image for all but the Morton order
- Relatively easy for all but the Peano-Hilbert order which is difficult (although possible) to decode and encode to obtain the corresponding *x* and *y* coordinate values
- Morton order
  1. use bit interleaving of binary representation of the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



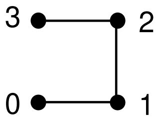
STABILITY OF SPACE ORDERING METHODS

- 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:



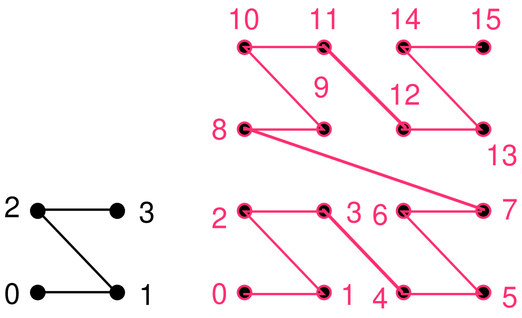
Peano-Hilbert:



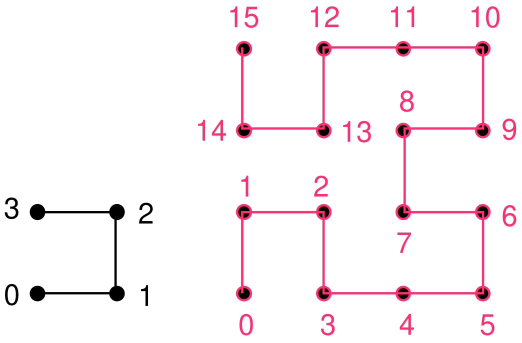
STABILITY OF SPACE ORDERING METHODS

- 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:

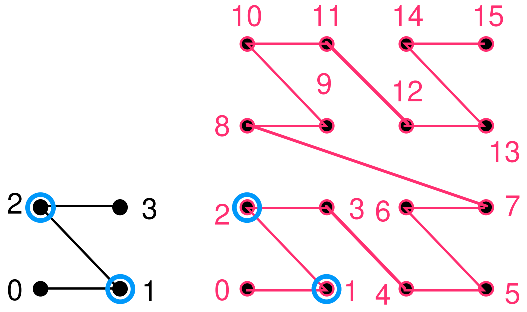


- Result of doubling the resolution (i.e., the coverage)

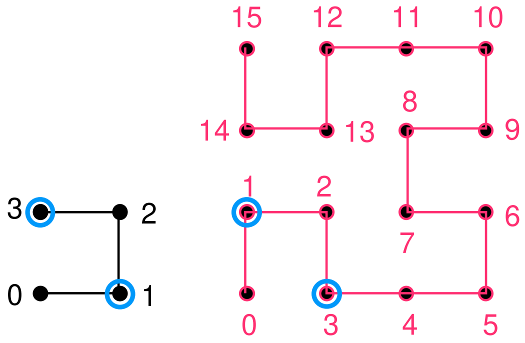
STABILITY OF SPACE ORDERING METHODS

- 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

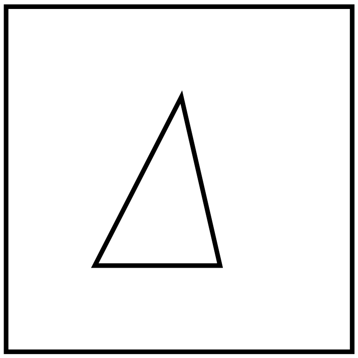
SPACE REQUIREMENTS

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)

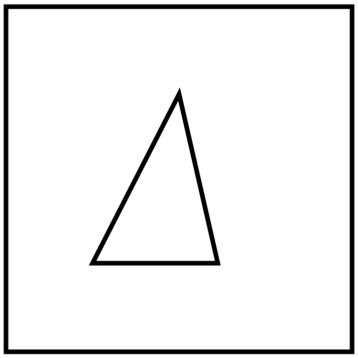
DIMENSION REDUCTION

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.

array



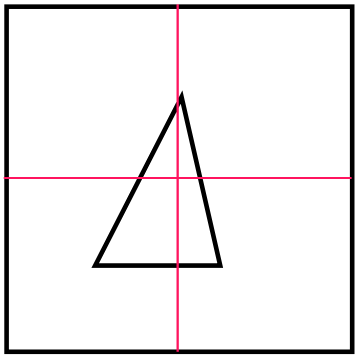
region quadtree



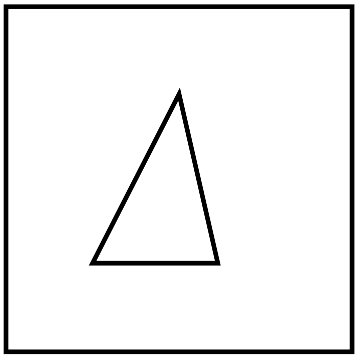
DIMENSION REDUCTION

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.

array



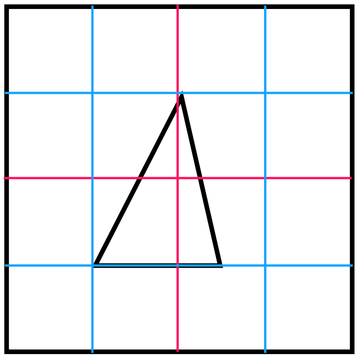
region quadtree



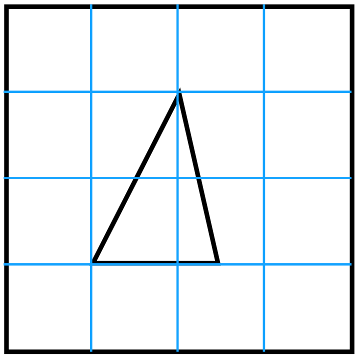
DIMENSION REDUCTION

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.

array

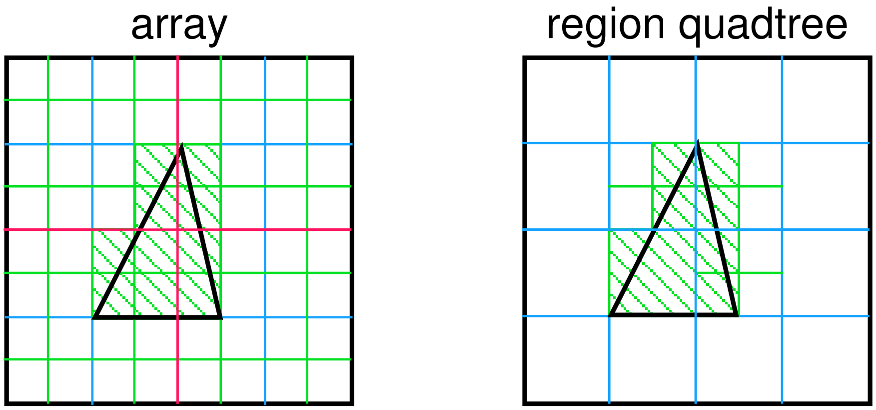


region quadtree



DIMENSION REDUCTION

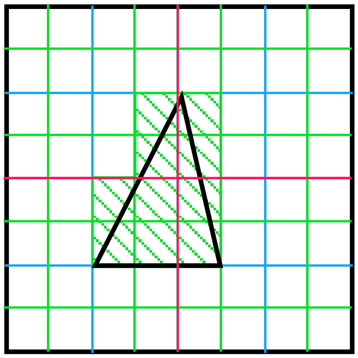
1. Number of blocks necessary to store a simple polygon as a region quadtree is proportional to its perimeter (Hunter)
    - implies that many quadtree algorithms execute in *O*(perimeter) time as they are tree traversals
    - the region quadtree is a dimension reducing device as perimeter (ignoring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


array
region quadtree

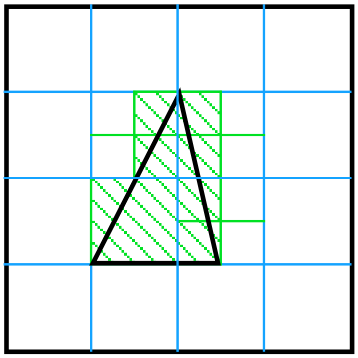
DIMENSION REDUCTION

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.

array



region quadtree



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

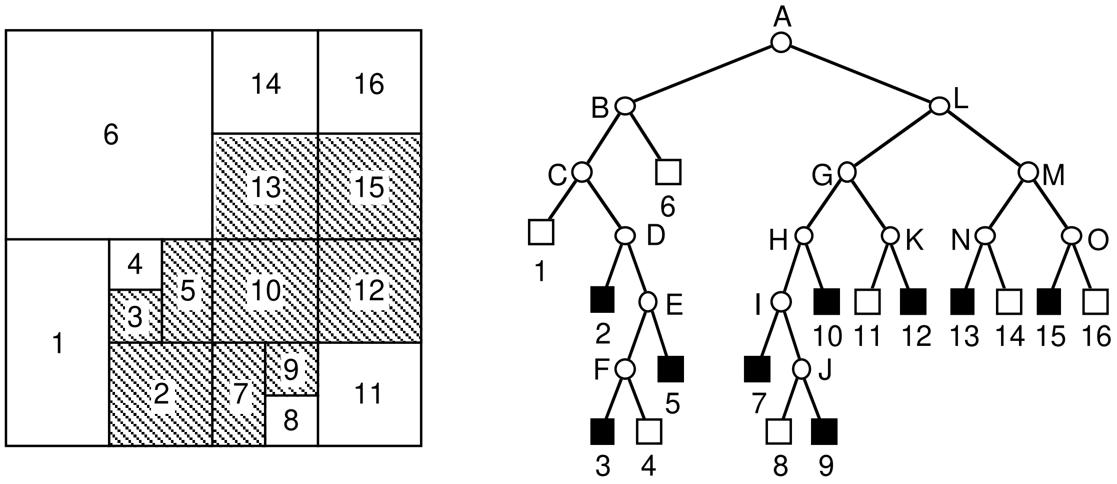
ALTERNATIVE REPRESENTATIONS

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

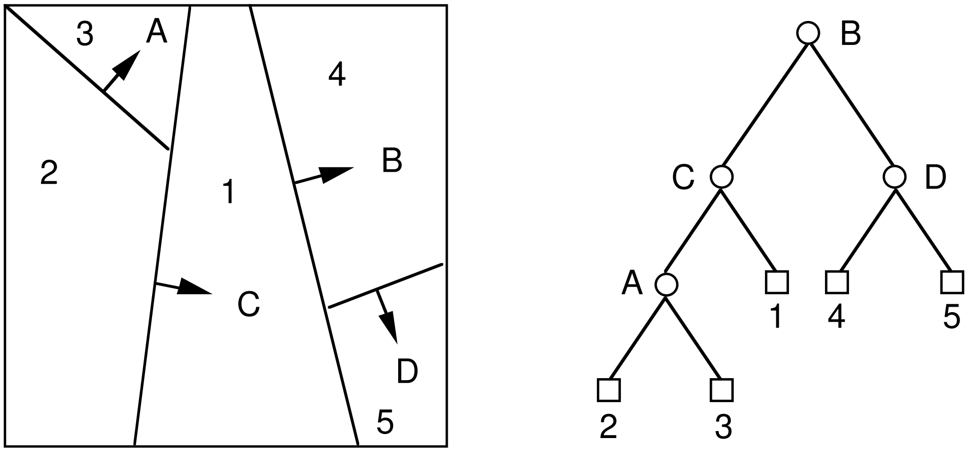
BINTREES (Tamminen, Knowlton)

- 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)

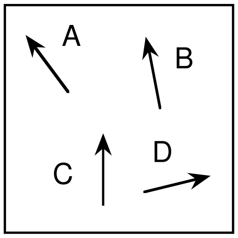
- Like a bintree except that the decomposition lines are at arbitrary orientations (i.e., they need not be parallel or orthogonal)
- For data of arbitrary dimensions
- In 2D (3D), partition along the edges (faces) of a polygon (polyhedron)
- Ex: arrows indicate direction of positive area



- 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

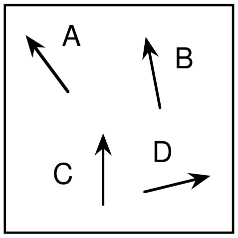
DRAWBACKS OF BSP TREES

- 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

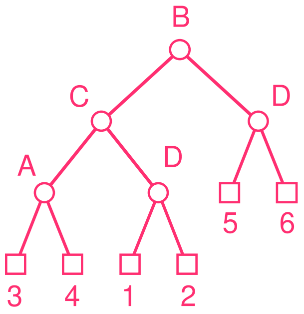


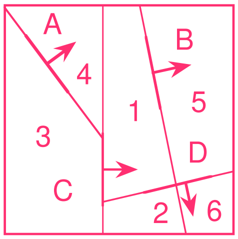
DRAWBACKS OF BSP TREES

- 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



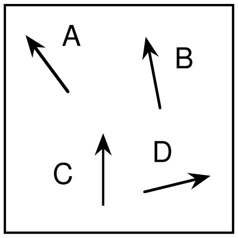
1. partition induced by choosing B as the root



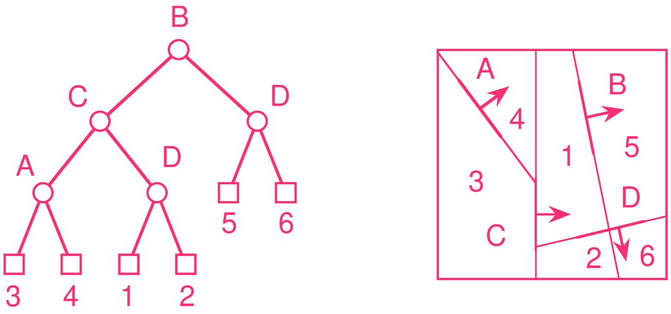


DRAWBACKS OF BSP TREES

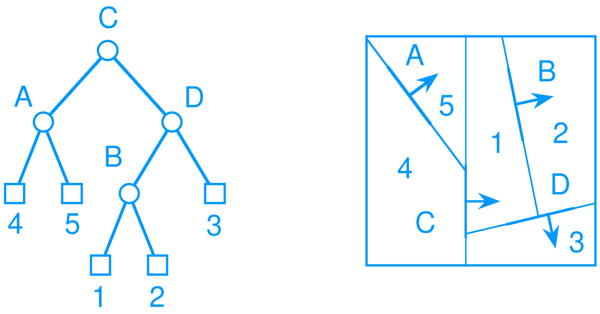
- A polygon may be included in both the left and right subtrees of node
- Same issues of duplicate reporting as in representations based on a disjoint decomposition of the underlying space
- Shape of the BSP tree depends on the order in which the polygons are processed and on the polygons chosen to serve 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



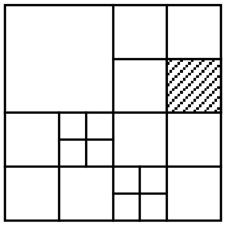
2. partition induced by choosing C as the root



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

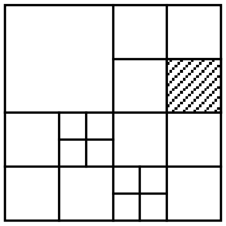
       Ex:



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

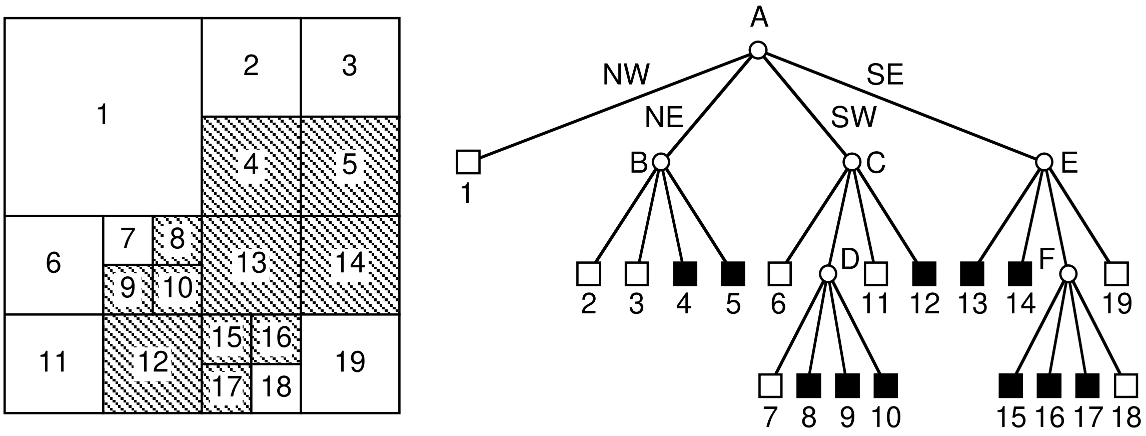
- Locational codes can be stored in a B-tree
- Locational codes are equivalent to a hashing function and are the basis of techniques known as *order preserving linear hashing*
- Sorting locational codes in increasing order has the effect of a space-filling curve and is equivalent to traversing the leaf 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



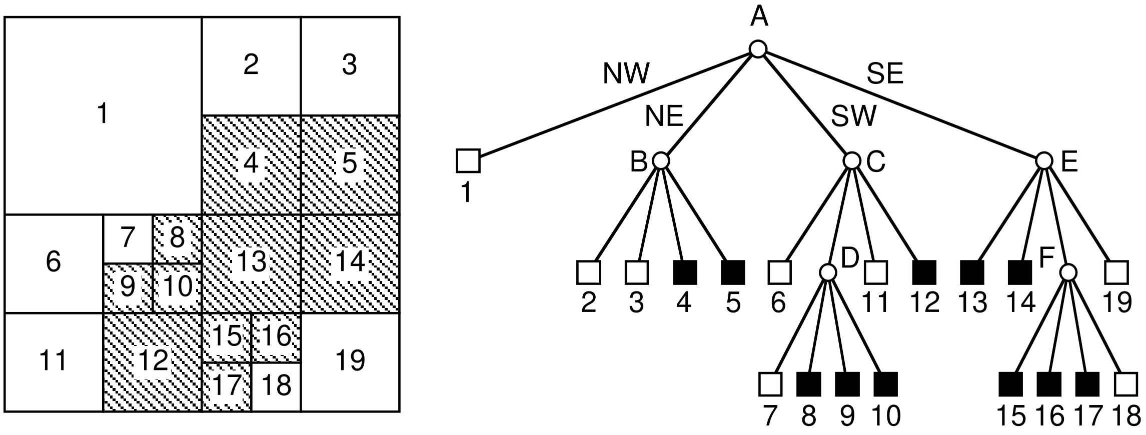
- Drawback: random access is impossible (e.g., for neighbor finding — must always start at the first element in the list and visit all elements prior to the one being searched for
- Useful whenever have to process entire set of nodes in preorder (e.g., NW, NE, SW, SE)
  1. centroid computation
  2. set-theoretic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

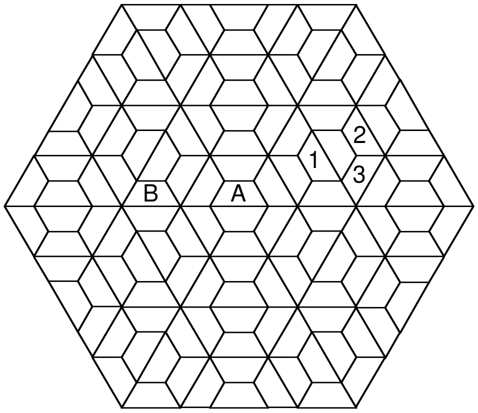
TESSELLATIONS

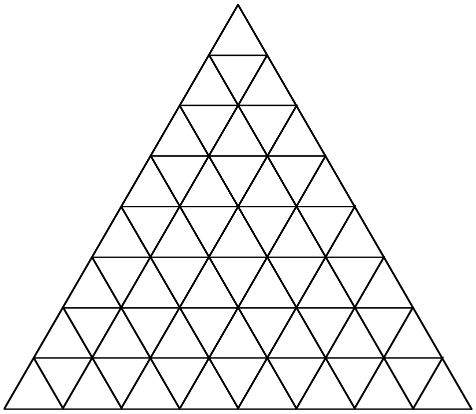
Hanan Samet

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

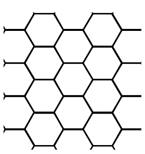
ALTERNATIVE DECOMPOSITION METHODS

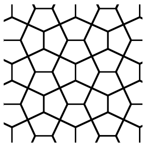
- A planar decomposition for image representation should be:
  1. infinitely repetitive
  2. infinitely decomposable into successively finer patterns
- Classification of tilings (Bell, Diaz, Holroyd, and Jackson)
  1. isohedral — all tiles are equivalent under the symmetry group of the tiling (i.e.,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

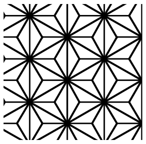


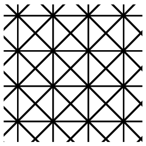


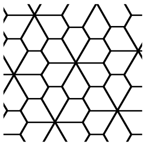
  2. regular — each tile is a regular polygon
- There are 81 types if classify by their symmetry groups
- Only 11 types if classify by their adjacency structure

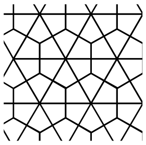

$[3^6]$

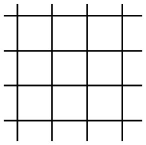

$[3^2.4.3.2]$

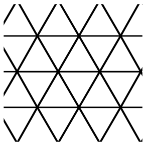

$[3.12^2]$

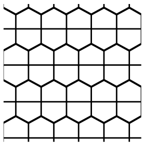

$[4.8^2]$

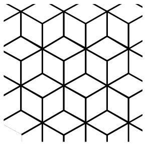

$[3^4.6]$

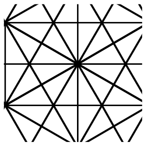

$[3.4.6.4]$


$[4^4]$


$[6^3]$


$[3^3.4^2]$


$[3.6.3.6]$

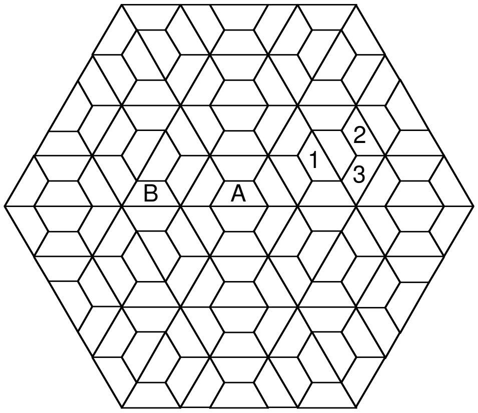

$[4.6.12]$

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

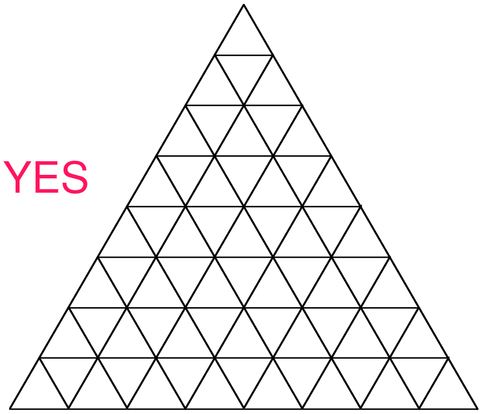
ALTERNATIVE DECOMPOSITION METHODS

- 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)

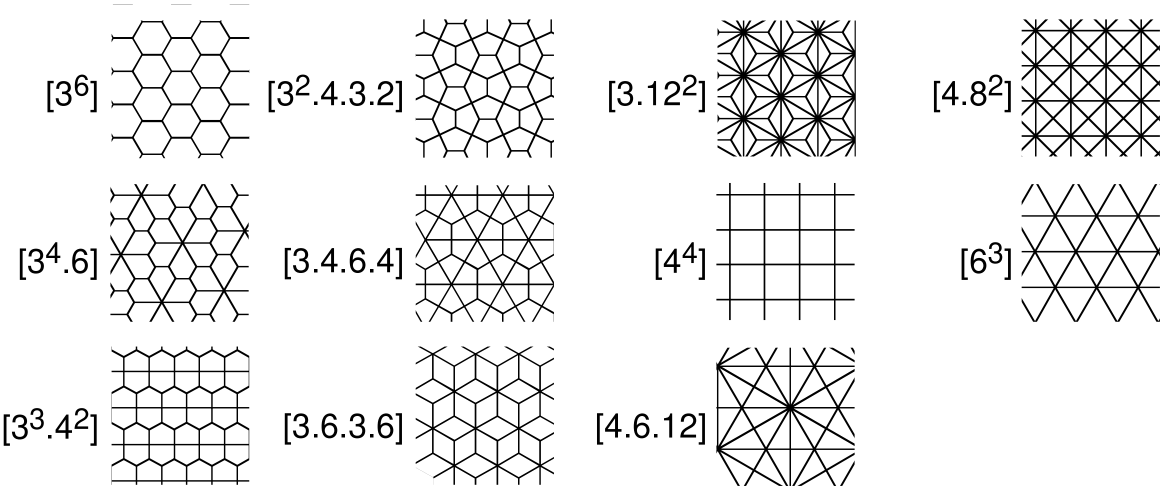
NO



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

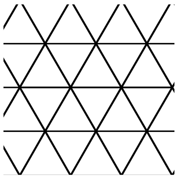


- [3.12^2] means 3 edges at the first vertex of the polygonal tile followed by 12 edges at the next two vertices

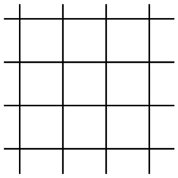
PROPERTIES OF TILINGS — SIMILARITY

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

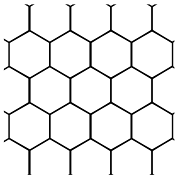
$[6^3]$



$[4^4]$

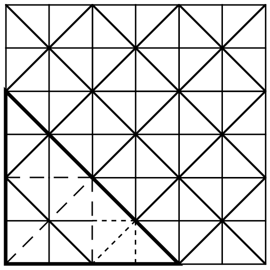


$[3^6]$

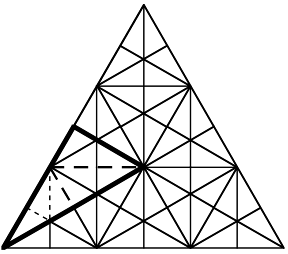


- Limited ≡ NOT similar (i.e., cannot be decomposed infinitely into smaller tiles of the same shape)
- Unlimited: each edge of each tile lies on an infinitely straight line composed entirely of edges
- Only 4 unlimited tilings $[4^4]$, $[6^3]$, $[4.8^2]$, and [4.6.12]

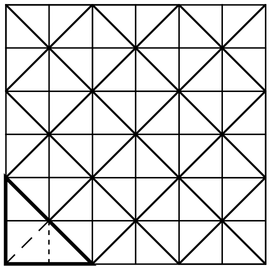
$[4.8^2]$



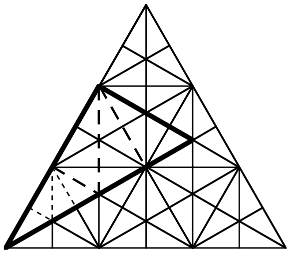
[4.6.12]



- Two additional hierarchies:



rotation of 135° between levels



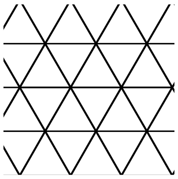
reflection between levels

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

PROPERTIES OF TILINGS — SIMILARITY

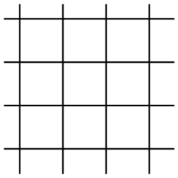
- Similarity — a tile at level *k* has the same shape as a tile at level 0 (basic tile shape)

[6^3]



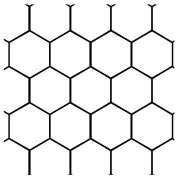
YES

[4^4]



YES

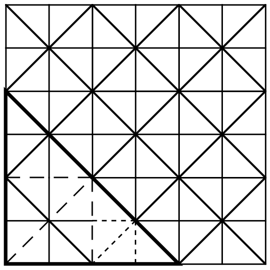
[3^6]



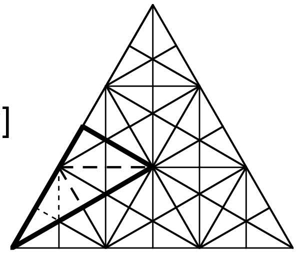
NO

- 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]

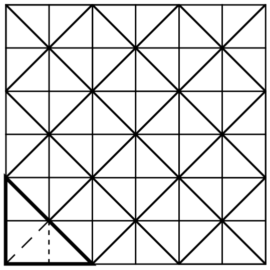
[4.8^2]



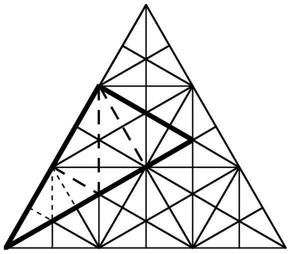
[4.6.12]



- Two additional hierarchies:



rotation of 135° between levels



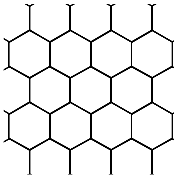
reflection between levels

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

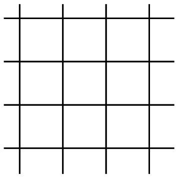
PROPERTIES OF TILINGS — ADJACENCY

- 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

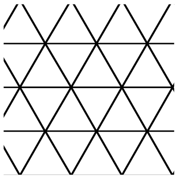
$[3^6]$



$[4^4]$

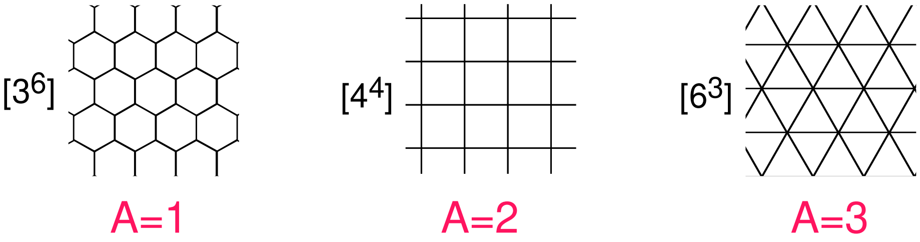


$[6^3]$



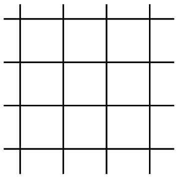
PROPERTIES OF TILINGS — ADJACENCY

- 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

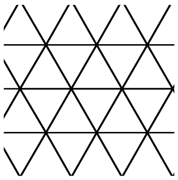


PROPERTIES OF TILINGS — UNIFORM ORIENTATION

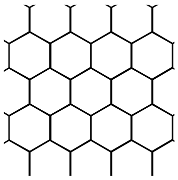
- Uniform orientation
- All tiles with the same orientation can be mapped into each other by translations of the plane which do not involve rotation for reflection



[4^4]



[6^3]



[3^6]

Conclusion:

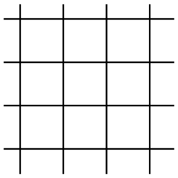
- [4^4] has a lower adjacency number than [6^3]
- [4^4] has a uniform orientation while [6^3] does not
- [4^4] is unlimited while [3^6] is limited

Use [4^4]!

PROPERTIES OF TILINGS — UNIFORM ORIENTATION

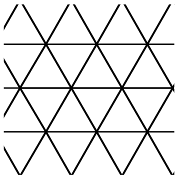
- Uniform orientation
- All tiles with the same orientation can be mapped into each other by translations of the plane which do not involve rotation for reflection

$[4^4]$



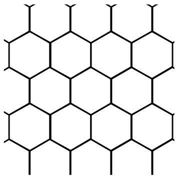
YES

$[6^3]$



NO

$[3^6]$



YES

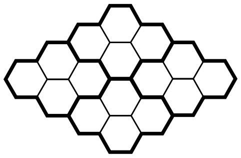
Conclusion:

- $[4^4]$ 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

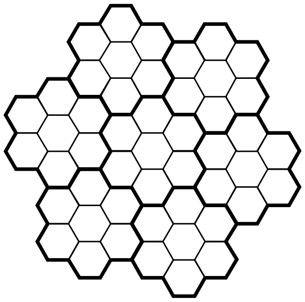
  Use $[4^4]$!

HEXAGONAL TESSELLATIONS [3^6]

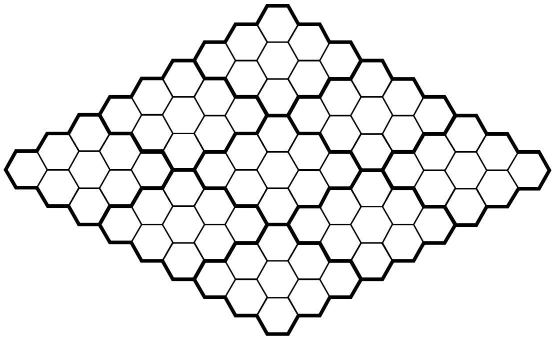
1. Still of interest
   - regular
   - uniform orientation
   - uniform adjacency



4-shape



7-shape



9-shape

2. Several tiling hierarchies (*n*-shapes) NOT UNIQUE!
   - *n* ≡ 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 (≡ rosette and termed a septree)

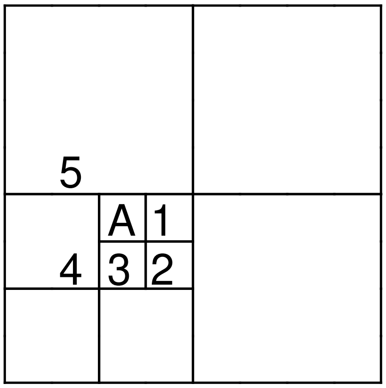
NEIGHBOR FINDING METHODS IN QUADTREES

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

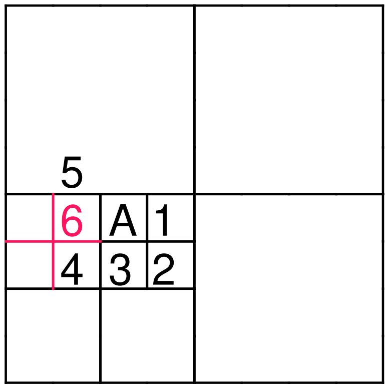


A has 5 neighbors

- 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

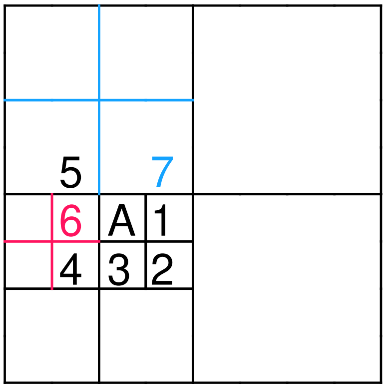


A has ~~5~~ 6 neighbors

- 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

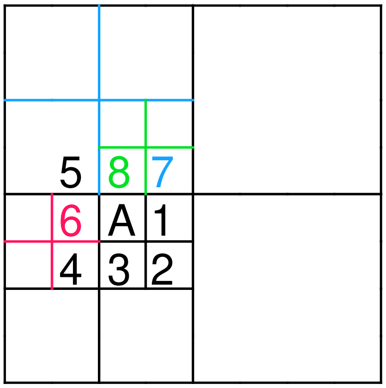


A has ~~5~~ ~~6~~ 7 neighbors

- 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

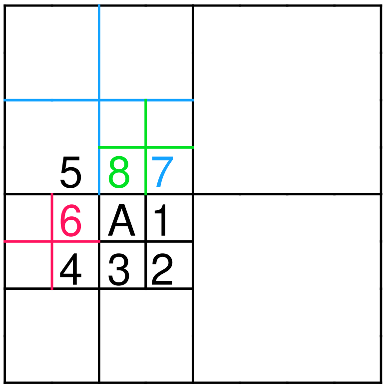


A has ~~5~~ ~~6~~ ~~7~~ 8 neighbors

- 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



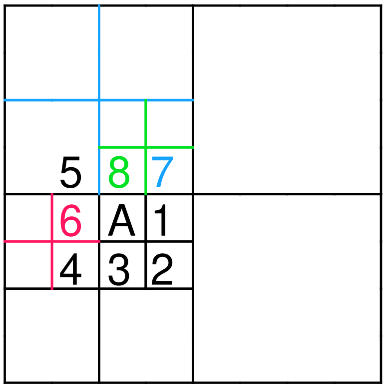
A has ~~5~~ ~~6~~ ~~7~~ 8 neighbors

- 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

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

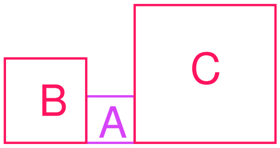
NEIGHBOR FINDING OPERATIONS USING QUADTREES

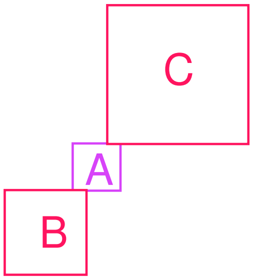
- Many image processing operations involve traversing an image and applying an operation to a pixel and some of its neighboring (i.e., adjacent) pixels
  - For quadtree/octree representations replace pixel/voxel by block
  - Neighbor is defined to be an adjacent block of greater than or equal size



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

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



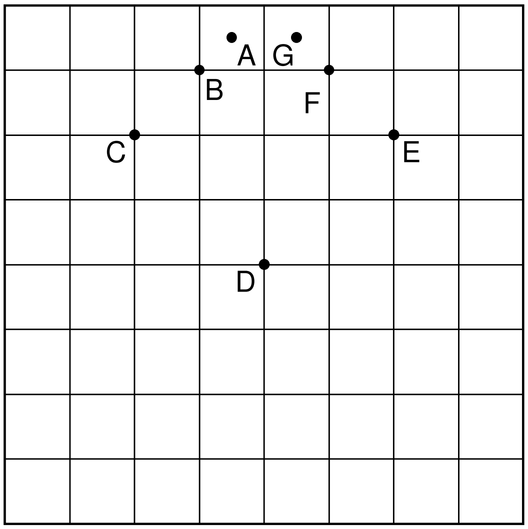


FINDING LATERAL NEIGHBORS OF EQUAL SIZE

Algorithm: based on finding the nearest common ancestor

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

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

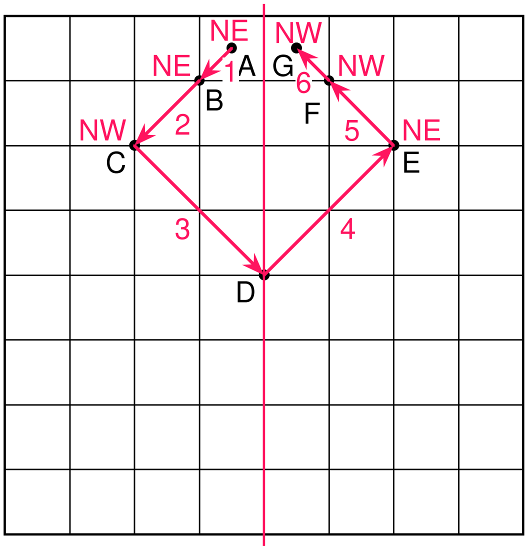


FINDING LATERAL NEIGHBORS OF EQUAL SIZE

Algorithm: based on finding the nearest common ancestor

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

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

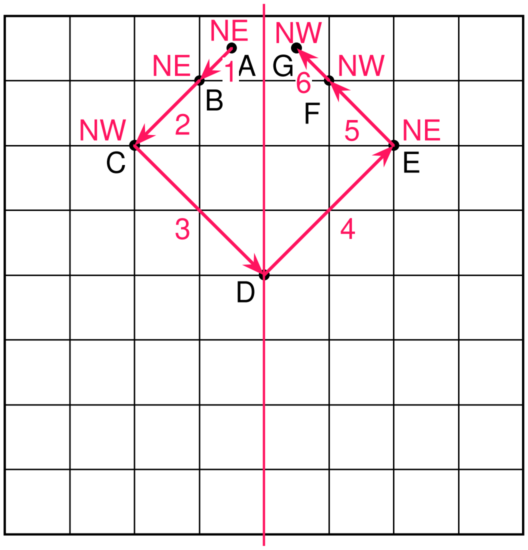


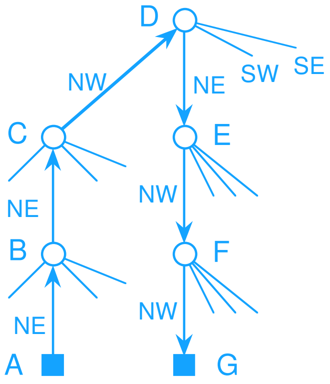
FINDING LATERAL NEIGHBORS OF EQUAL SIZE

Algorithm: based on finding the nearest common ancestor

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

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



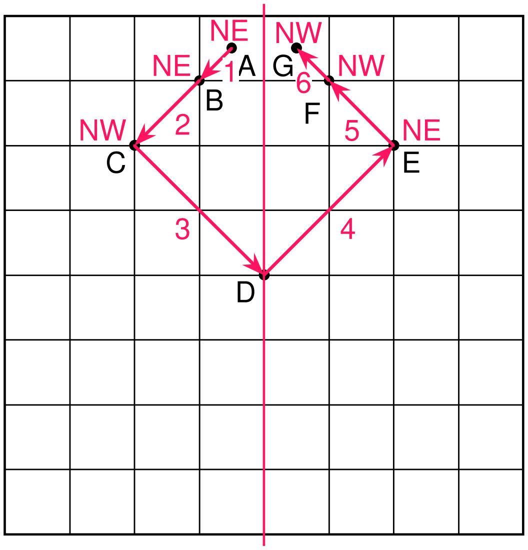


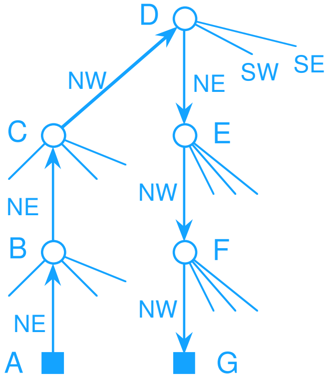
FINDING LATERAL NEIGHBORS OF EQUAL SIZE

Algorithm: based on finding the nearest common ancestor

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

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





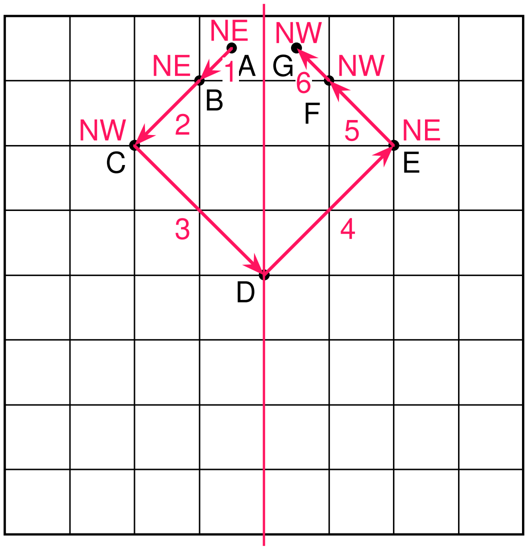
```
node procedure EQUAL_LATERAL_NEIGHBOR(P,D);
/* Find = size neighbor of P in direction D */
begin
  value pointer node P;
  value direction D;
  return(SON(if ADJ(D,SONTYPE(P)) then
                 EQUAL_LATERAL_NEIGHBOR(FATHER(P),D)
             else FATHER(P),
             REFLECT(D,SONTYPE(P))));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

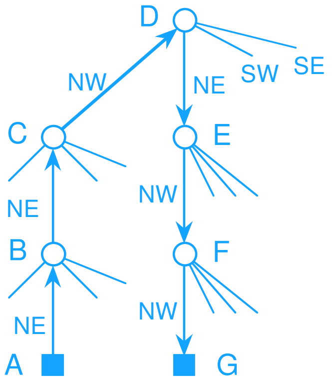
FINDING LATERAL NEIGHBORS OF EQUAL SIZE

Algorithm: based on finding the nearest common ancestor

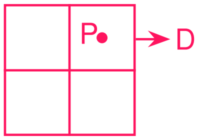
1. Ascend the tree if the node is a son of the same type as the direction of the neighbor (ADJ)
2. Otherwise, the father F is the nearest common ancestor and retrace the path starting at F making mirror image moves about the edge shared by the neighboring blocks

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





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



ADJ(A,B)

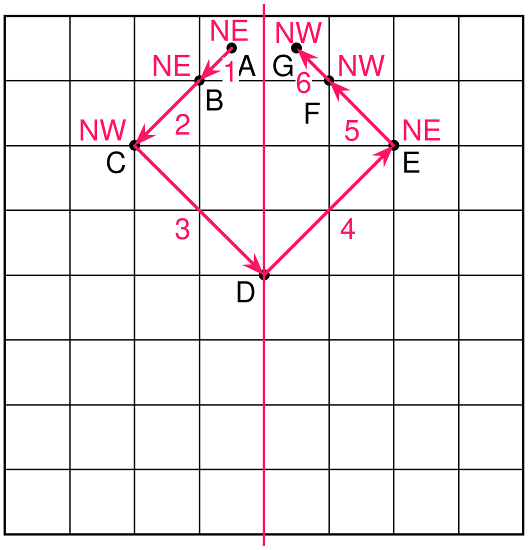
A \ B	NW	NE	SW	SE
N	T	T	F	F
E	F	T	F	T
S	F	F	T	T
W	T	F	T	F

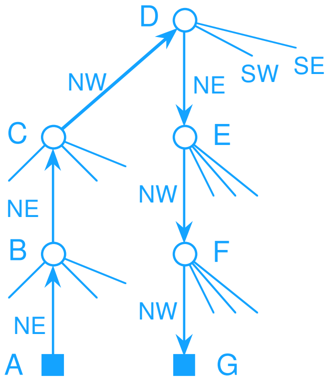
FINDING LATERAL NEIGHBORS OF EQUAL SIZE

Algorithm: based on finding the nearest common ancestor

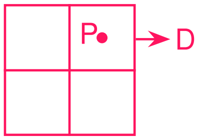
1. Ascend the tree if the node is a son of the same type as the direction of the neighbor (ADJ)
2. Otherwise, the father F is the nearest common ancestor and retrace the path starting at F making mirror image moves about the edge shared by the neighboring blocks

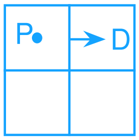
Ex: E neighbor of A (i.e., G)





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





ADJ(A,B)

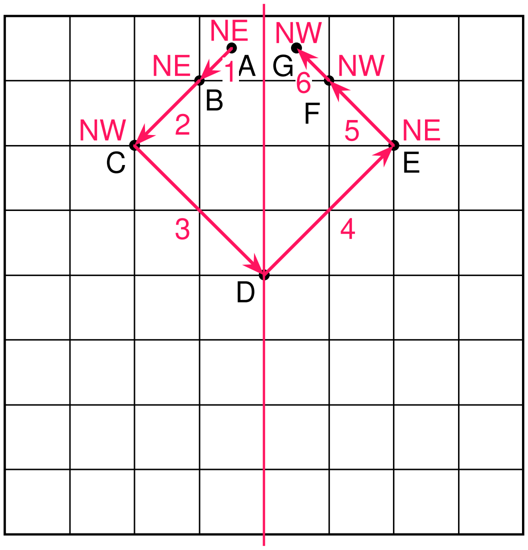
A \ B	NW	NE	SW	SE
N	T	T	F	F
E	F	T	F	T
S	F	F	T	T
W	T	F	T	F

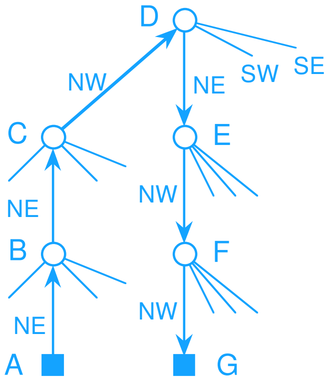
FINDING LATERAL NEIGHBORS OF EQUAL SIZE

Algorithm: based on finding the nearest common ancestor

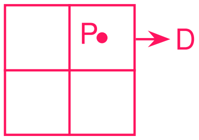
1. Ascend the tree if the node is a son of the same type as the direction of the neighbor (ADJ)
2. Otherwise, the father F is the nearest common ancestor and retrace the path starting at F making mirror image moves about the edge shared by the neighboring blocks

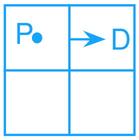
Ex: E neighbor of A (i.e., G)





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





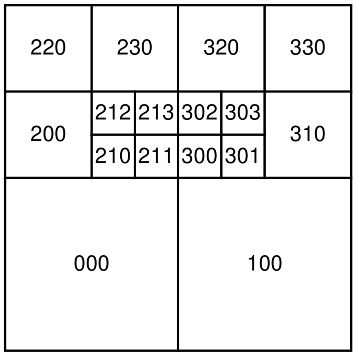
ADJ(A,B)

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

REFLECT(A,B)

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

FINDING NEIGHBORS IN A POINTERLESS REPRESENTATION



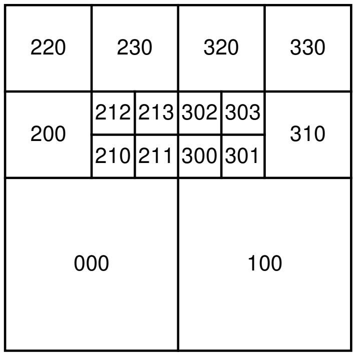
- Assume a bit-interleaved representation that also indicates the depth of each block
- Ex: *y* bit is more significant than *x* bit — i.e., $y_n\, x_n \ldots y_1\, x_1\, y_0\, x_0$

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., $y_n\, x_n \ldots y_1\, x_1\, y_0\, x_0$



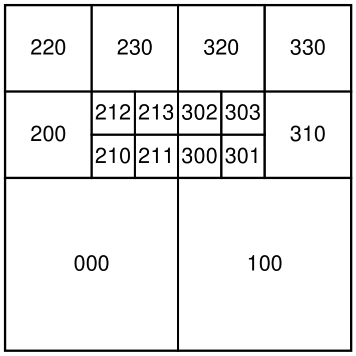
1. Mimic tree algorithm in the sense that to find a lateral neighbor (say in the *x* direction), search for the first *x* bit position that is different while complementing the bits encountered in the process

   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., $y_n\ x_n \ldots y_1\ x_1\ y_0\ x_0$



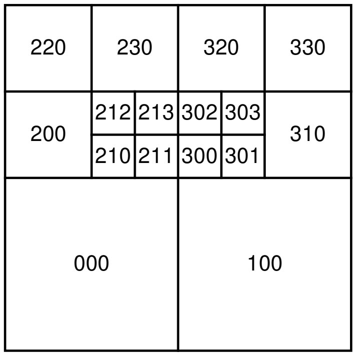
1. Mimic tree algorithm in the sense that to find a lateral neighbor (say in the *x* direction), search for the first *x* bit position that is different while complementing the bits encountered in the process

   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., $y_n\ x_n \ldots y_1\ x_1\ y_0\ x_0$

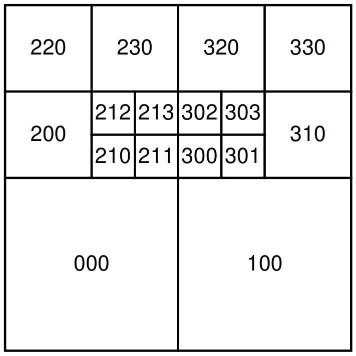


1. Mimic tree algorithm in the sense that to find a lateral neighbor (say in the *x* direction), search for the first *x* bit position that is different while complementing the bits encountered in the process

   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`

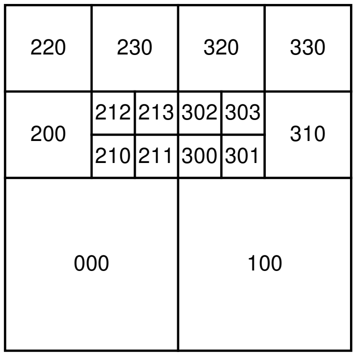
   - search for an x bit with a value of 0
   - result is `302 = 11 00 10`
   - cumbersome as many bit operations are required

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

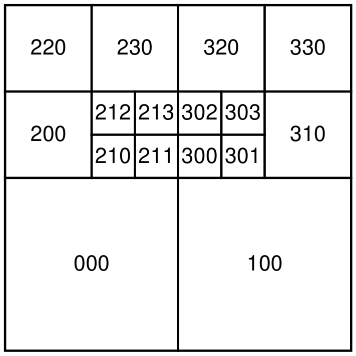
2. Use arithmetic by adding or subtracting from the appropriate *x* (*y*) bit and skip the positions corresponding to the *y* (or *x*) bits

   - 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

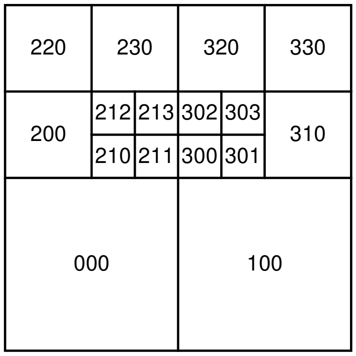
   Ex: find pixel-sized eastern neighbor of `213 = 10 01 11`

   - 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

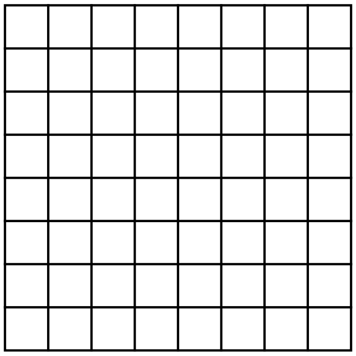
Ex: find pixel-sized eastern neighbor of `213 = 10 01 11`
the addend 00 00 01 becomes `+ 10 10 11`
result `01 00 10`
restoring *y* bit positions to previous `11 00 10 = 302`

- 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

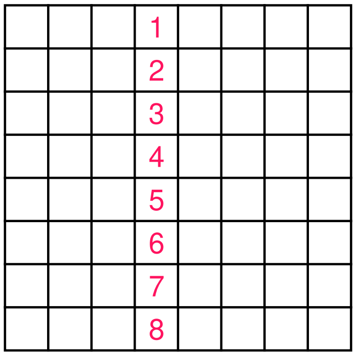
ANALYSIS OF FINDING LATERAL NEIGHBORS



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

NCA = nearest common ancestor

ANALYSIS OF FINDING LATERAL NEIGHBORS

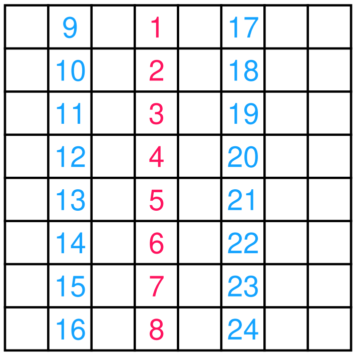


$2^3\cdot(2^3-1)$ neighbor pairs of equal sized nodes in direction E

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		

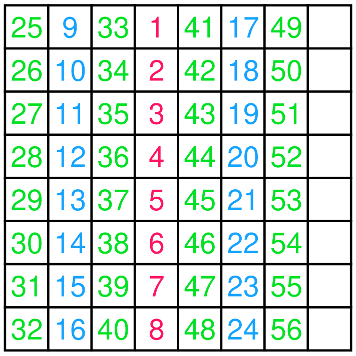
$2^3\cdot(2^3-1)$ neighbor pairs of equal sized nodes in direction E

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

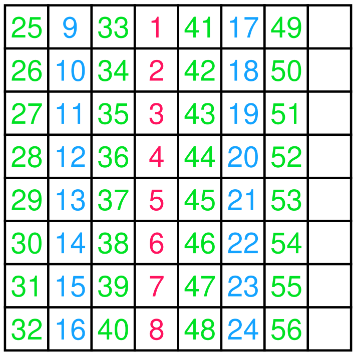
Theorem: average number of nodes visited by `EQUAL_LATERAL_NEIGHBOR` is ≤ 4

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

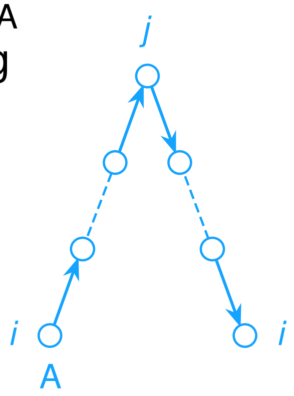
25–56 have NCA at level 1

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



ANALYSIS OF FINDING LATERAL NEIGHBORS

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

$2^3\cdot(2^3-1)$ neighbor pairs of equal sized nodes in direction E
NCA = nearest common ancestor

1–8 have NCA at level 3
9–24 have NCA at level 2
25–56 have NCA at level 1

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

Proof:

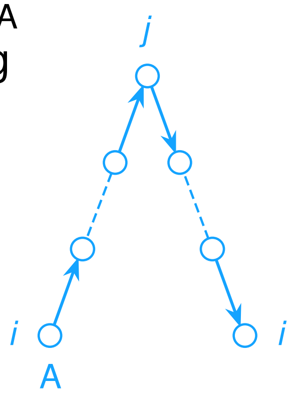
- Let node A be at level i (i.e., a $2^i \times 2^i$ block)
- There are $2^{n-i}\cdot(2^{n-i}-1)$ possible positions for node A such that an equal sized neighbor exists in a given horizontal or vertical direction

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

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

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

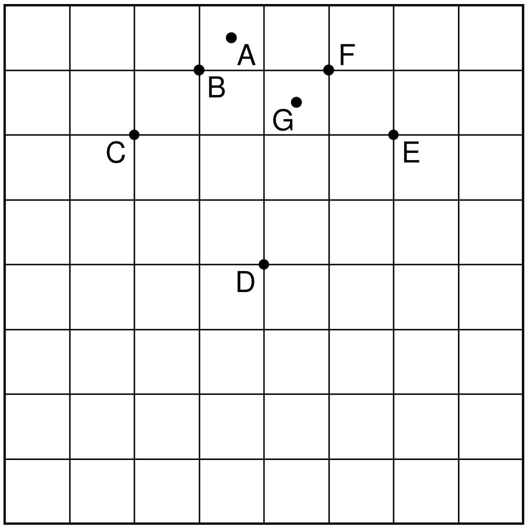
nodes are visited on the average ≤ 4



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

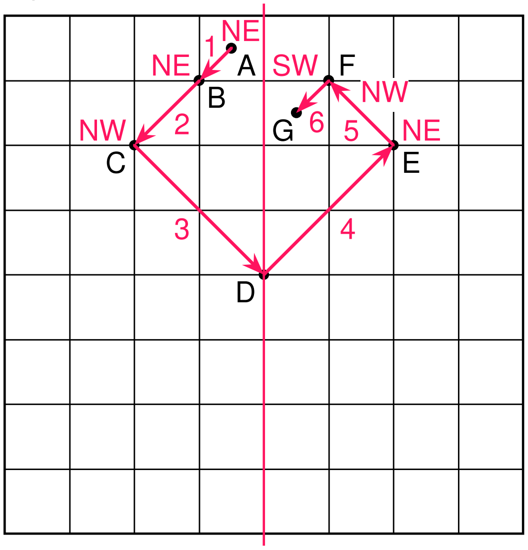


Ex: SE neighbor of A (i.e., G)

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



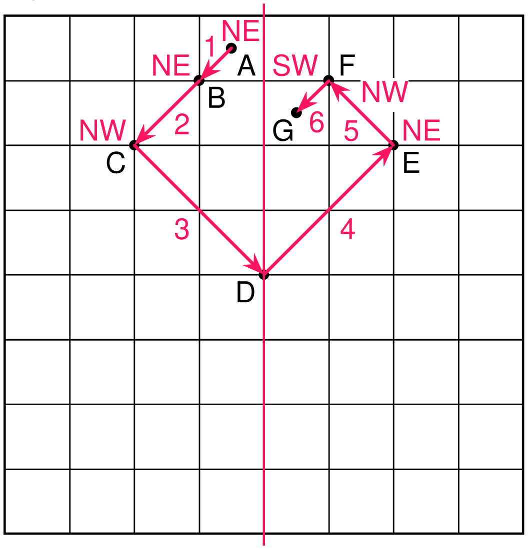
Ex: SE neighbor of A (i.e., G)

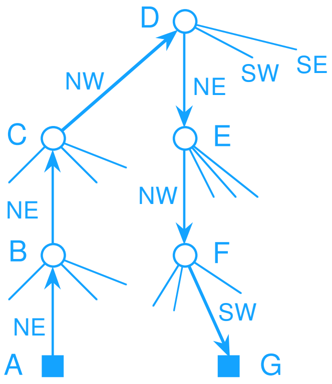
FINDING DIAGONAL NEIGHBORS OF EQUAL SIZE

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)

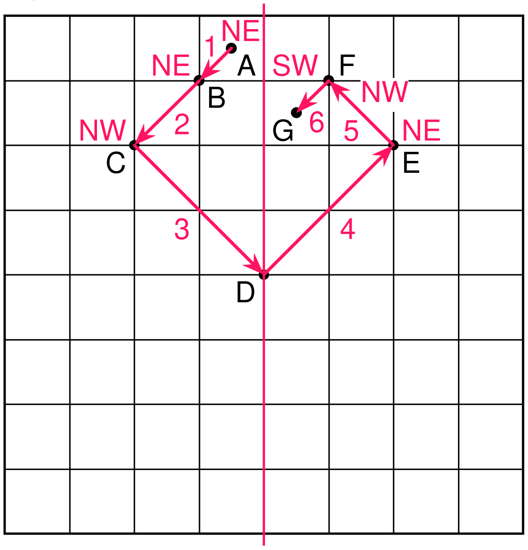




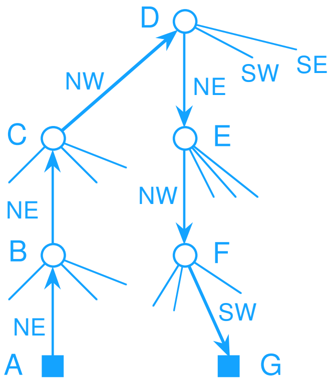
FINDING DIAGONAL NEIGHBORS OF EQUAL SIZE

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)

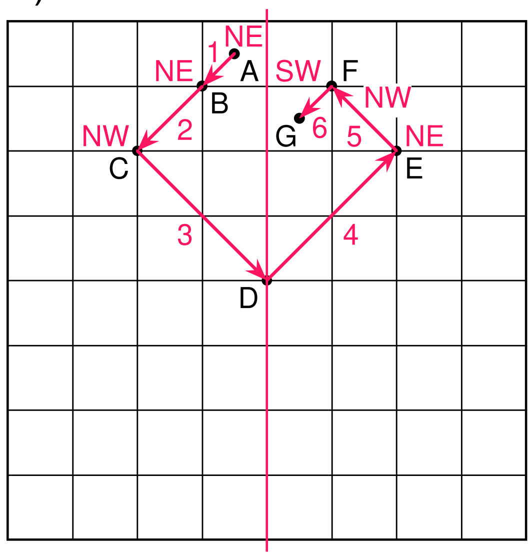


```
node procedure EQUAL_DIAGONAL_NEIGHBOR(P,C);
/* Find = size neighbor of P towards quadrant C */
begin
  value pointer node P;
  value quadrant C;
  return(SON(if ADJ(C,SONTYPE(P)) then
               EQUAL_DIAGONAL_NEIGHBOR(FATHER(P),C)
             else if COMMON_EDGE(C,SONTYPE(P))≠Ω then
     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

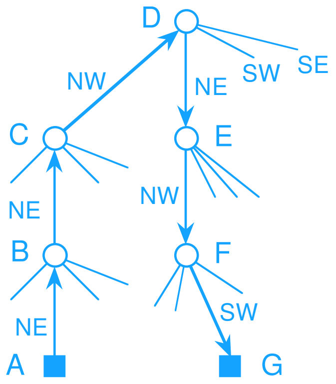
FINDING DIAGONAL NEIGHBORS OF EQUAL SIZE

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

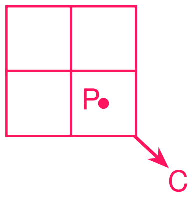


Ex: `SE` neighbor of `A` (i.e., `G`)



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)`

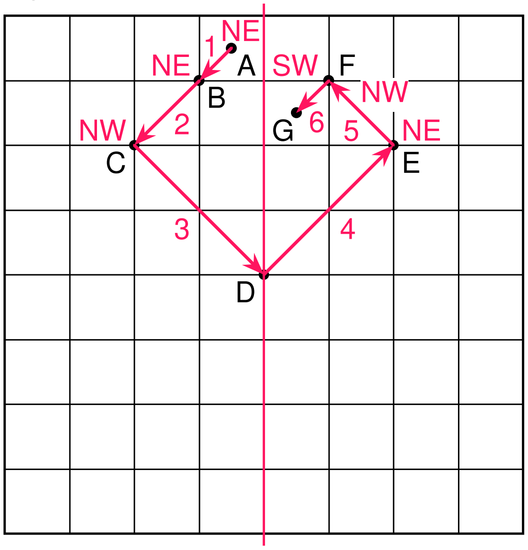


```
node procedure EQUAL_DIAGONAL_NEIGHBOR(P,C);
/* Find = size neighbor of P towards quadrant C */
begin
  value pointer node P;
  value quadrant C;
  return(SON(if ADJ(C,SONTYPE(P)) then
                 EQUAL_DIAGONAL_NEIGHBOR(FATHER(P),C)
               else if COMMON_EDGE(C,SONTYPE(P))≠Ω then
 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

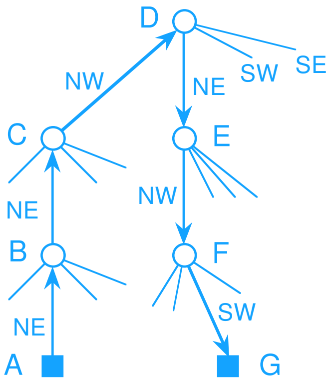
FINDING DIAGONAL NEIGHBORS OF EQUAL SIZE

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)



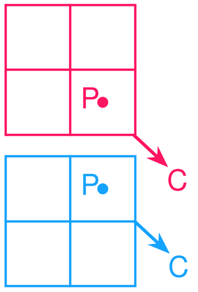
A \ B	NW	NE	SW	SE
NW	T	F	F	F
NE	F	T	F	F
SW	F	F	T	F
SE	F	F	F	T

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;
```

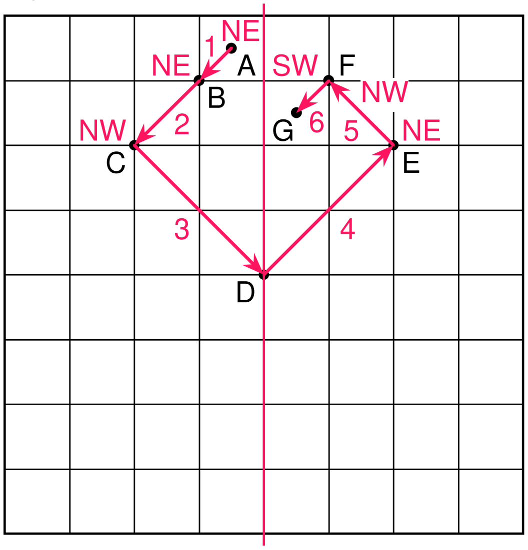


7 6 5 4 3 2 1
g z r b z r b

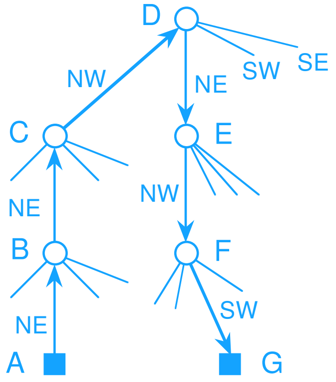
FINDING DIAGONAL NEIGHBORS OF EQUAL SIZE

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`)



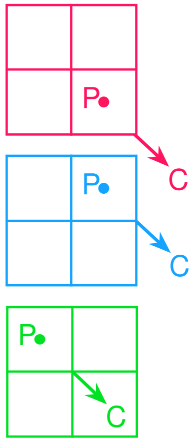
B A	NW	NE	SW	SE
NW	T	F	F	F
NE	F	T	F	F
SW	F	F	T	F
SE	F	F	F	T

`ADJ(A,B)`

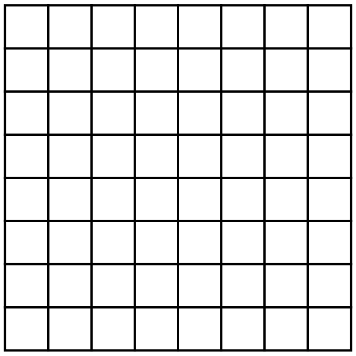
B A	NW	NE	SW	SE
NW	Ω	N	W	Ω
NE	N	Ω	Ω	E
SW	W	Ω	Ω	S
SE	Ω	E	S	Ω

`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;
```



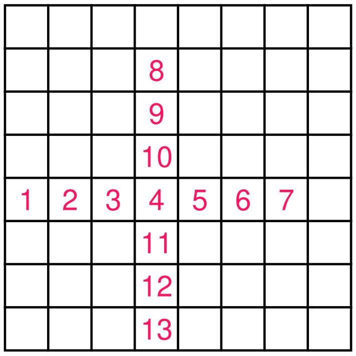
ANALYSIS OF FINDING DIAGONAL NEIGHBORS



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

NCA = nearest common ancestor

ANALYSIS OF FINDING DIAGONAL NEIGHBORS

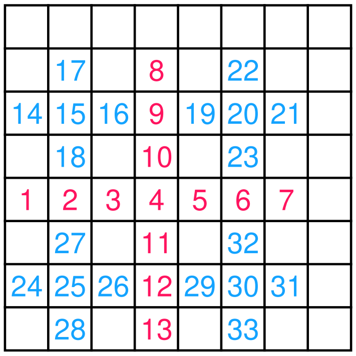


$(2^3–1)^2$ neighbor pairs of equal sized nodes in direction NE

NCA = nearest common ancestor

1–13 have NCA at level 3

ANALYSIS OF FINDING DIAGONAL NEIGHBORS



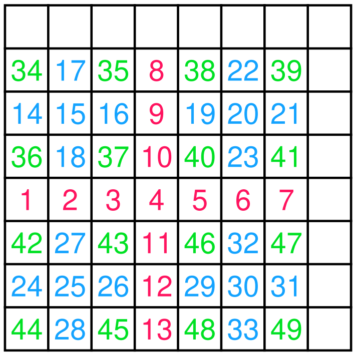
	17		8		22		
14	15	16	9	19	20	21	
	18		10		23		
1	2	3	4	5	6	7	
	27		11		32		
24	25	26	12	29	30	31	
	28		13		33		

$(2^3-1)^2$ 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

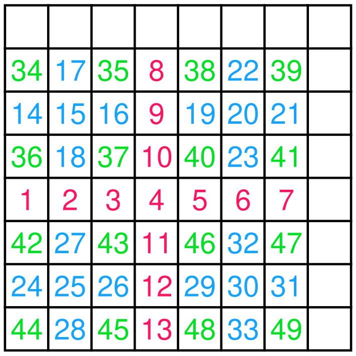
NCA = nearest common ancestor

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 $\leq 16/3$

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

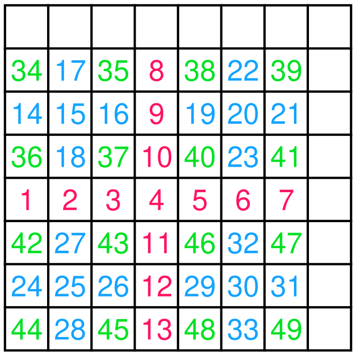
$4^0 \cdot (2 \cdot (2^{n-i}-1)-1)$ have NCA at level n

$4^1 \cdot (2 \cdot (2^{n-i-1}-1)-1)$ 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 $\leq 16/3$

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

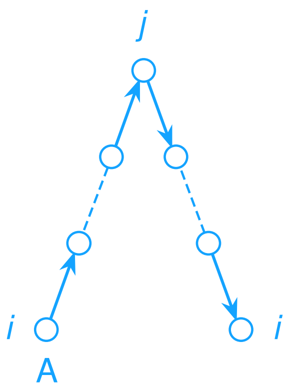
Theorem: average number of nodes visited by `EQUAL_DIAGONAL_NEIGHBOR` is $\leq 16/3$

Proof:

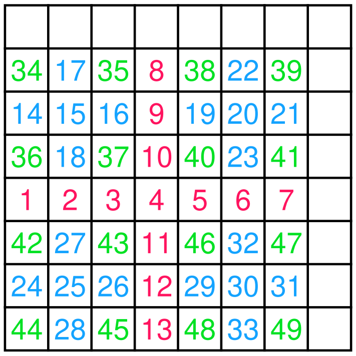
- Let node A be at level i (i.e., a $2^i \times 2^i$ block)
- There are $(2^{n-i}-1)^2$ possible positions for node A such that an equal size neighbor exists in a given corner direction

  $4^0 \cdot (2 \cdot (2^{n-i}-1)-1)$ 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 $\boxed{i}$, direction D, and the NCA at level $\boxed{j}$, $\boxed{2 \cdot (j-i)}$ nodes are visited in locating an equal-sized neighbor at level $\boxed{i}$



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 $\leq 16/3$

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 $n-1$

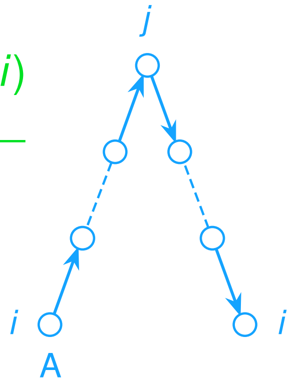
  …

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

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

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

nodes are visited on the average $\leq 16/3$



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

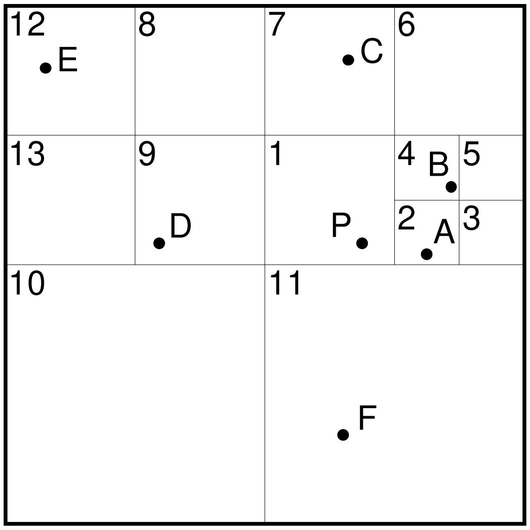
	Flood	Topo	Land	Pebble	Average	Predicted
lateral neighbor	2.01	2.00	2.00	1.99	2.00	1.98
diagonal neighbor	2.69	2.67	2.66	2.65	2.67	2.62

- 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

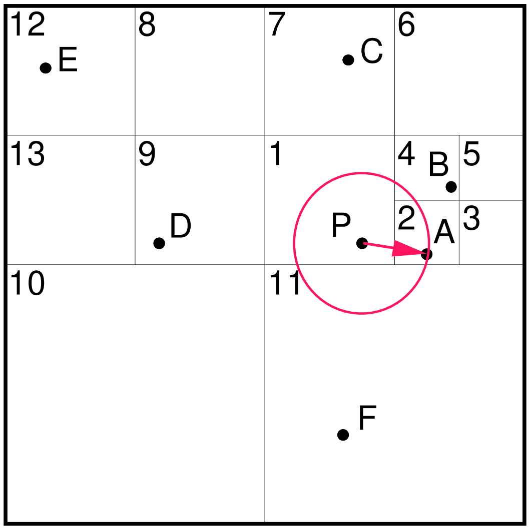
- Ex: find the nearest object to P



- 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

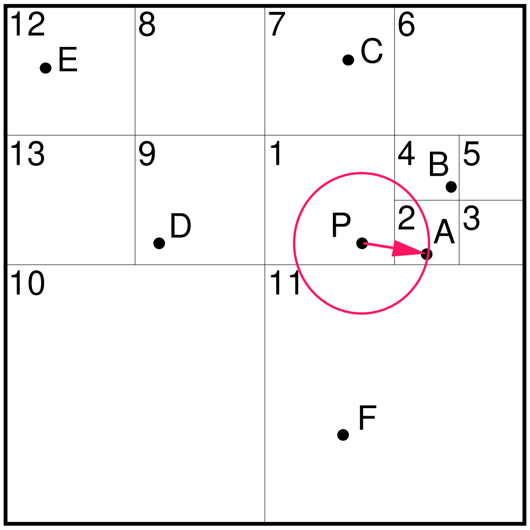


- Assume PR quadtree for points (i.e., at most one point per block)
- Search neighbors of block 1 in counterclockwise order
- Points are sorted with respect to the space they occupy which enables pruning the search space
- Algorithm:

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

FINDING THE NEAREST OBJECT

- Ex: find the nearest object to P

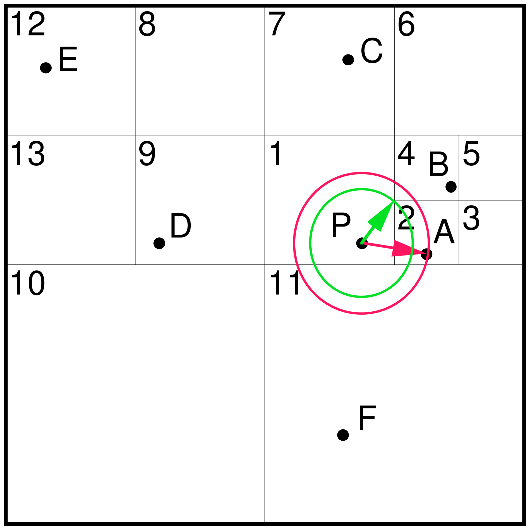


- Assume PR quadtree for points (i.e., at most one point per block)
- Search neighbors of block 1 in counterclockwise order
- Points are sorted with respect to the space they occupy which enables pruning the search space
- Algorithm:

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

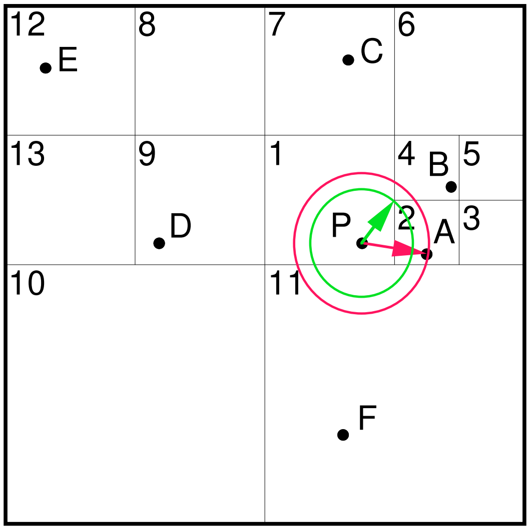


- Assume PR quadtree for points (i.e., at most one point per block)
- Search neighbors of block 1 in counterclockwise order
- Points are sorted with respect to the space they occupy which enables pruning the search space
- Algorithm:

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

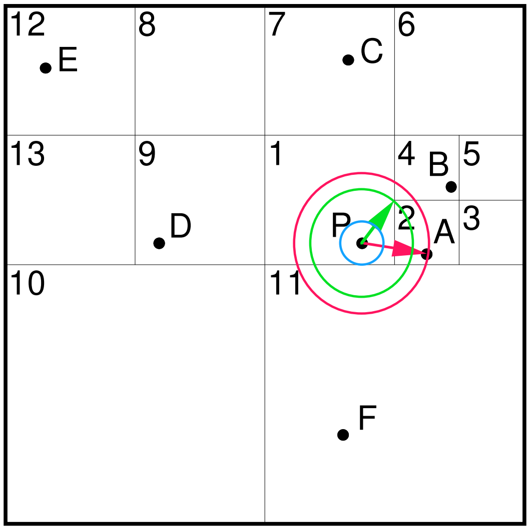


- Assume PR quadtree for points (i.e., at most one point per block)
- Search neighbors of block 1 in counterclockwise order
- Points are sorted with respect to the space they occupy which enables pruning the search space
- Algorithm:

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

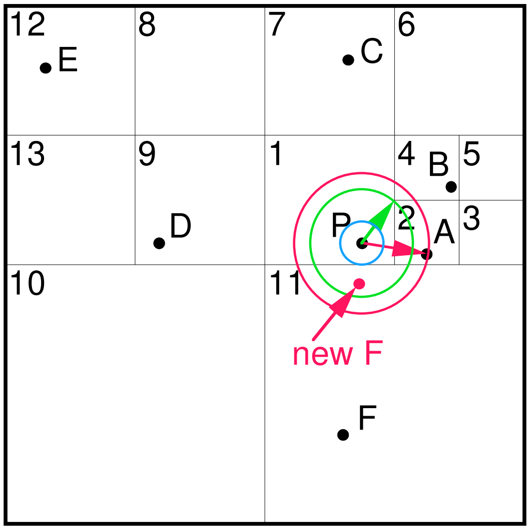


- Assume PR quadtree for points (i.e., at most one point per block)
- Search neighbors of block 1 in counterclockwise order
- Points are sorted with respect to the space they occupy which enables pruning the search space
- Algorithm:

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

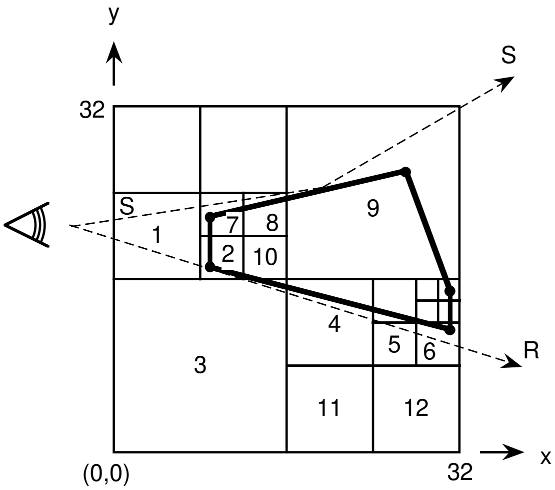
- Ex: find the nearest object to P



- 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



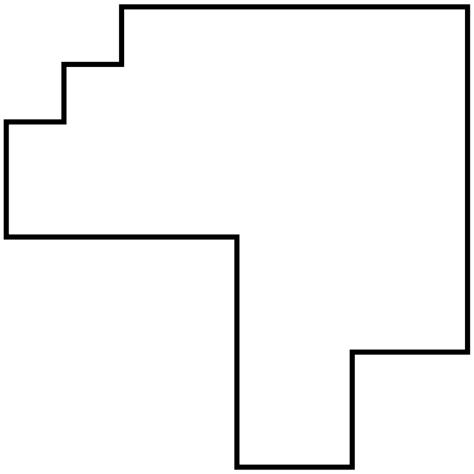
DATA STRUCTURE CONVERSION

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

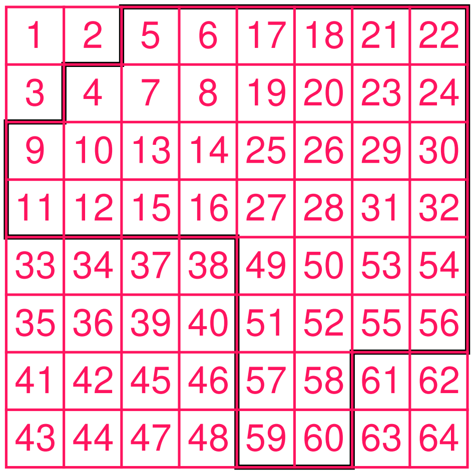
QUADTREES FROM BINARY ARRAYS

- Traverse pixels in Morton order
- No need to merge four leaf nodes of the same color as only create maximal leaf nodes
- Execution time proportional to number of pixels
- Ex:



QUADTREES FROM BINARY ARRAYS

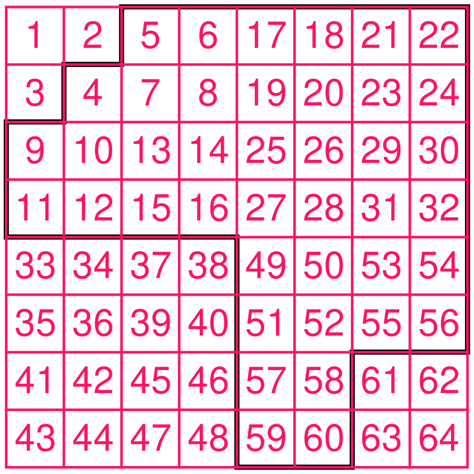
- Traverse pixels in Morton order
- No need to merge four leaf nodes of the same color as only create maximal leaf nodes
- Execution time proportional to number of pixels
- Ex:



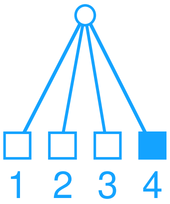
1	2	5	6	17	18	21	22
3	4	7	8	19	20	23	24
9	10	13	14	25	26	29	30
11	12	15	16	27	28	31	32
33	34	37	38	49	50	53	54
35	36	39	40	51	52	55	56
41	42	45	46	57	58	61	62
43	44	47	48	59	60	63	64

QUADTREES FROM BINARY ARRAYS

- Traverse pixels in Morton order
- No need to merge four leaf nodes of the same color as only create maximal leaf nodes
- Execution time proportional to number of pixels
- Ex:

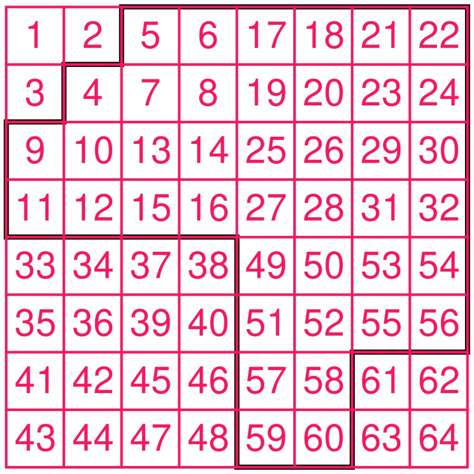


1	2	5	6	17	18	21	22
3	4	7	8	19	20	23	24
9	10	13	14	25	26	29	30
11	12	15	16	27	28	31	32
33	34	37	38	49	50	53	54
35	36	39	40	51	52	55	56
41	42	45	46	57	58	61	62
43	44	47	48	59	60	63	64

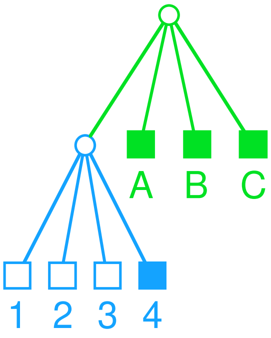


QUADTREES FROM BINARY ARRAYS

- Traverse pixels in Morton order
- No need to merge four leaf nodes of the same color as only create maximal leaf nodes
- Execution time proportional to number of pixels
- Ex:

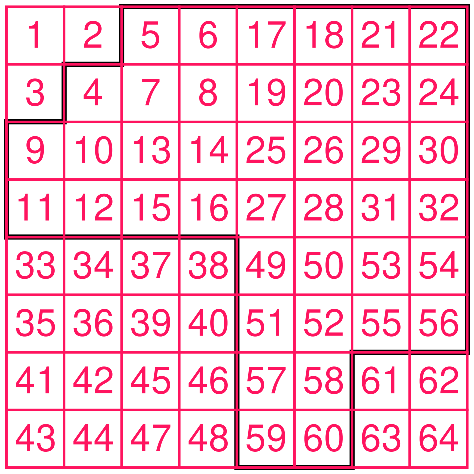


1	2	5	6	17	18	21	22
3	4	7	8	19	20	23	24
9	10	13	14	25	26	29	30
11	12	15	16	27	28	31	32
33	34	37	38	49	50	53	54
35	36	39	40	51	52	55	56
41	42	45	46	57	58	61	62
43	44	47	48	59	60	63	64

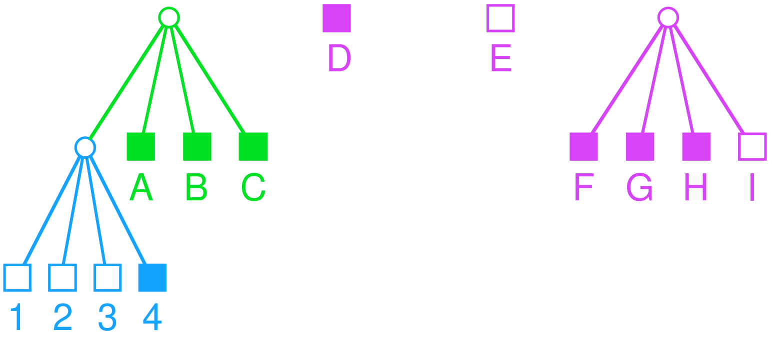


QUADTREES FROM BINARY ARRAYS

- Traverse pixels in Morton order
- No need to merge four leaf nodes of the same color as only create maximal leaf nodes
- Execution time proportional to number of pixels
- Ex:

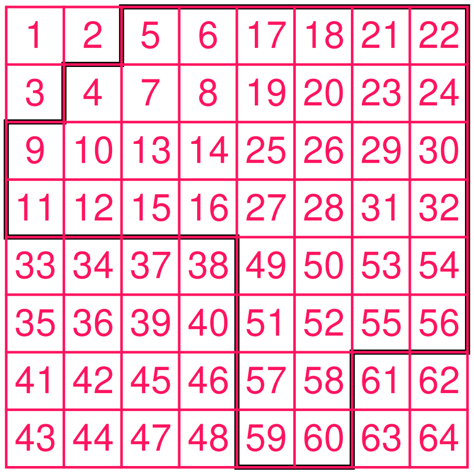


1	2	5	6	17	18	21	22
3	4	7	8	19	20	23	24
9	10	13	14	25	26	29	30
11	12	15	16	27	28	31	32
33	34	37	38	49	50	53	54
35	36	39	40	51	52	55	56
41	42	45	46	57	58	61	62
43	44	47	48	59	60	63	64

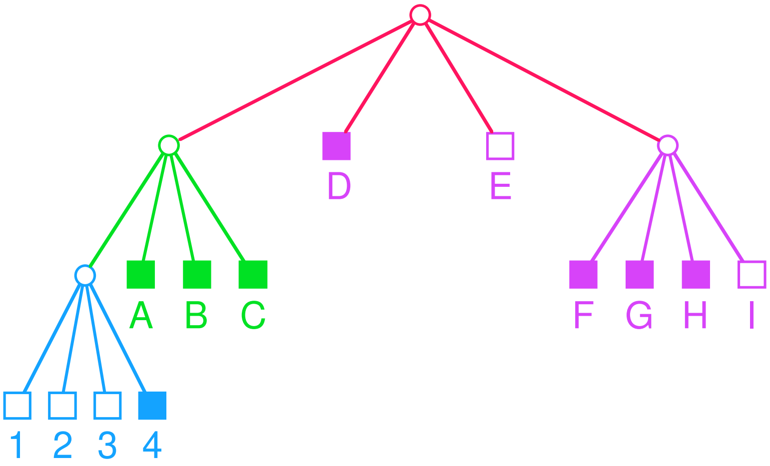


QUADTREES FROM BINARY ARRAYS

- Traverse pixels in Morton order
- No need to merge four leaf nodes of the same color as only create maximal leaf nodes
- Execution time proportional to number of pixels
- Ex:

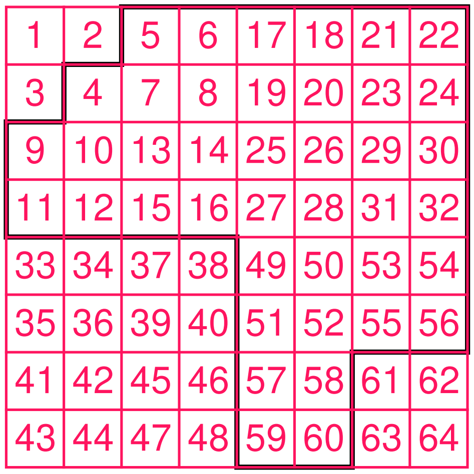


1	2	5	6	17	18	21	22
3	4	7	8	19	20	23	24
9	10	13	14	25	26	29	30
11	12	15	16	27	28	31	32
33	34	37	38	49	50	53	54
35	36	39	40	51	52	55	56
41	42	45	46	57	58	61	62
43	44	47	48	59	60	63	64

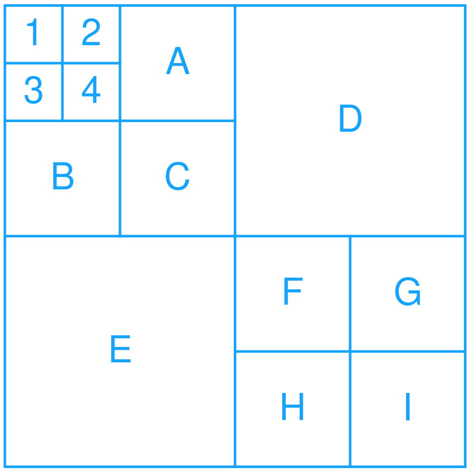


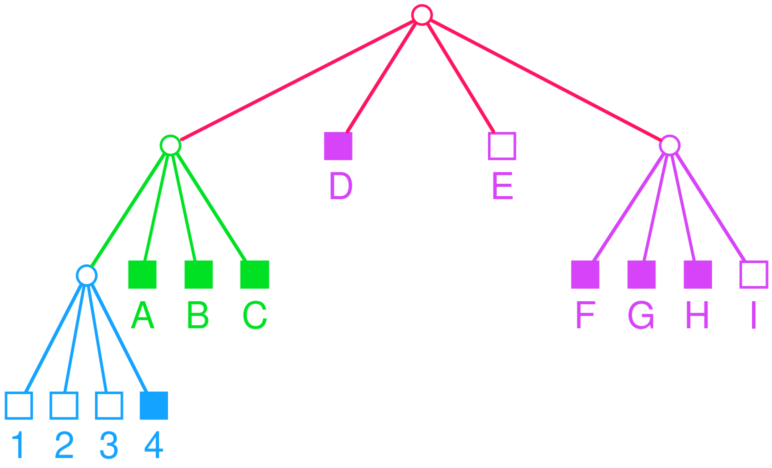
QUADTREES FROM BINARY ARRAYS

- Traverse pixels in Morton order
- No need to merge four leaf nodes of the same color as only create maximal leaf nodes
- Execution time proportional to number of pixels
- Ex:



1	2	5	6	17	18	21	22
3	4	7	8	19	20	23	24
9	10	13	14	25	26	29	30
11	12	15	16	27	28	31	32
33	34	37	38	49	50	53	54
35	36	39	40	51	52	55	56
41	42	45	46	57	58	61	62
43	44	47	48	59	60	63	64



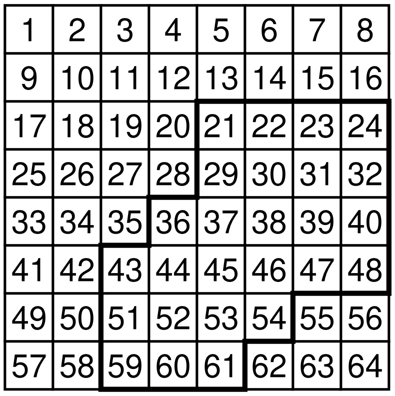


BUILDING QUADTREES FROM RASTERS

- Using a variant of EQUAL_LATERAL_NEIGHBOR to find neighbors in the eastern and southern directions and adding them if they are not present

Algorithm: process array row by row

- Odd row — add nodes to tree
- Even row — add nodes to tree; attempt to merge
- Execution time is O(number of pixels)
- Ex:



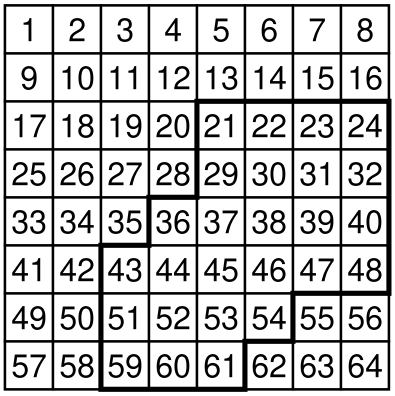
1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16
17	18	19	20	21	22	23	24
25	26	27	28	29	30	31	32
33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48
49	50	51	52	53	54	55	56
57	58	59	60	61	62	63	64

BUILDING QUADTREES FROM RASTERS

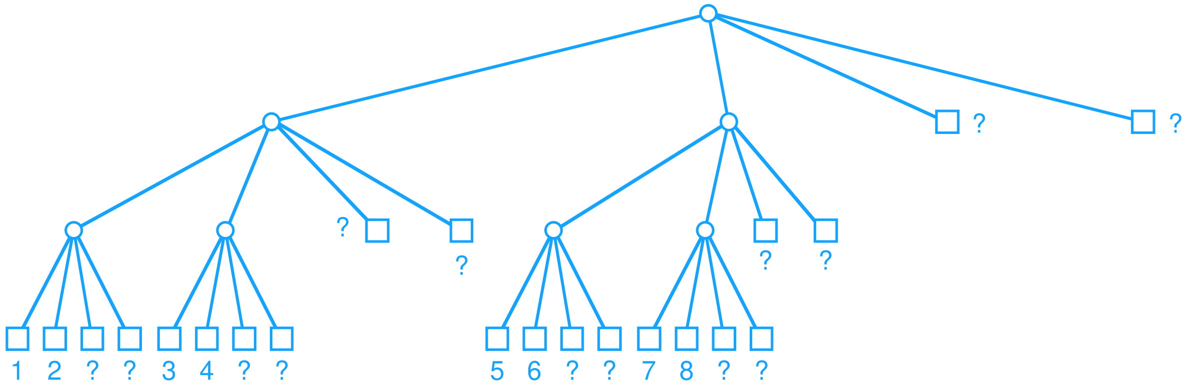
- Using a variant of EQUAL_LATERAL_NEIGHBOR to find neighbors in the eastern and southern directions and adding them if they are not present

Algorithm: process array row by row

- Odd row — add nodes to tree
- Even row — add nodes to tree; attempt to merge
- Execution time is O(number of pixels)
- Ex:



1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16
17	18	19	20	21	22	23	24
25	26	27	28	29	30	31	32
33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48
49	50	51	52	53	54	55	56
57	58	59	60	61	62	63	64



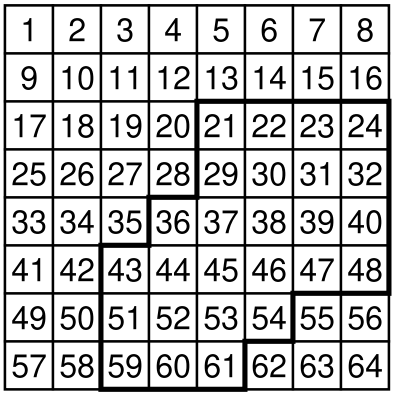
- Result of processing the first row

BUILDING QUADTREES FROM RASTERS

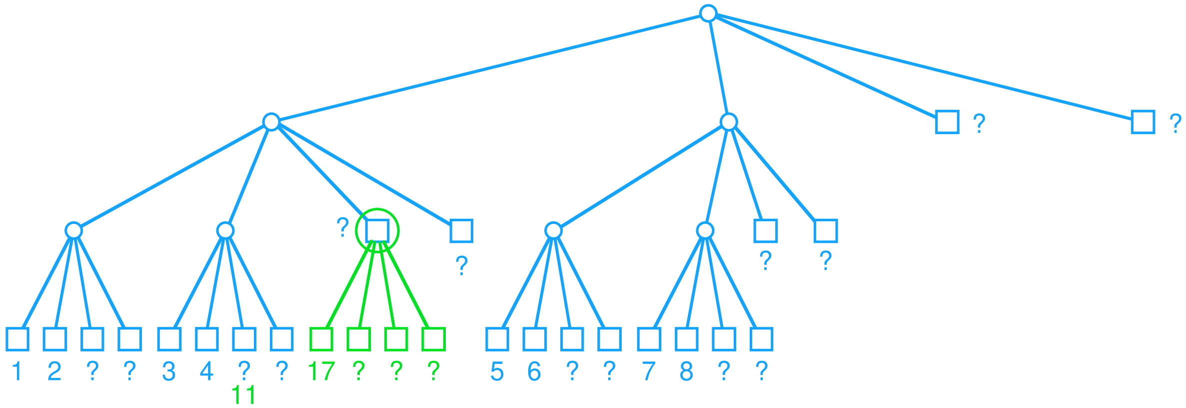
- Using a variant of EQUAL_LATERAL_NEIGHBOR to find neighbors in the eastern and southern directions and adding them if they are not present

Algorithm: process array row by row

- Odd row — add nodes to tree
- Even row — add nodes to tree; attempt to merge
- Execution time is *O*(number of pixels)
- Ex:



1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16
17	18	19	20	21	22	23	24
25	26	27	28	29	30	31	32
33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48
49	50	51	52	53	54	55	56
57	58	59	60	61	62	63	64



- Result of processing the first row
- Result just prior to merging nodes corresponding to pixels 1, 2, 9, and 10

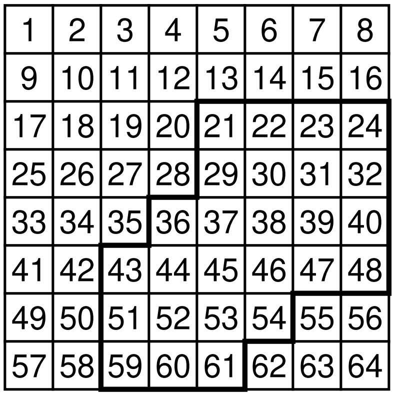
4 3 2 1
v g z b

BUILDING QUADTREES FROM RASTERS

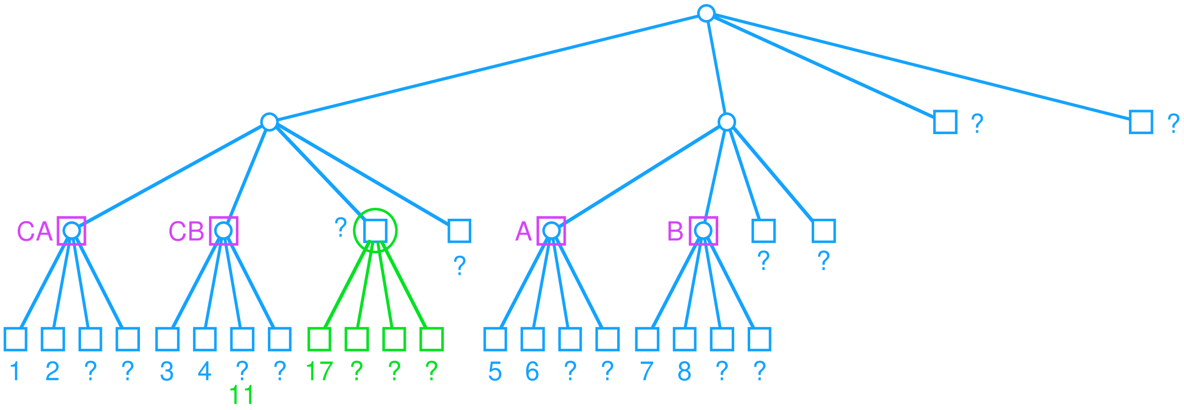
- Using a variant of EQUAL_LATERAL_NEIGHBOR to find neighbors in the eastern and southern directions and adding them if they are not present

Algorithm: process array row by row

- Odd row — add nodes to tree
- Even row — add nodes to tree; attempt to merge
- Execution time is *O*(number of pixels)
- Ex:



1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16
17	18	19	20	21	22	23	24
25	26	27	28	29	30	31	32
33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48
49	50	51	52	53	54	55	56
57	58	59	60	61	62	63	64



- Result of processing the first row
- Result just prior to merging nodes corresponding to pixels 1, 2, 9, and 10
- Result of processing the second row

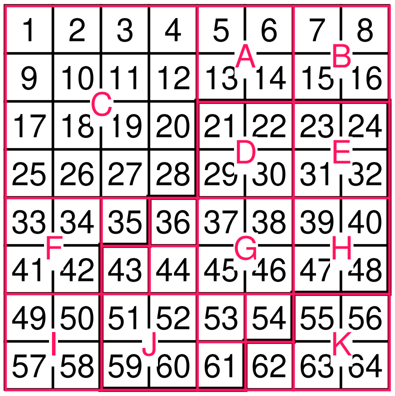
5 4 3 2 1
r v g z b

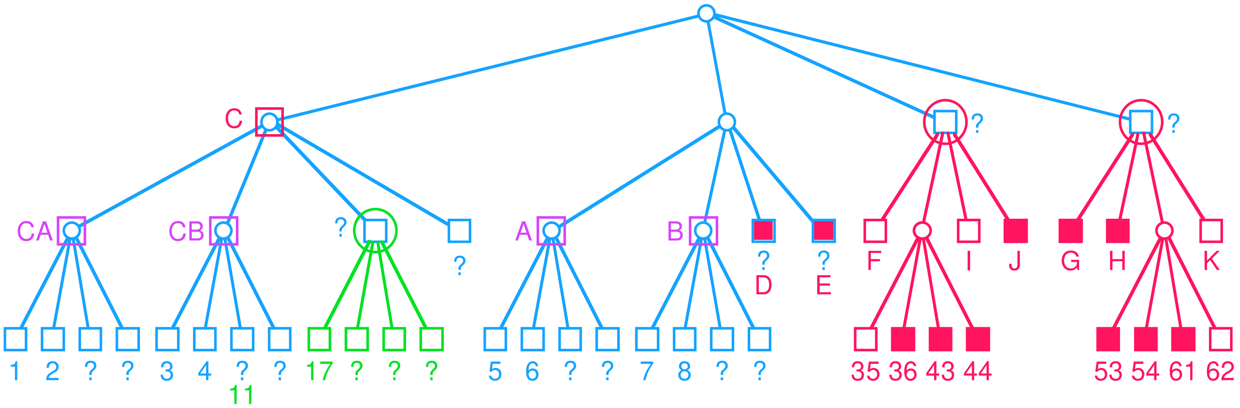
BUILDING QUADTREES FROM RASTERS

- Using a variant of EQUAL_LATERAL_NEIGHBOR to find neighbors in the eastern and southern directions and adding them if they are not present

Algorithm: process array row by row

- Odd row — add nodes to tree
- Even row — add nodes to tree; attempt to merge
- Execution time is *O*(number of pixels)
- Ex:

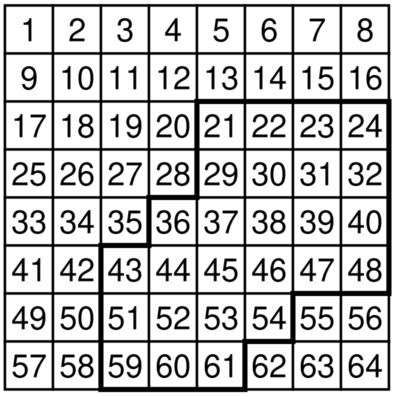




- Result of processing the first row
- Result just prior to merging nodes corresponding to pixels 1, 2, 9, and 10
- Result of processing the second row
- Final quadtree

INITIAL RASTER TO QUADTREE BUILDING SEQUENCE

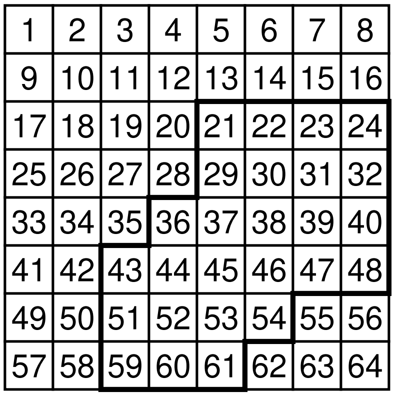
- Ex: build quadtree for first four pixels in the first row



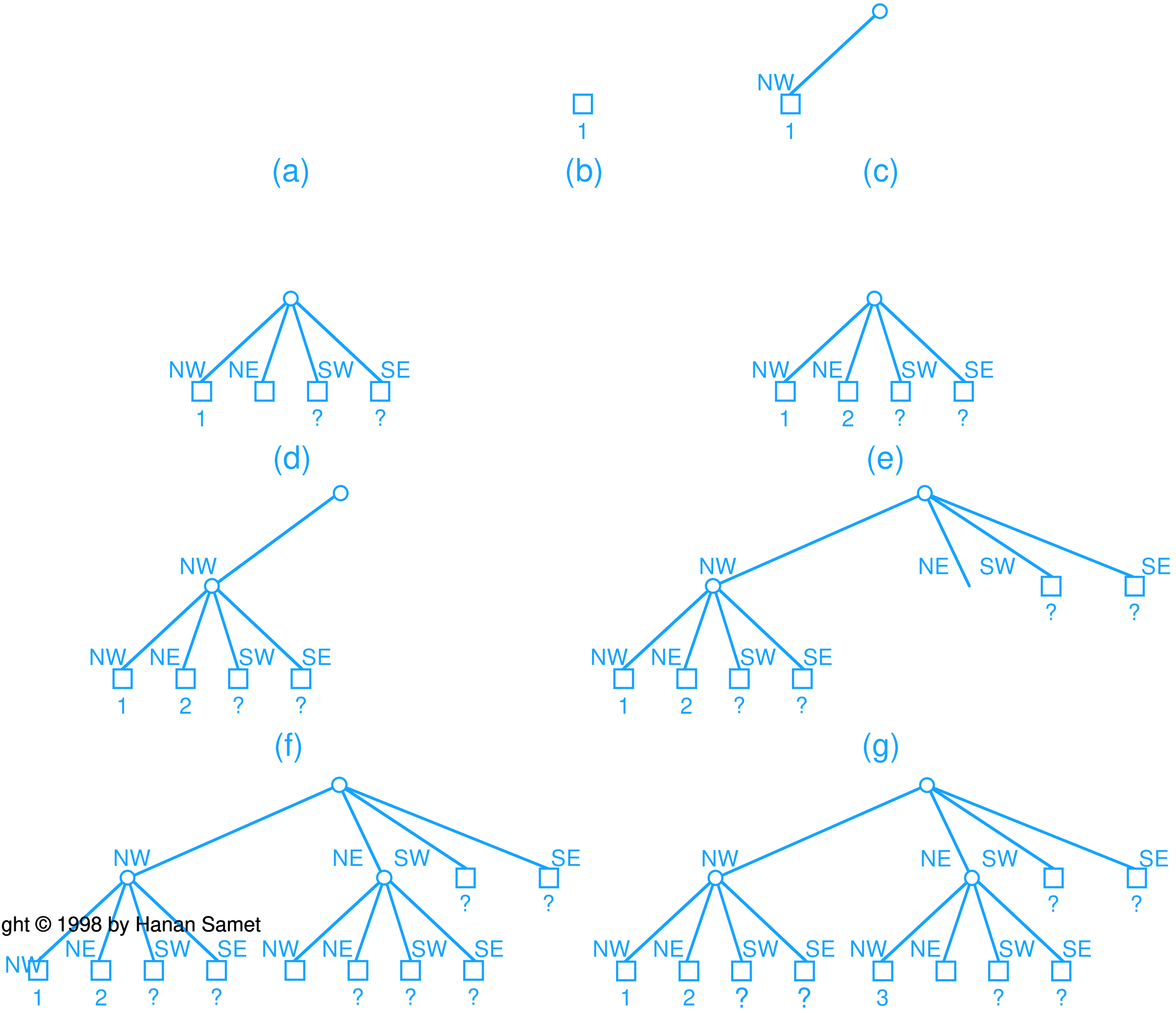
1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16
17	18	19	20	21	22	23	24
25	26	27	28	29	30	31	32
33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48
49	50	51	52	53	54	55	56
57	58	59	60	61	62	63	64

INITIAL RASTER TO QUADTREE BUILDING SEQUENCE

- Ex: build quadtree for first four pixels in the first row
- Use a variant of EQUAL_LATERAL_NEIGHBOR to find neighbors in the eastern direction and add them if they are not present



1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16
17	18	19	20	21	22	23	24
25	26	27	28	29	30	31	32
33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48
49	50	51	52	53	54	55	56
57	58	59	60	61	62	63	64



(a) (b) (c)

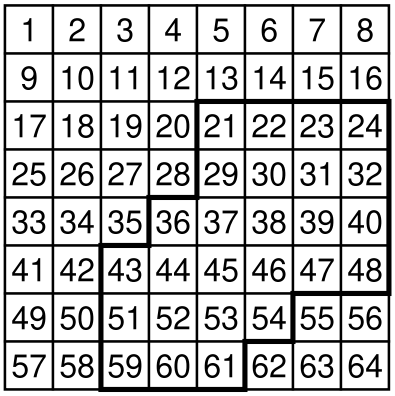
(d) (e)

(f) (g)

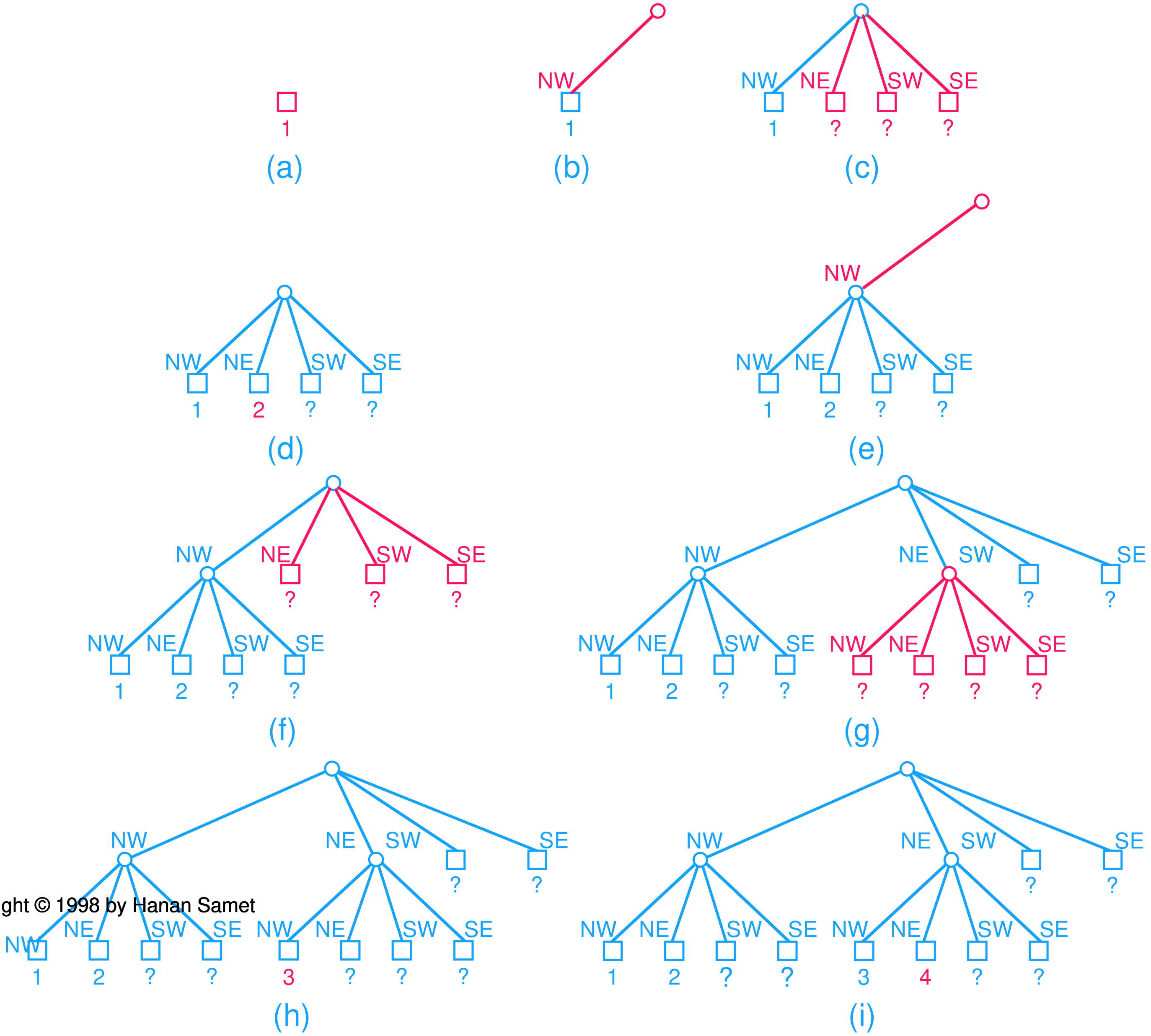
(h) (i)

INITIAL RASTER TO QUADTREE BUILDING SEQUENCE

- Ex: build quadtree for first four pixels in the first row
- Use a variant of EQUAL_LATERAL_NEIGHBOR to find neighbors in the eastern direction and add them if they are not present



1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16
17	18	19	20	21	22	23	24
25	26	27	28	29	30	31	32
33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48
49	50	51	52	53	54	55	56
57	58	59	60	61	62	63	64



(a) (b) (c) (d) (e) (f) (g) (h) (i)

OPTIMAL QUADTREE BUILDING

I = cost of a block (i.e., node) insertion operation
c = cost of examining a pixel
$2^n \times 2^n$ image
N = number of blocks in the output quadtree

Naive algorithm:

- Examine each pixel and insert it into the quadtree
- Cost = $2^{2n} \cdot (c+I)$

Optimal algorithm:

- Examine each pixel but only insert the largest block for which the current pixel is the first (i.e., upper leftmost) pixel
- Avoids the need for merging
- Cost = $c \cdot 2^{2n} + I \cdot N$
- Speedup is obvious because $c << I$ and $N << 2^{2n}$

Map Name	Num of Blocks	Num of inserts		Time (secs)	
		Naive	Optimal	Naive	Optimal
Floodplain	5266	180000	2352	413.2	13.8
Topography	24859	180000	12400	429.8	51.2
Landuse	28447	180000	14675	436.7	56.9
Center	4687	262144	2121	603.8	16.1
Pebble	44950	262144	20770	630.1	111.0
Stone	31969	262144	14612	629.5	70.2

Storage requirements:

- Higher for optimal algorithm since it must keep track of active blocks
- Size of intermediate quadtree is smaller as there is no merging

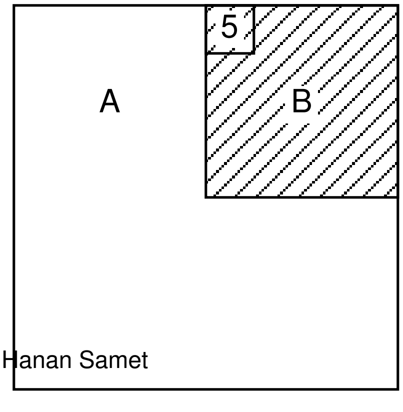
OPTIMAL ALGORITHM

- When building a quadtree there is a processed and unprocessed part of the image; blocks covering both parts have been assigned

Def: a block is *active* if at least one pixel, but not all of the pixels, covered by the block has been processed and it differs in color from a block that contains it

Ex: blocks A and B are both active after processing pixel 5

Algorithm: traverse image in raster scan order; for each pixel:



OPTIMAL ALGORITHM

- When building a quadtree there is a processed and unprocessed part of the image; blocks covering both parts have been assigned

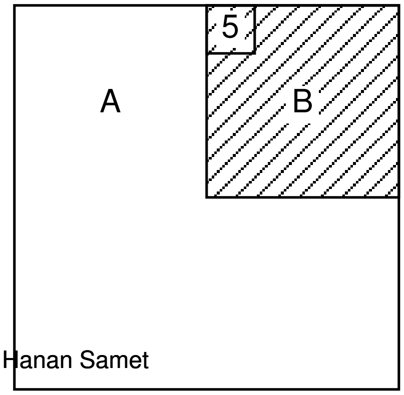
Def: a block is *active* if at least one pixel, but not all of the pixels, covered by the block has been processed and it differs in color from a block that contains it

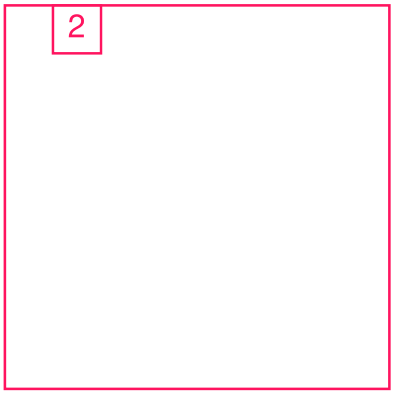
Ex: blocks A and B are both active after processing pixel 5

Algorithm: traverse image in raster scan order; for each pixel:

1. if the pixel is the same color as the appropriate active block, then do nothing

   Ex: pixel 2





OPTIMAL ALGORITHM

- When building a quadtree there is a processed and unprocessed part of the image; blocks covering both parts have been assigned

Def: a block is *active* if at least one pixel, but not all of the pixels, covered by the block has been processed and it differs in color from a block that contains it

Ex: blocks A and B are both active after processing pixel 5

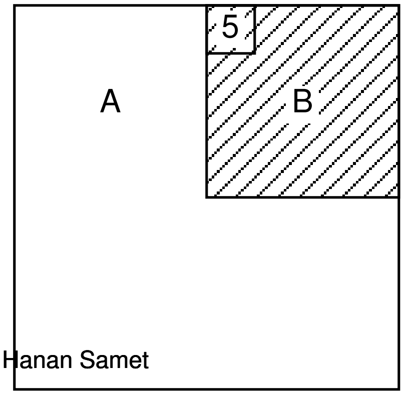
Algorithm: traverse image in raster scan order; for each pixel:

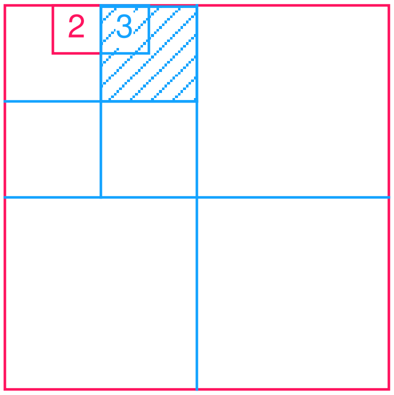
1. if the pixel is the same color as the appropriate active block, then do nothing

   Ex: pixel 2

2. else insert the largest possible block for which this is the first (i.e., upper leftmost) pixel and (if it is not a 1×1 block) add it to the list of active blocks

   Ex: pixel 3





OPTIMAL ALGORITHM

- When building a quadtree there is a processed and unprocessed part of the image; blocks covering both parts have been assigned

Def: a block is *active* if at least one pixel, but not all of the pixels, covered by the block has been processed and it differs in color from a block that contains it

Ex: blocks A and B are both active after processing pixel 5

Algorithm: traverse image in raster scan order; for each pixel:

1. if the pixel is the same color as the appropriate active block, then do nothing

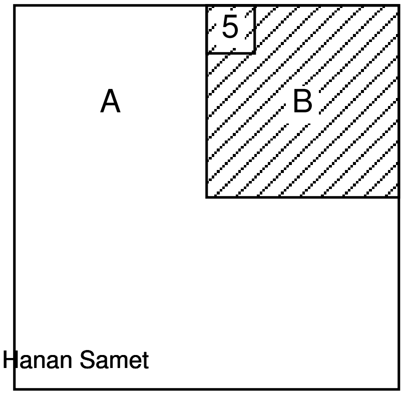
   Ex: pixel 2

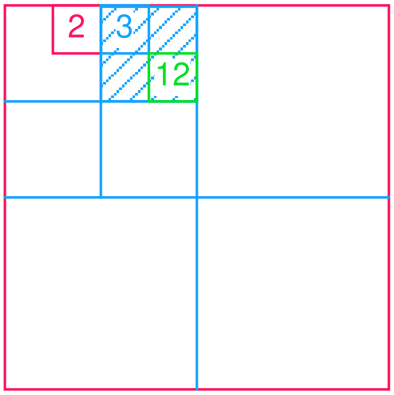
2. else insert the largest possible block for which this is the first (i.e., upper leftmost) pixel and (if it is not a 1×1 block) add it to the list of active blocks

   Ex: pixel 3

3. remove any active blocks for which this is the last (i.e., lower rightmost) pixel

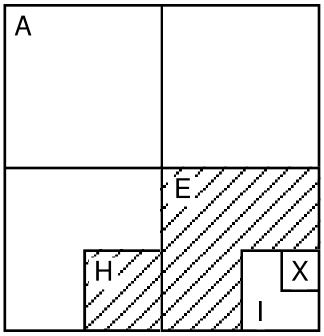
   Ex: pixel 12



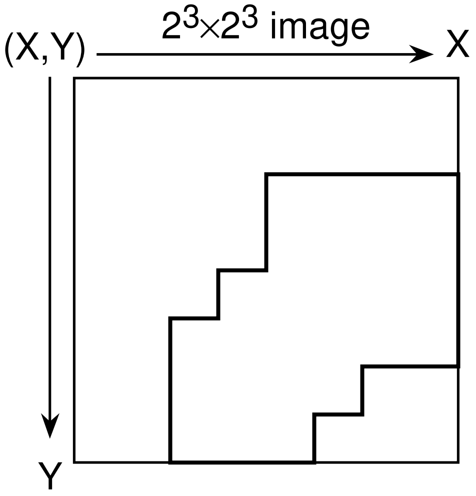


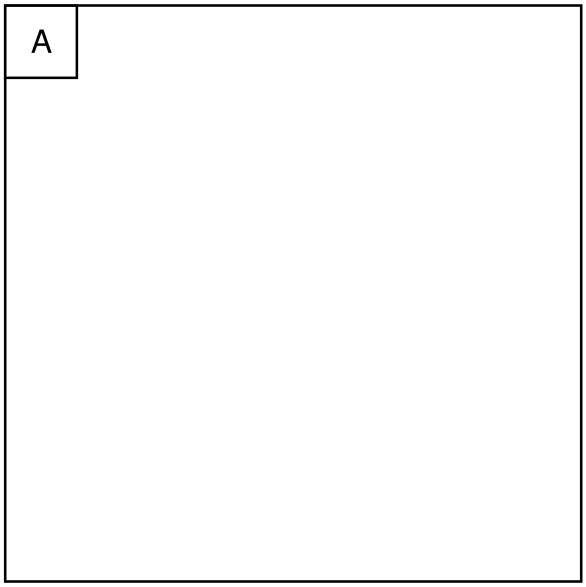
IMPLEMENTATION

- The algorithm must keep track of all the active blocks
- For a $2^n \times 2^n$ image the number of active blocks is $\leq 2^{n-1}$
- Use a data structure called TABLE to keep a list of the active blocks organized by levels
  1. one row for each level in quadtree (except for level 0)
  2. row i has 2^{n-i} entries
  3. for a pixel in column j of image, the color of the active block (if there is one) at level i is in entry $\lfloor j / 2^i \rfloor$ of row i
- Problem: must search for the smallest active block covering a pixel
  1. start at the lowest level and stop at first non-empty entry
  2. slow — could require n steps
- Solution: use an access array LIST
  1. LIST[$\lfloor j / 2 \rfloor$] indicates the row of TABLE corresponding to the smallest active block that contains the pixel at column j
  2. initially, each entry of LIST points to n — i.e., the row of TABLE corresponding to the root of a $2^n \times 2^n$ image
  3. as active blocks are inserted and completed (i.e., deleted from the active block data structure), TABLE and LIST are updated
- Ex: blocks A, E, H, and I are all active after processing pixel X, while blocks A, E, and I all cover this pixel



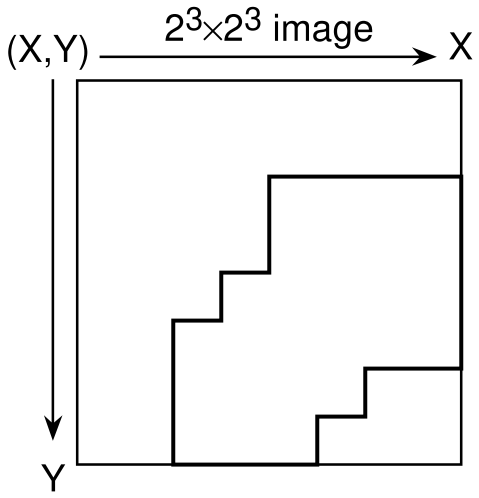
EXAMPLE OF OPTIMAL QUADTREE BUILDING

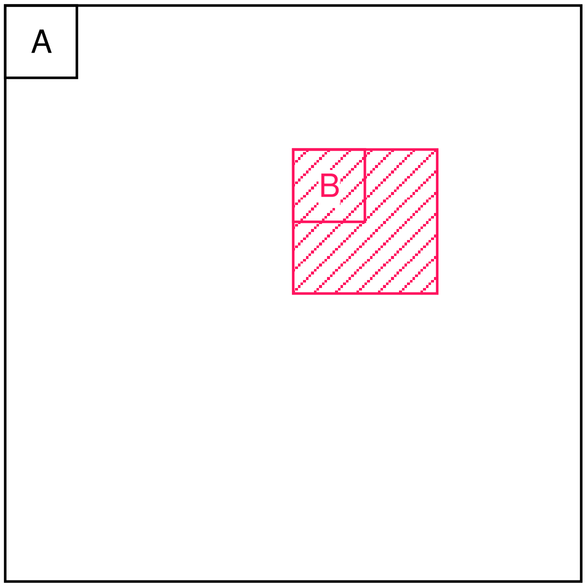




Pixel	Action	Size	Active Blocks
(0,0)	insert WHITE block A	8×8	3:A

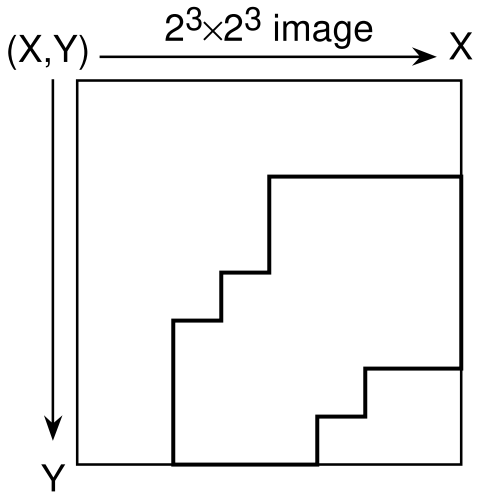
EXAMPLE OF OPTIMAL QUADTREE BUILDING

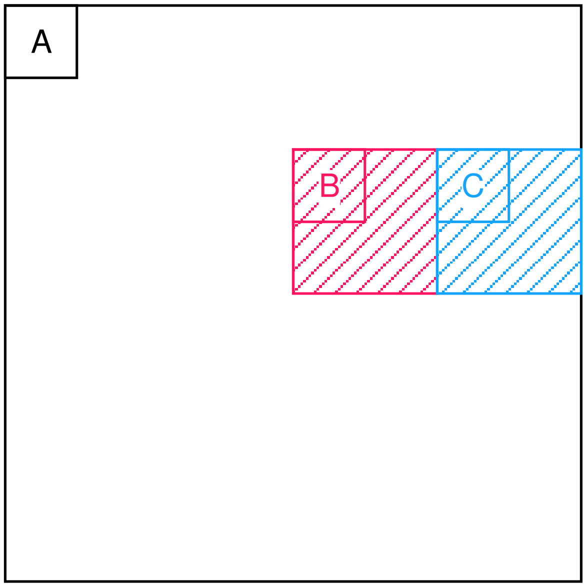




Pixel	Action	Size	Active Blocks
(0,0)	insert WHITE block A	8×8	3:A
(2,4)	insert BLACK block B	2×2	3:A 1:B

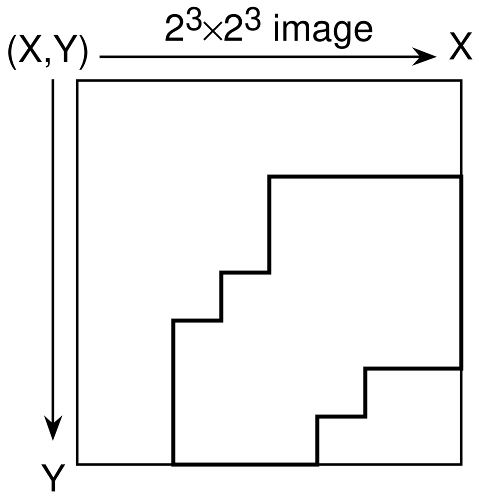
EXAMPLE OF OPTIMAL QUADTREE BUILDING

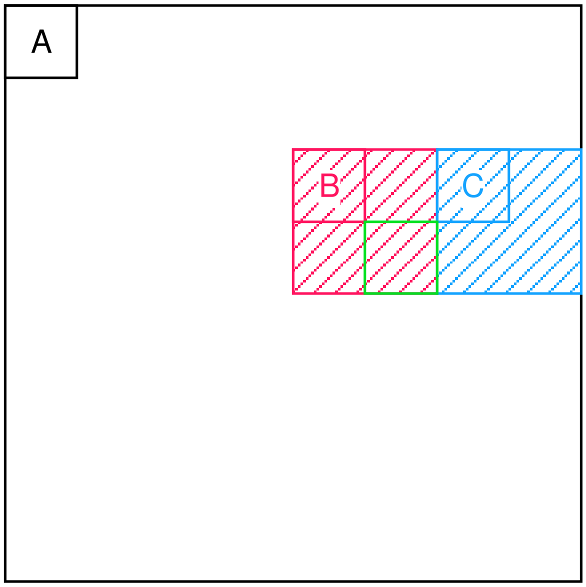




Pixel	Action	Size	Active Blocks
(0,0)	insert WHITE block A	8×8	3:A
(2,4)	insert BLACK block B	2×2	3:A 1:B
(2,6)	insert BLACK block C	2×2	3:A 1:B,C

EXAMPLE OF OPTIMAL QUADTREE BUILDING

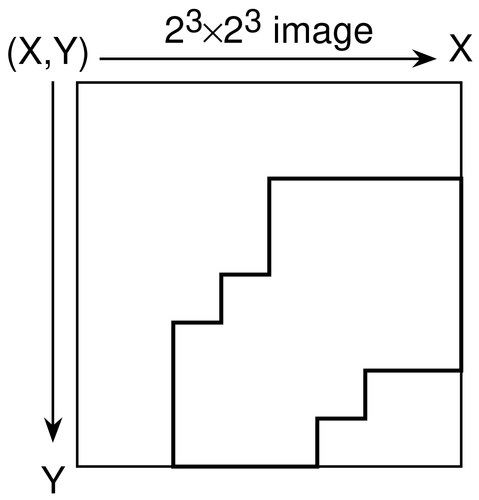


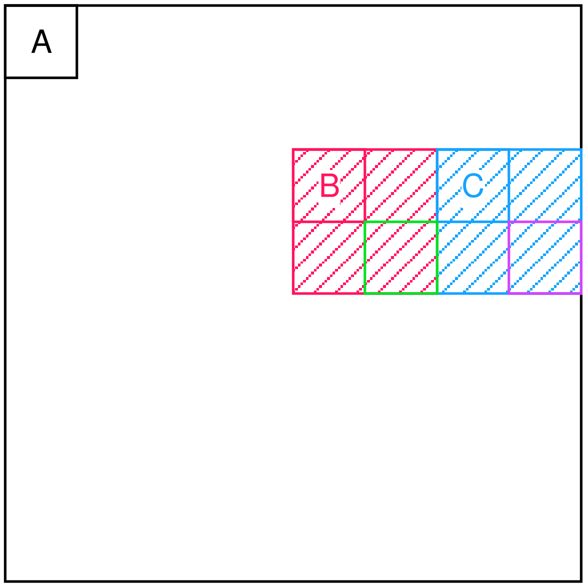


Pixel	Action	Size	Active Blocks
(0,0)	insert WHITE block A	8×8	3:A
(2,4)	insert BLACK block B	2×2	3:A 1:B
(2,6)	insert BLACK block C	2×2	3:A 1:B,C
(3,5)	remove B from active		3:A 1:C

5 4 3 2 1
v g z r b

EXAMPLE OF OPTIMAL QUADTREE BUILDING



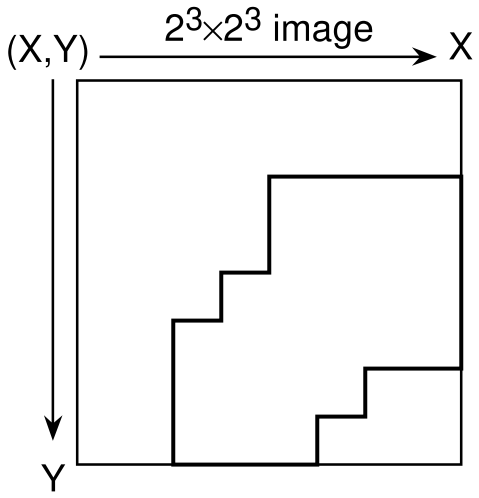


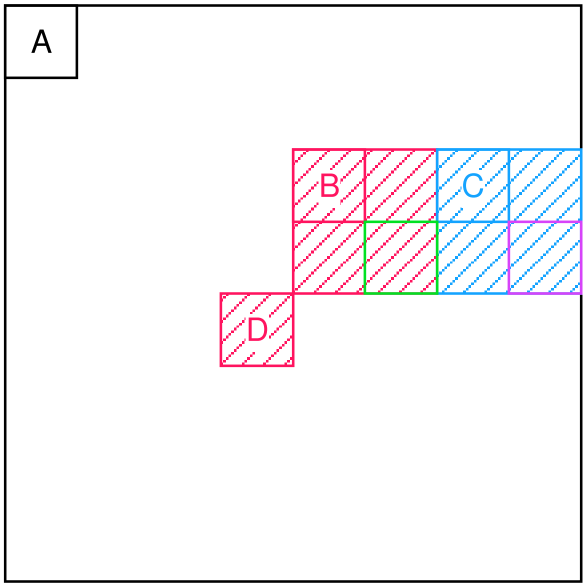
Pixel	Action	Size	Active Blocks
(0,0)	insert WHITE block A	8×8	3:A
(2,4)	insert BLACK block B	2×2	3:A 1:B
(2,6)	insert BLACK block C	2×2	3:A 1:B,C
(3,5)	remove B from active		3:A 1:C
(3,7)	remove C from active		3:A

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

qb7

EXAMPLE OF OPTIMAL QUADTREE BUILDING

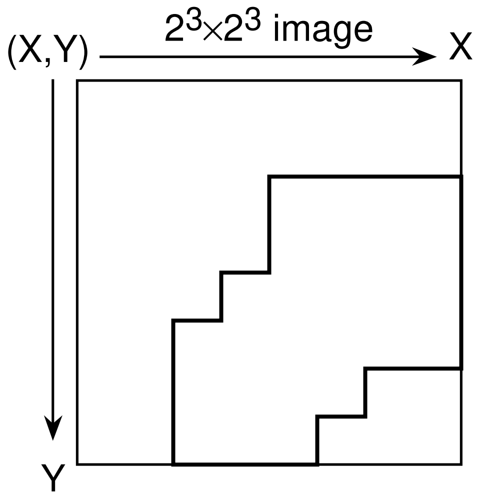


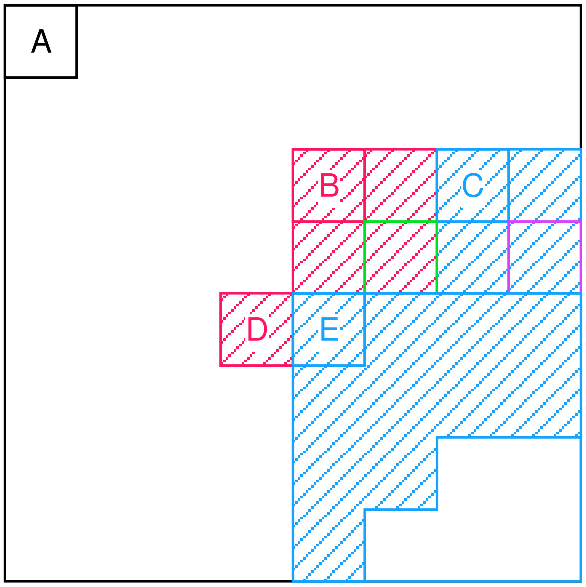


Pixel	Action	Size	Active Blocks
(0,0)	insert WHITE block A	8×8	3:A
(2,4)	insert BLACK block B	2×2	3:A 1:B
(2,6)	insert BLACK block C	2×2	3:A 1:B,C
(3,5)	remove B from active		3:A 1:C
(3,7)	remove C from active		3:A
(4,3)	insert BLACK block D		3:A

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

EXAMPLE OF OPTIMAL QUADTREE BUILDING

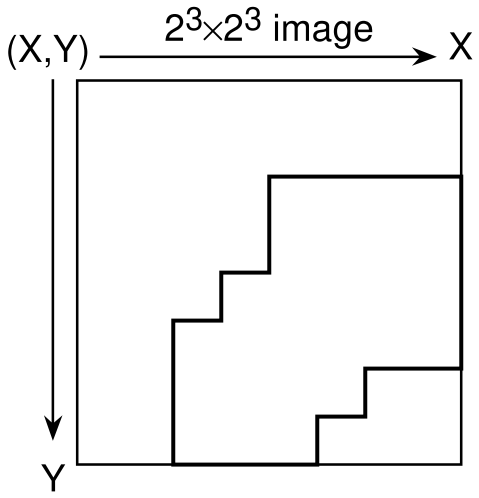


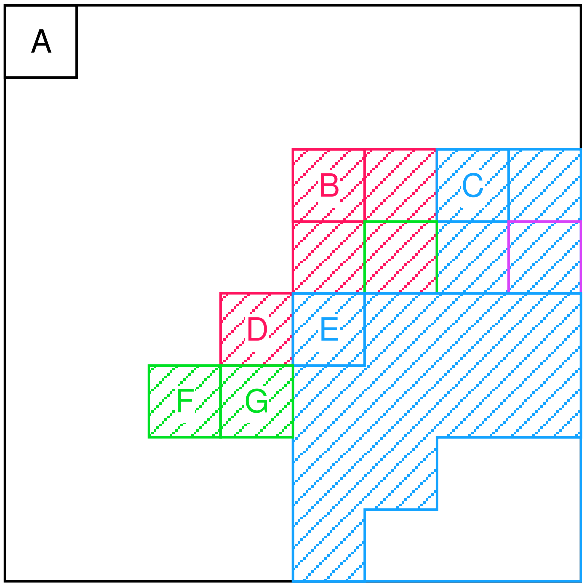


Pixel	Action	Size	Active Blocks
(0,0)	insert WHITE block A	8×8	3:A
(2,4)	insert BLACK block B	2×2	3:A 1:B
(2,6)	insert BLACK block C	2×2	3:A 1:B,C
(3,5)	remove B from active		3:A 1:C
(3,7)	remove C from active		3:A
(4,3)	insert BLACK block D		3:A
(4,4)	insert BLACK block E	4×4	3:A 2:E

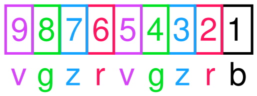
8 7 6 5 4 3 2 1
g z r v g z r b

EXAMPLE OF OPTIMAL QUADTREE BUILDING

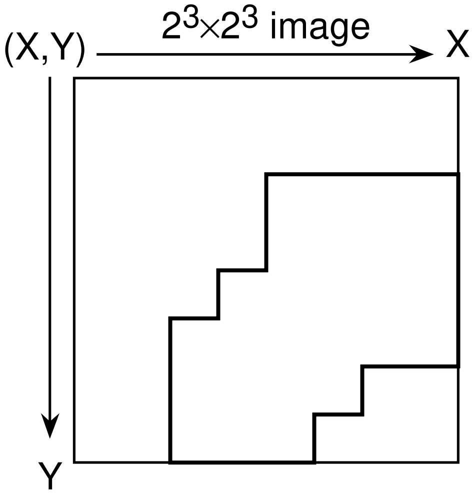


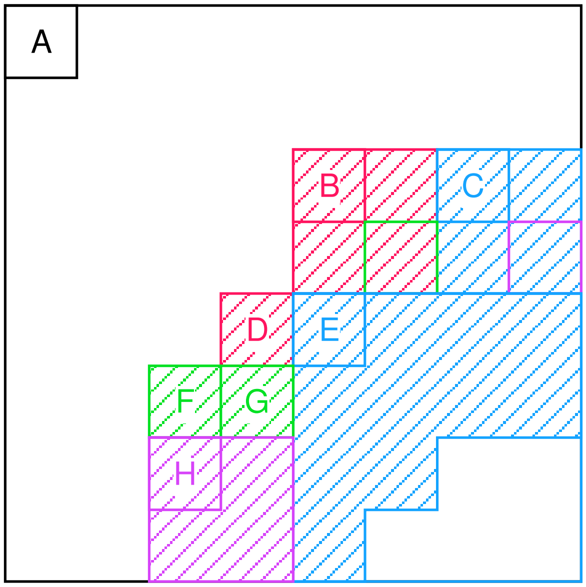


Pixel	Action	Size	Active Blocks
(0,0)	insert WHITE block A	8×8	3:A
(2,4)	insert BLACK block B	2×2	3:A 1:B
(2,6)	insert BLACK block C	2×2	3:A 1:B,C
(3,5)	remove B from active		3:A 1:C
(3,7)	remove C from active		3:A
(4,3)	insert BLACK block D		3:A
(4,4)	insert BLACK block E	4×4	3:A 2:E
(5,2) (5,3)	insert BLACK blocks F, G		3:A 2:E



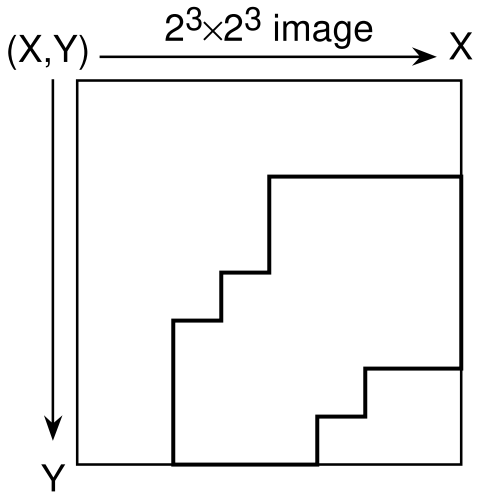
EXAMPLE OF OPTIMAL QUADTREE BUILDING

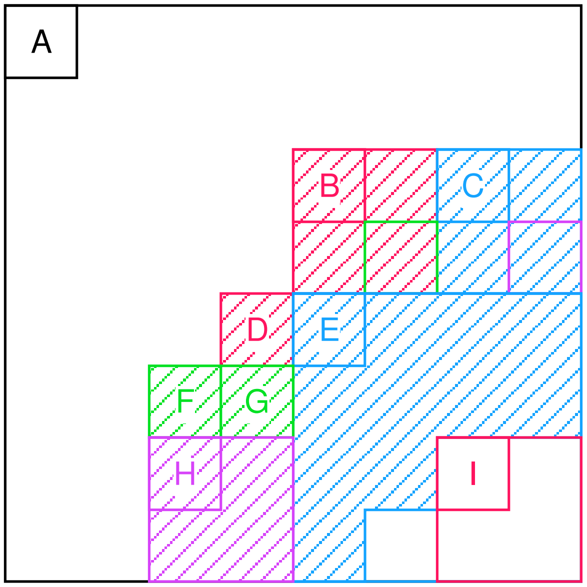




Pixel	Action	Size	Active Blocks
(0,0)	insert WHITE block A	8×8	3:A
(2,4)	insert BLACK block B	2×2	3:A 1:B
(2,6)	insert BLACK block C	2×2	3:A 1:B,C
(3,5)	remove B from active		3:A 1:C
(3,7)	remove C from active		3:A
(4,3)	insert BLACK block D		3:A
(4,4)	insert BLACK block E	4×4	3:A 2:E
(5,2) (5,3)	insert BLACK blocks F, G		3:A 2:E
(6,2)	insert BLACK block H	2×2	3:A 2:E 1:H

EXAMPLE OF OPTIMAL QUADTREE BUILDING

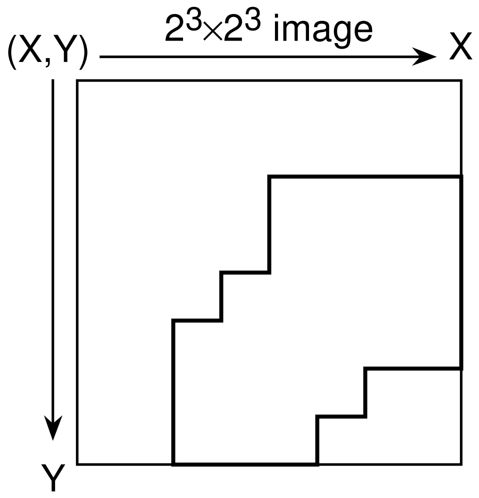


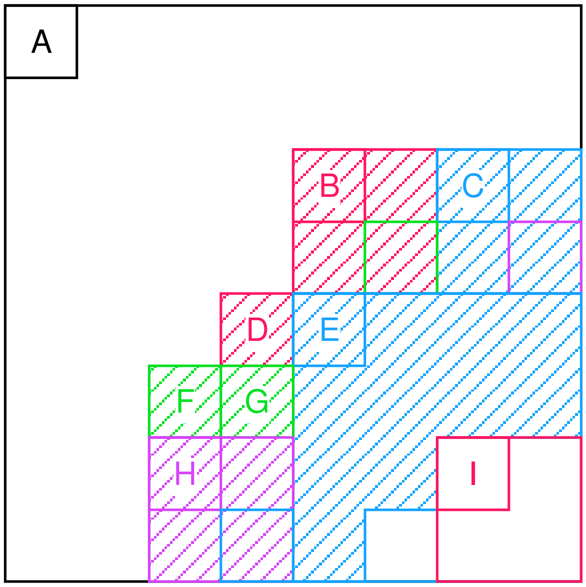


Pixel	Action	Size	Active Blocks
(0,0)	insert WHITE block A	8×8	3:A
(2,4)	insert BLACK block B	2×2	3:A 1:B
(2,6)	insert BLACK block C	2×2	3:A 1:B,C
(3,5)	remove B from active		3:A 1:C
(3,7)	remove C from active		3:A
(4,3)	insert BLACK block D		3:A
(4,4)	insert BLACK block E	4×4	3:A 2:E
(5,2) (5,3)	insert BLACK blocks F, G		3:A 2:E
(6,2)	insert BLACK block H	2×2	3:A 2:E 1:H
(6,6)	insert WHITE block I	2×2	3:A 2:E 1:H,I

11	10	9	8	7	6	5	4	3	2	1
z	r	v	g	z	r	v	g	z	r	b

EXAMPLE OF OPTIMAL QUADTREE BUILDING

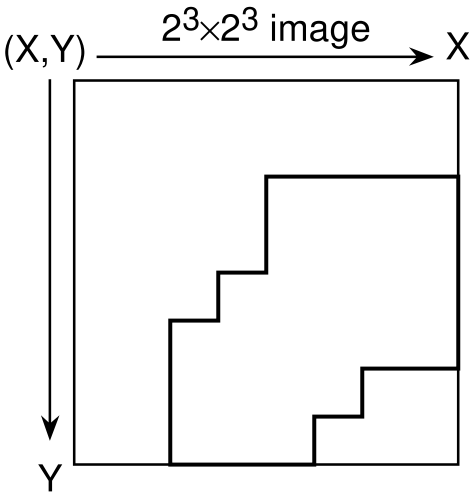


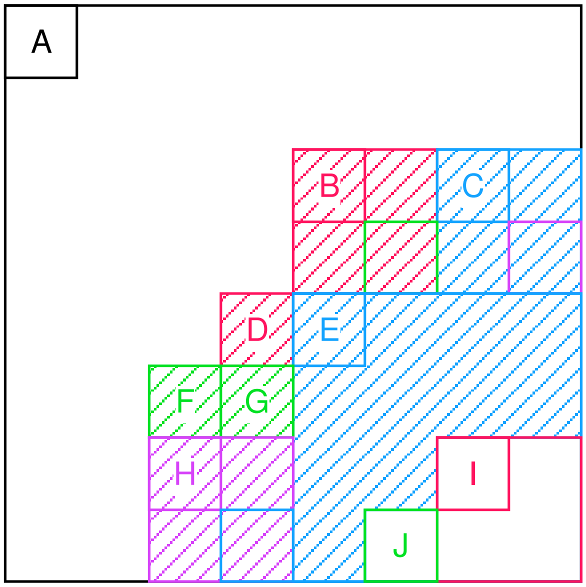


Pixel	Action	Size	Active Blocks
(0,0)	insert WHITE block A	8×8	3:A
(2,4)	insert BLACK block B	2×2	3:A 1:B
(2,6)	insert BLACK block C	2×2	3:A 1:B,C
(3,5)	remove B from active		3:A 1:C
(3,7)	remove C from active		3:A
(4,3)	insert BLACK block D		3:A
(4,4)	insert BLACK block E	4×4	3:A 2:E
(5,2) (5,3)	insert BLACK blocks F, G		3:A 2:E
(6,2)	insert BLACK block H	2×2	3:A 2:E 1:H
(6,6)	insert WHITE block I	2×2	3:A 2:E 1:H,I
(7,3)	remove H from active		3:A 2:E 1:I

12	11	10	9	8	7	6	5	4	3	2	1
g	z	r	v	g	z	r	v	g	z	r	b

EXAMPLE OF OPTIMAL QUADTREE BUILDING

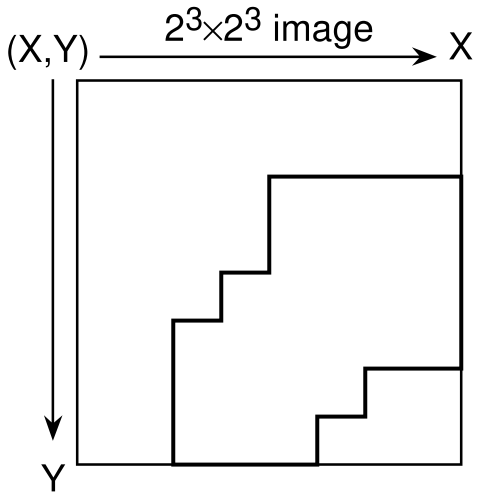


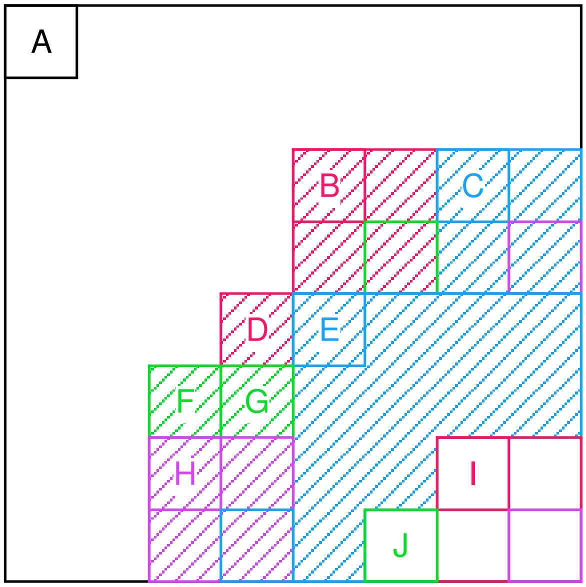


Pixel	Action	Size	Active Blocks
(0,0)	insert WHITE block A	8×8	3:A
(2,4)	insert BLACK block B	2×2	3:A 1:B
(2,6)	insert BLACK block C	2×2	3:A 1:B,C
(3,5)	remove B from active		3:A 1:C
(3,7)	remove C from active		3:A
(4,3)	insert BLACK block D		3:A
(4,4)	insert BLACK block E	4×4	3:A 2:E
(5,2) (5,3)	insert BLACK blocks F, G		3:A 2:E
(6,2)	insert BLACK block H	2×2	3:A 2:E 1:H
(6,6)	insert WHITE block I	2×2	3:A 2:E 1:H,I
(7,3)	remove H from active		3:A 2:E 1:I
(7,5)	insert WHITE block J		3:A 2:E 1:I

13	12	11	10	9	8	7	6	5	4	3	2	1
v	g	z	r	v	g	z	r	v	g	z	r	b

EXAMPLE OF OPTIMAL QUADTREE BUILDING

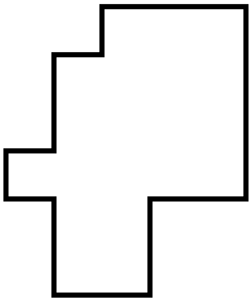




Pixel	Action	Size	Active Blocks
(0,0)	insert WHITE block A	8×8	3:A
(2,4)	insert BLACK block B	2×2	3:A 1:B
(2,6)	insert BLACK block C	2×2	3:A 1:B,C
(3,5)	remove B from active		3:A 1:C
(3,7)	remove C from active		3:A
(4,3)	insert BLACK block D		3:A
(4,4)	insert BLACK block E	4×4	3:A 2:E
(5,2) (5,3)	insert BLACK blocks F, G		3:A 2:E
(6,2)	insert BLACK block H	2×2	3:A 2:E 1:H
(6,6)	insert WHITE block I	2×2	3:A 2:E 1:H,I
(7,3)	remove H from active		3:A 2:E 1:I
(7,5)	insert WHITE block J		3:A 2:E 1:I
(7,7)	remove I, E, A from active		—

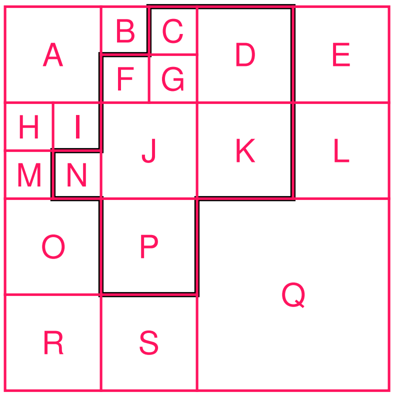
CONVERTING QUADTREES TO RASTERS

- Useful when outputting an image
- Generate row-by-row by visiting each node once for every row that intersects it — 2^k visits to a $2^k \times 2^k$ block
- Preferable to generating the entire array at once and then outputting as it takes too much memory to store the array
- Two approaches
  1. top-down: starts at the root each time it visits a node in the row
  2. bottom-up: use neighbor-finding techniques (variant of EQUAL_LATERAL_NEIGHBOR) to visit adjacent nodes in the row
- Ex:



CONVERTING QUADTREES TO RASTERS

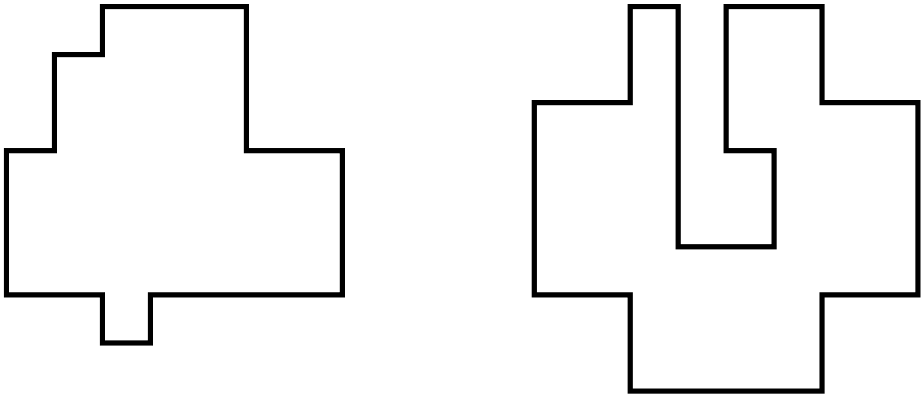
- Useful when outputting an image
- Generate row-by-row by visiting each node once for every row that intersects it — 2^k visits to a $2^k \times 2^k$ block
- Preferable to generating the entire array at once and then outputting as it takes too much memory to store the array
- Two approaches
  1. top-down: starts at the root each time it visits a node in the row
  2. bottom-up: use neighbor-finding techniques (variant of EQUAL_LATERAL_NEIGHBOR) to visit adjacent nodes in the row
- Ex:



COMPLEXITY OF CONVERTING QUADTREES TO RASTERS

- Assume a $2^n \times 2^n$ image
- Top-down algorithm visits $(n-i+1) \cdot 2^i$ nodes when it outputs a node of size $2^i \times 2^i$
- Bottom-up algorithm:
  1. number of nodes visited < 4 times the size of the sides of the blocks in the image
  2. alternative complexity measure
     - let b_i = number of blocks of size $2^i \times 2^i$
     - visits $2^{2n+2} - 2^{n+2} \sum_{i=1}^{n} b_i \cdot (2^{2i+2} - 2^{i+2})$ nodes
- Bottom-up is preferable for images larger than $2^4 \times 2^4$ as neighbor finding cost is bounded by 4 and maximum depth is greater than 4 which is a factor in the cost of the bottom-up method
- Amount of work is identical for different images with the same block configuration

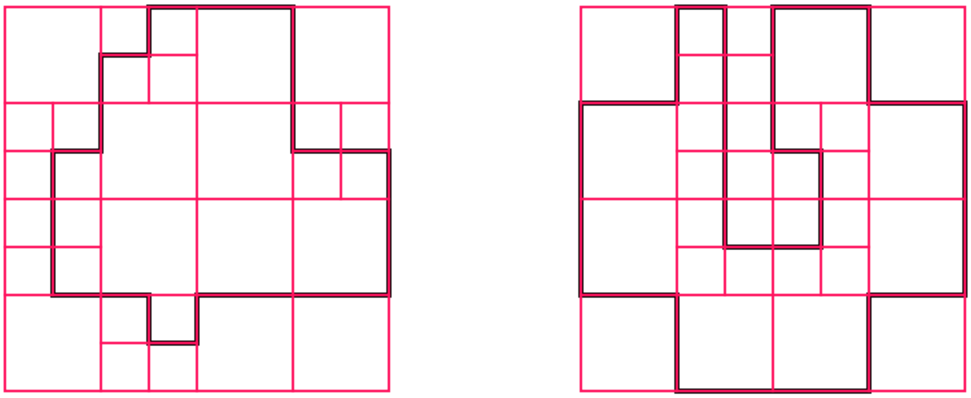
Ex:



COMPLEXITY OF CONVERTING QUADTREES TO RASTERS

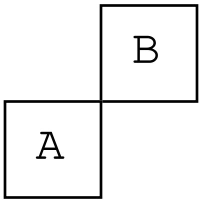
- Assume a $2^n \times 2^n$ image
- Top-down algorithm visits $(n-i+1) \cdot 2^i$ nodes when it outputs a node of size $2^i \times 2^i$
- Bottom-up algorithm:
  1. number of nodes visited < 4 times the size of the sides of the blocks in the image
  2. alternative complexity measure
     - let b_i = number of blocks of size $2^i \times 2^i$
     - visits $2^{2n+2} - 2^{n+2}\sum_{i=1}^{n} b_i \cdot (2^{2i+2} - 2^{i+2})$ nodes
- Bottom-up is preferable for images larger than $2^4 \times 2^4$ as neighbor finding cost is bounded by 4 and maximum depth is greater than 4 which is a factor in the cost of the bottom-up method
- Amount of work is identical for different images with the same block configuration

Ex:

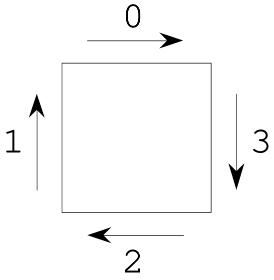


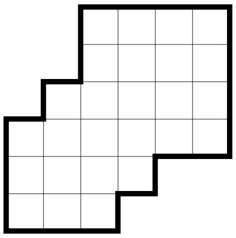
CHAIN CODE

- Records relative position of boundaries of similarly-valued adjacent grid squares
- Four directions
- Usually assume the image is to the right
- Assume four-connected, meaning that `A` and `B` are not in same region unless eight-connected



- Aggregation similar to runlength representation
  1. based on direction of boundary rather than value of location
  2. no 2 consecutive elements on the aggregated boundary have the same direction
  3. implement with a number to indicate length, OR





starting at lower left corner yields

$1^3\ 0^1\ 1^1\ 0^1\ 1^2\ 0^4$

$3^4\ 2^2\ 3^1\ 2^1\ 3^1\ 2^3$

  - can use +1 and –1 to indicate a change of direction
  - more compact than absolute directions 0,1,2,3

  4. just store the locations where the directions change

code = direction /90

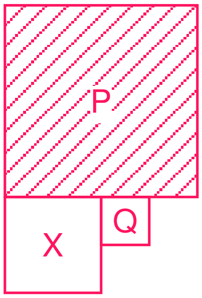
- Comparison
  1. 3 is relative while 4 is absolute
  2. 3 is more compact than 4
  3. 4 is more robust as an error means that only one element is corrupted

CONVERTING FROM QUADTREES TO CHAIN CODES

- Trace the boundary of a region in the clockwise direction once an appropriate starting point has been determined
- Starting point is an adjacency between a BLACK block P and a WHITE block Q
- Assume P is to the north of Q
- Possible overlap relationships between P and Q

CONVERTING FROM QUADTREES TO CHAIN CODES

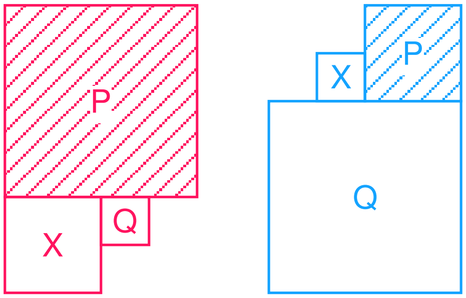
- Trace the boundary of a region in the clockwise direction once an appropriate starting point has been determined
- Starting point is an adjacency between a BLACK block P and a WHITE block Q
- Assume P is to the north of Q
- Possible overlap relationships between P and Q



1. P extends past Q

CONVERTING FROM QUADTREES TO CHAIN CODES

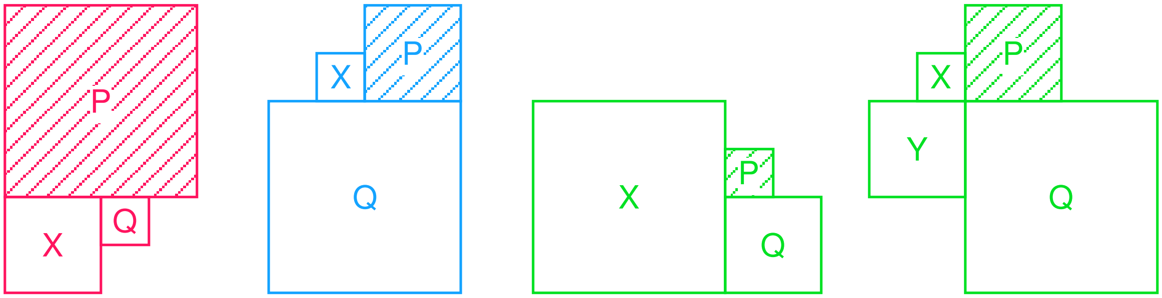
- Trace the boundary of a region in the clockwise direction once an appropriate starting point has been determined
- Starting point is an adjacency between a BLACK block P and a WHITE block Q
- Assume P is to the north of Q
- Possible overlap relationships between P and Q



1. P extends past Q
2. Q extends past P

CONVERTING FROM QUADTREES TO CHAIN CODES

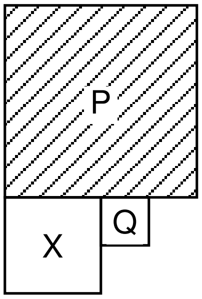
- Trace the boundary of a region in the clockwise direction once an appropriate starting point has been determined
- Starting point is an adjacency between a BLACK block P and a WHITE block Q
- Assume P is to the north of Q
- Possible overlap relationships between P and Q

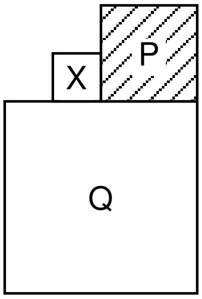


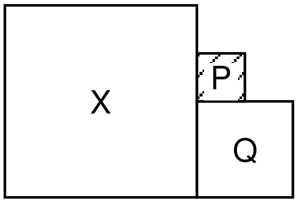
1. P extends past Q
2. Q extends past P
3. P and Q meet at the same point

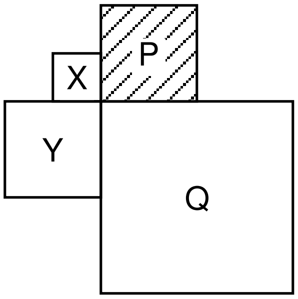
ALGORITHM TO CONVERT FROM QUADTREES TO CHAIN CODES

- Possible block configurations





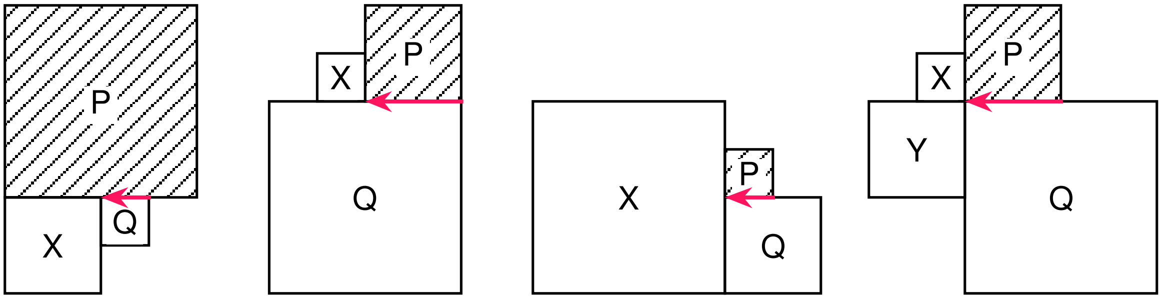




- Algorithm

ALGORITHM TO CONVERT FROM QUADTREES TO CHAIN CODES

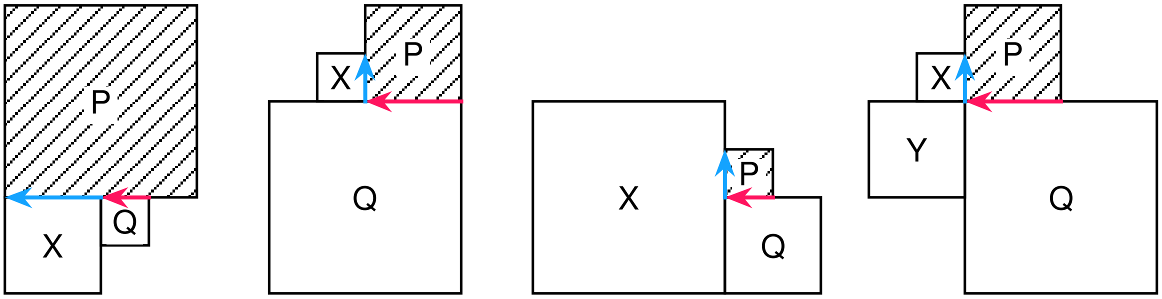
- Possible block configurations



- Algorithm
  1. Output the links of the chain code associated with the part of BLACK block P's border that is adjacent to WHITE block Q
     - length of chain = minimum of sides of P and Q

ALGORITHM TO CONVERT FROM QUADTREES TO CHAIN CODES

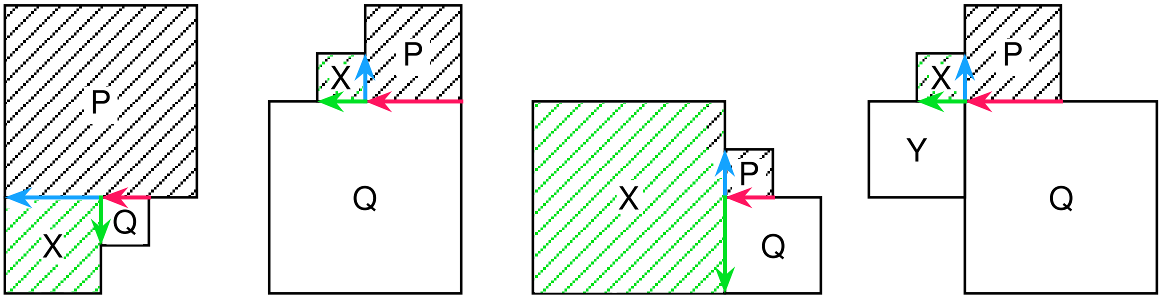
- Possible block configurations



- Algorithm
  1. Output the links of the chain code associated with the part of BLACK block P’s border that is adjacent to WHITE block Q
     - length of chain = minimum of sides of P and Q
  2. Determine the BLACK/WHITE block pair to be visited next
     - may need to inspect some adjacent blocks X and Y
     - use neighbor finding (variant of EQUAL_LATERAL_NEIGHBOR)

ALGORITHM TO CONVERT FROM QUADTREES TO CHAIN CODES

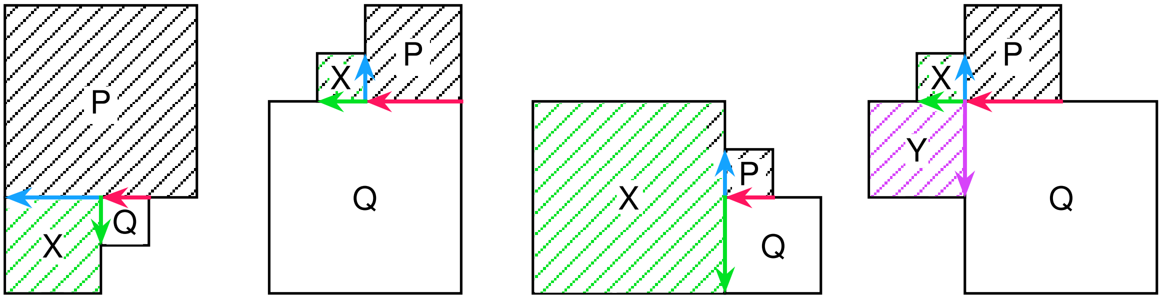
- Possible block configurations



- Algorithm
  1. Output the links of the chain code associated with the part of BLACK block P's border that is adjacent to WHITE block Q
     - length of chain = minimum of sides of P and Q
  2. Determine the BLACK/WHITE block pair to be visited next
     - may need to inspect some adjacent blocks X and Y
     - use neighbor finding (variant of EQUAL_LATERAL_NEIGHBOR)
     - different result if X is BLACK

ALGORITHM TO CONVERT FROM QUADTREES TO CHAIN CODES

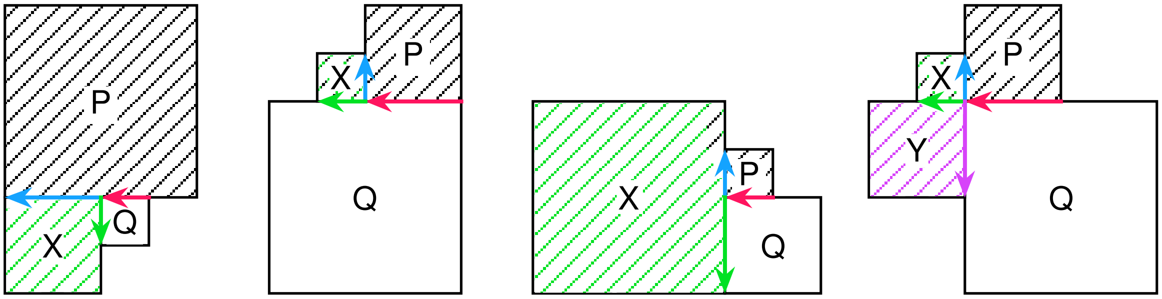
- Possible block configurations



- Algorithm
  1. Output the links of the chain code associated with the part of BLACK block P's border that is adjacent to WHITE block Q
     - length of chain = minimum of sides of P and Q
  2. Determine the BLACK/WHITE block pair to be visited next
     - may need to inspect some adjacent blocks X and Y
     - use neighbor finding (variant of EQUAL_LATERAL_NEIGHBOR)
     - different result if X is BLACK
     - another result if Y is BLACK given X is BLACK
     - don't care about Y if X is WHITE

ALGORITHM TO CONVERT FROM QUADTREES TO CHAIN CODES

- Possible block configurations



- Algorithm
  1. Output the links of the chain code associated with the part of BLACK block P's border that is adjacent to WHITE block Q
     - length of chain = minimum of sides of P and Q
  2. Determine the BLACK/WHITE block pair to be visited next
     - may need to inspect some adjacent blocks X and Y
     - use neighbor finding (variant of EQUAL_LATERAL_NEIGHBOR)
     - different result if X is BLACK
     - another result if Y is BLACK given X is BLACK
     - don't care about Y if X is WHITE
- *O*(perimeter) execution time assuming a constant cost for neighbor finding

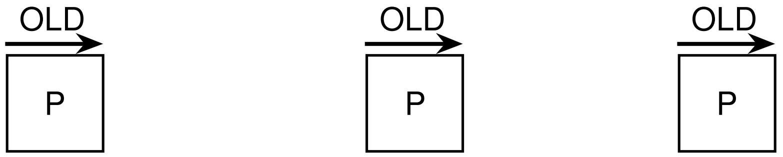
CONVERTING FROM CHAIN CODES TO QUADTREES

Algorithm:

1. traverse boundary in clockwise order and build a quadtree with BLACK nodes of unit size (i.e., 1 × 1) adjacent to the boundary
   - remaining nodes are left uncolored
2. assign the appropriate color to the uncolored nodes

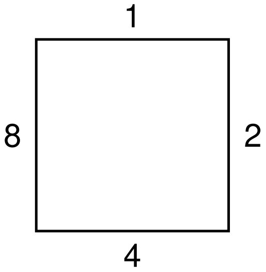
STEP 1 OF CHAIN CODE TO QUADTREE CONVERSION ALGORITHM

- Construct the tree by creating pixel-sized BLACK nodes adjacent to the links in the chain code using the relationship between successive links in the chain code

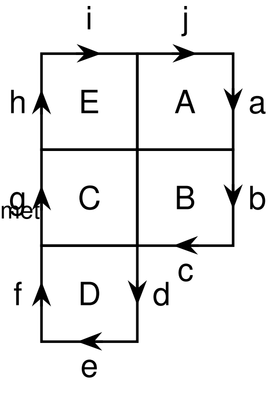


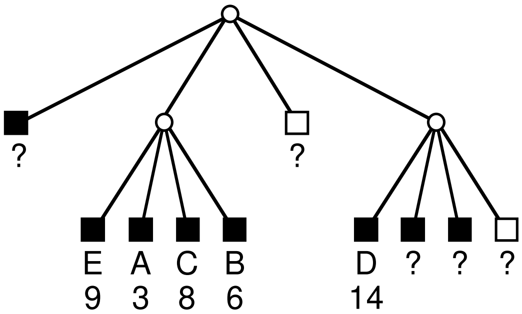
- Nodes are added by use of neighbor finding (variant of EQUAL_LATERAL_NEIGHBOR)
- As links in the chain code are processed, some nodes will be encountered more than once indicating that they are adjacent to the boundary on more than one side
- Use a code to keep track of the part of the boundary to which the node is adjacent on its right

  0: none of the sides of the node are adjacent to the boundary (such a node is said to be uncolored) 8
  1: to the right of the northern boundary
  2: to the right of the eastern boundary
  4: to the right of the southern boundary
  8: to the right of the western boundary



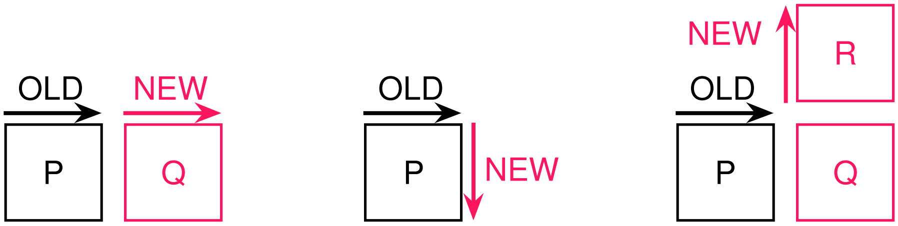
- Codes are additive
- Ex: sample region result of step 1



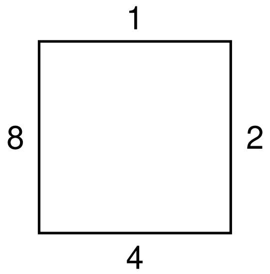


STEP 1 OF CHAIN CODE TO QUADTREE CONVERSION ALGORITHM

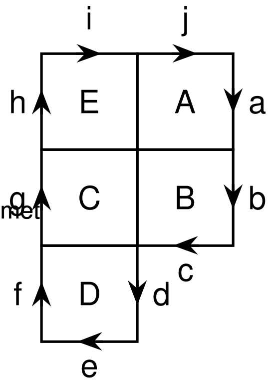
- Construct the tree by creating pixel-sized BLACK nodes adjacent to the links in the chain code using the relationship between successive links in the chain code

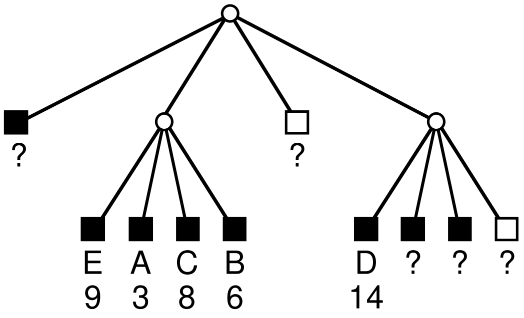


- Nodes are added by use of neighbor finding (variant of EQUAL_LATERAL_NEIGHBOR)
- As links in the chain code are processed, some nodes will be encountered more than once indicating that they are adjacent to the boundary on more than one side
- Use a code to keep track of the part of the boundary to which the node is adjacent on its right
  - 0: none of the sides of the node are adjacent to the boundary (such a node is said to be uncolored) 8
  - 1: to the right of the northern boundary
  - 2: to the right of the eastern boundary
  - 4: to the right of the southern boundary
  - 8: to the right of the western boundary



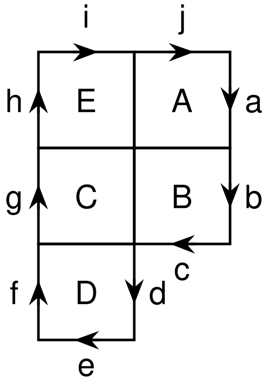
- Codes are additive
- Ex: sample region result of step 1

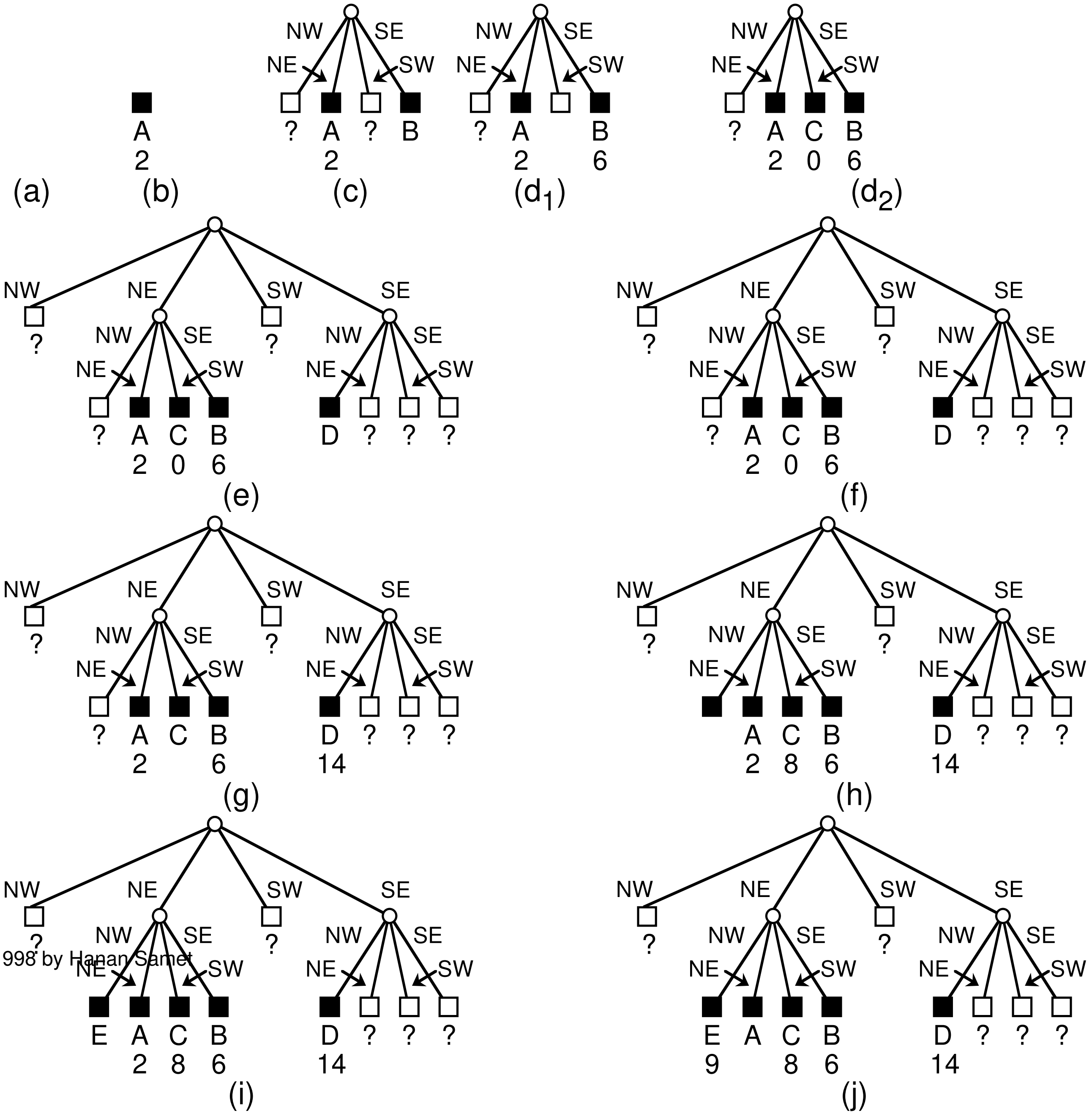




EXAMPLE OF STEP 1 OF CONVERTING FROM CHAIN CODES TO QUADTREES

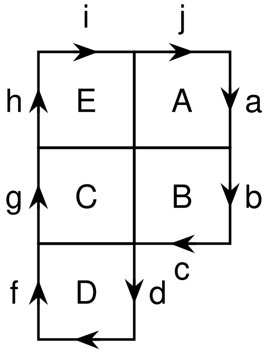
- Ex:
- Use a variant of EQUAL_LATERAL_NEIGHBOR to find neighbors in the eastern direction and add them if they are not present





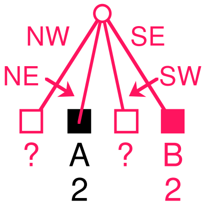
EXAMPLE OF STEP 1 OF CONVERTING FROM CHAIN CODES TO QUADTREES

- Ex:

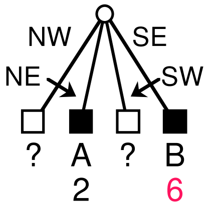


- Use a variant of EQUAL_LATERAL_NEIGHBOR to find neighbors in the eastern direction and add them if they are not present

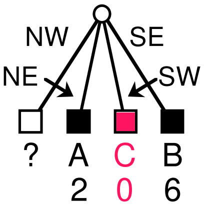
(a)



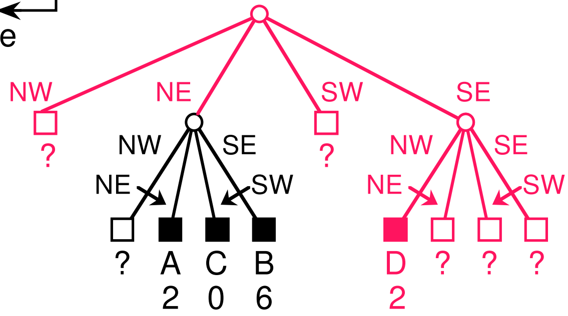
(b)



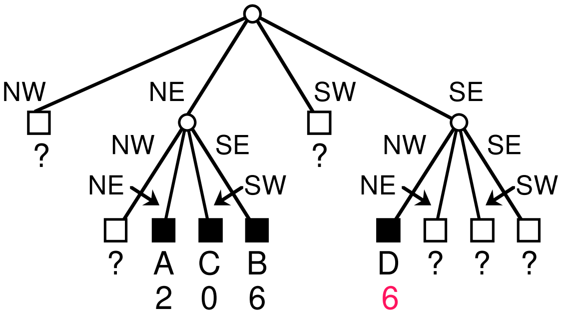
(c)



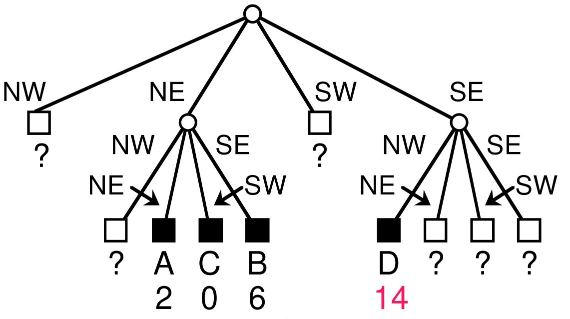
(d_1)



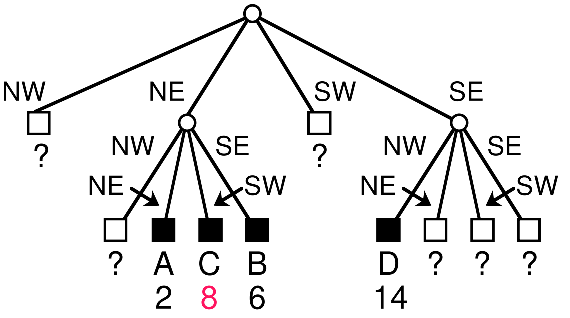
(d_2)



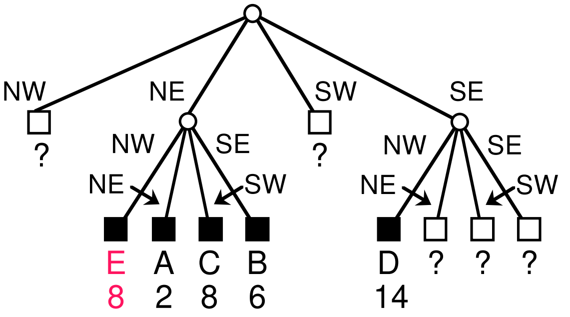
(e)



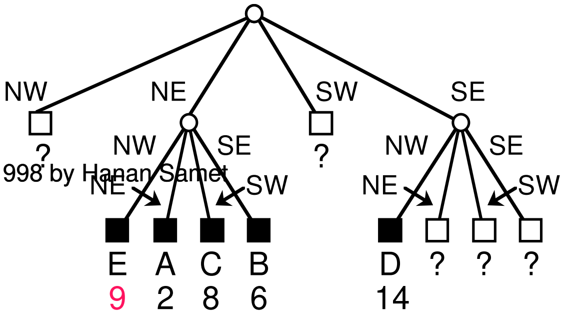
(f)



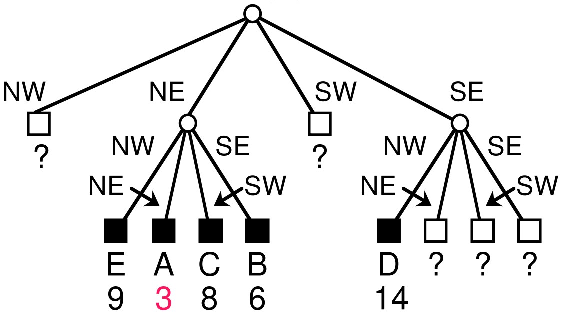
(g)



(h)



(i)

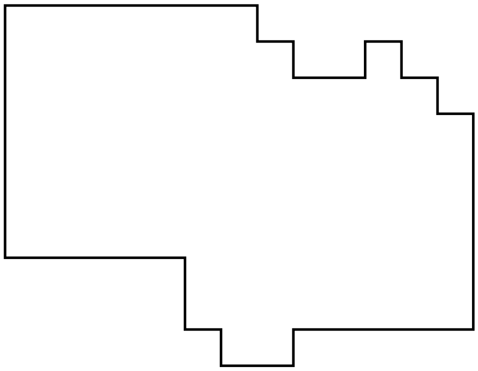


(j)

EXAMPLE OF STEP 2 OF CONVERTING FROM CHAIN CODES TO QUADTREES

- Need to determine which nodes are inside the region and which ones are outside — i.e., assign colors to uncolored nodes and merge if necessary
- Could use seed filling algorithm or point-in-polygon tests but cumbersome
- Instead, use the codes in the result of step 1
- Key observation is that if any uncolored sons are determined to be BLACK (WHITE), then all remaining uncolored brothers must also be BLACK (WHITE)
- Algorithm traverses tree in preorder looking for uncolored nodes that are adjacent to colored ones (along the border at deepest level of the tree) and merging whenever possible

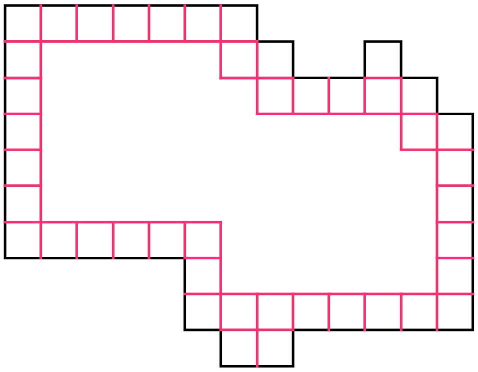
Ex:



EXAMPLE OF STEP 2 OF CONVERTING FROM CHAIN CODES TO QUADTREES

- Need to determine which nodes are inside the region and which ones are outside — i.e., assign colors to uncolored nodes and merge if necessary
- Could use seed filling algorithm or point-in-polygon tests but cumbersome
- Instead, use the codes in the result of step 1
- Key observation is that if any uncolored sons are determined to be BLACK (WHITE), then all remaining uncolored brothers must also be BLACK (WHITE)
- Algorithm traverses tree in preorder looking for uncolored nodes that are adjacent to colored ones (along the border at deepest level of the tree) and merging whenever possible

Ex:

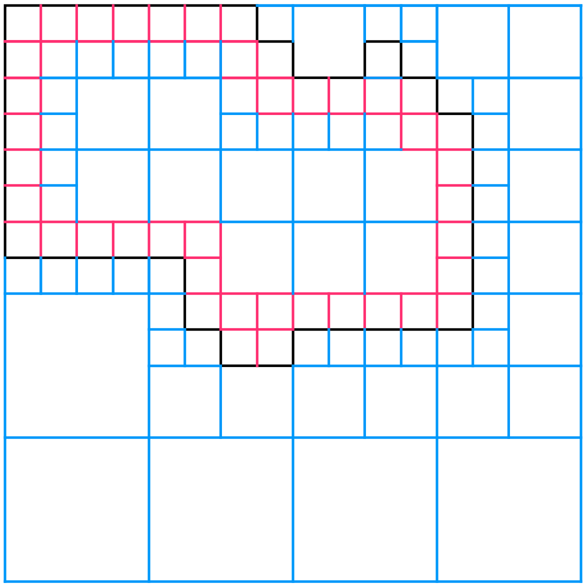


- Boundary elements

EXAMPLE OF STEP 2 OF CONVERTING FROM CHAIN CODES TO QUADTREES

- Need to determine which nodes are inside the region and which ones are outside — i.e., assign colors to uncolored nodes and merge if necessary
- Could use seed filling algorithm or point-in-polygon tests but cumbersome
- Instead, use the codes in the result of step 1
- Key observation is that if any uncolored sons are determined to be BLACK (WHITE), then all remaining uncolored brothers must also be BLACK (WHITE)
- Algorithm traverses tree in preorder looking for uncolored nodes that are adjacent to colored ones (along the border at deepest level of the tree) and merging whenever possible

Ex:

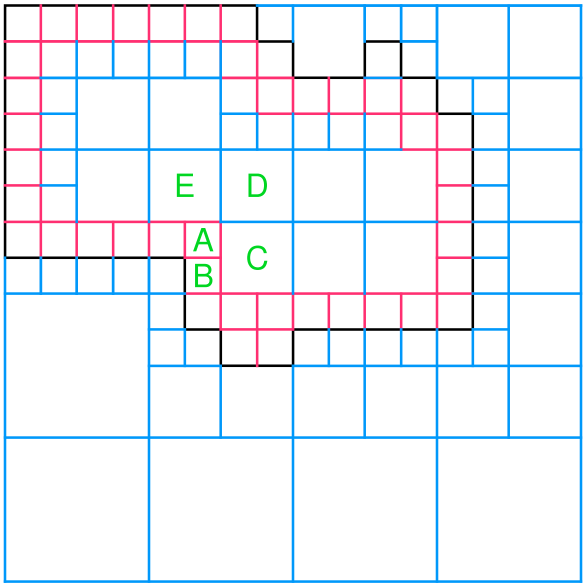


- Boundary elements
- Output of step 1

EXAMPLE OF STEP 2 OF CONVERTING FROM CHAIN CODES TO QUADTREES

- Need to determine which nodes are inside the region and which ones are outside — i.e., assign colors to uncolored nodes and merge if necessary
- Could use seed filling algorithm or point-in-polygon tests but cumbersome
- Instead, use the codes in the result of step 1
- Key observation is that if any uncolored sons are determined to be BLACK (WHITE), then all remaining uncolored brothers must also be BLACK (WHITE)
- Algorithm traverses tree in preorder looking for uncolored nodes that are adjacent to colored ones (along the border at deepest level of the tree) and merging whenever possible

Ex:

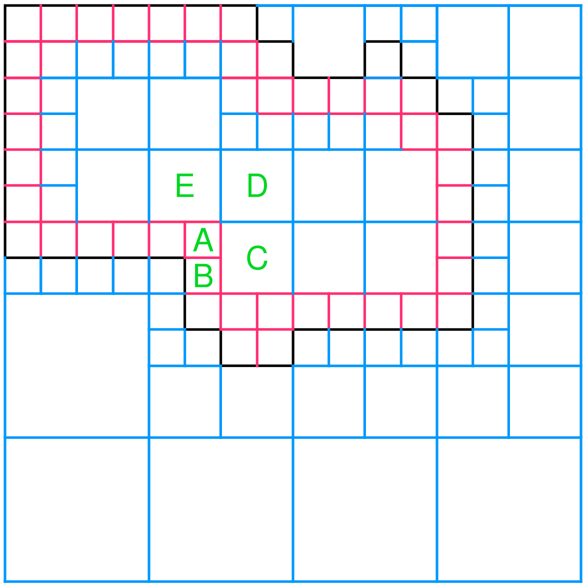


- Boundary elements
- Output of step 1
- Uncolored node C must be BLACK because it is adjacent to a GRAY node containing nodes A and B which are BLACK, and hence C's uncolored brothers D and E must also be BLACK

EXAMPLE OF STEP 2 OF CONVERTING FROM CHAIN CODES TO QUADTREES

- Need to determine which nodes are inside the region and which ones are outside — i.e., assign colors to uncolored nodes and merge if necessary
- Could use seed filling algorithm or point-in-polygon tests but cumbersome
- Instead, use the codes in the result of step 1
- Key observation is that if any uncolored sons are determined to be BLACK (WHITE), then all remaining uncolored brothers must also be BLACK (WHITE)
- Algorithm traverses tree in preorder looking for uncolored nodes that are adjacent to colored ones (along the border at deepest level of the tree) and merging whenever possible

Ex:



- Boundary elements
- Output of step 1
- Uncolored node C must be BLACK because it is adjacent to a GRAY node containing nodes A and B which are BLACK, and hence C's uncolored brothers D and E must also be BLACK
- Average execution time ≈ the region's perimeter

GEOMETRIC PROPERTIES 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

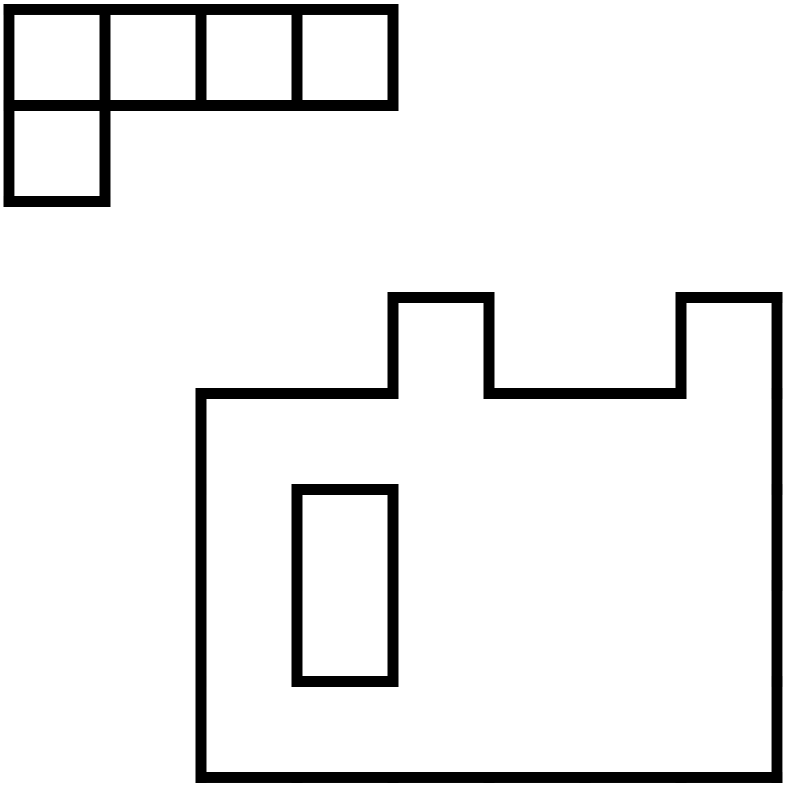
CONNECTED COMPONENT LABELING

- Assign the same label to all adjacent pixels with the same color (i.e., that share an edge rather than just a corner)
- Follows thresholding step in all image processing applications and is a prerequisite to segmentation
- Two approaches

  1. fill algorithm or polygon coloring in computer graphics
     - for each pixel find all adjacent pixels with the same color and assign them the same label
     - analogous to treating pixels as vertices in a graph and adjacencies between pixels of the same color as edges
     - can be time consuming when the image is stored on disk as the image is traversed in an arbitrary order
     - analogous to a depth-first traversal of the graph of finding its connected components
     - execution time is proportional to the number of pixels in the image as there is a bound on the number of edges in which each pixel can participate (4 in two dimensions)
  2. predetermined
     - visit all pairs of adjacent pixels row by row propagating labels between adjacent pixels to the left and above with the same color
     - merge the equivalences (using the union-find algorithm) and update the labels of affected pixels
     - graph analogy holds where now the algorithm is applied to all the edges in the graph rather than all vertices in the graph
     - useful when image is stored on disk as edges can be processed in the order in which they are stored on disk

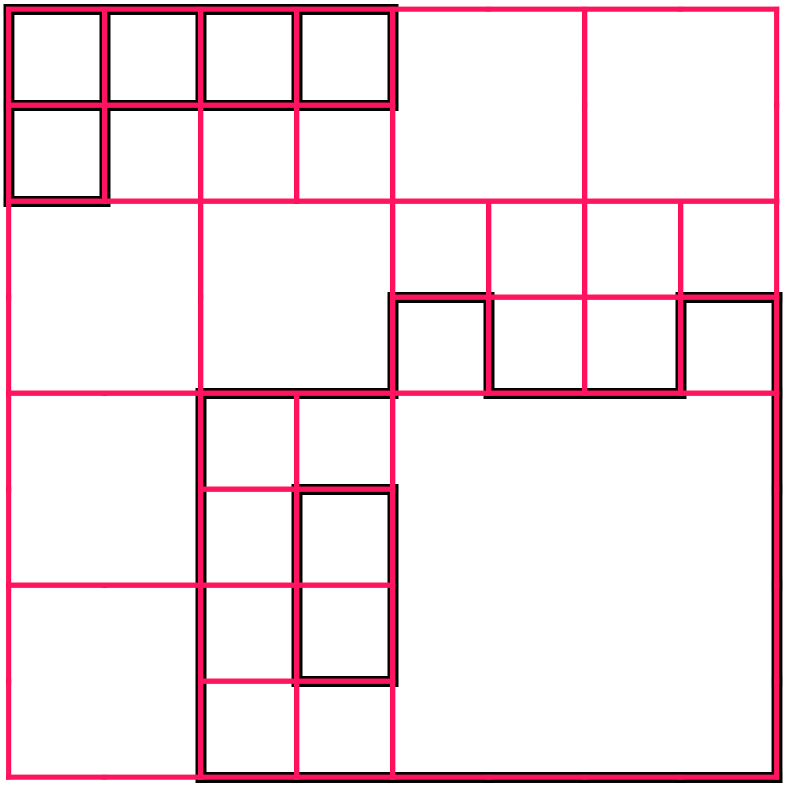
CONNECTED COMPONENT LABELING IN QUADTREES

- Assume predetermined approach
- Find all black-black adjacencies
- Ex:



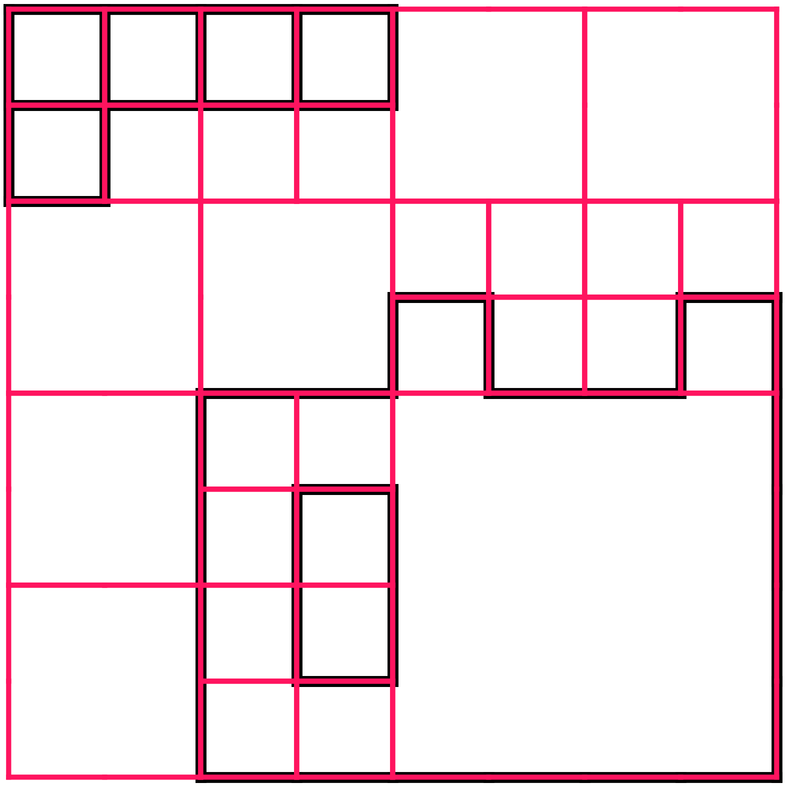
CONNECTED COMPONENT LABELING IN QUADTREES

- Assume predetermined approach
- Find all black-black adjacencies
- Ex:



CONNECTED COMPONENT LABELING IN QUADTREES

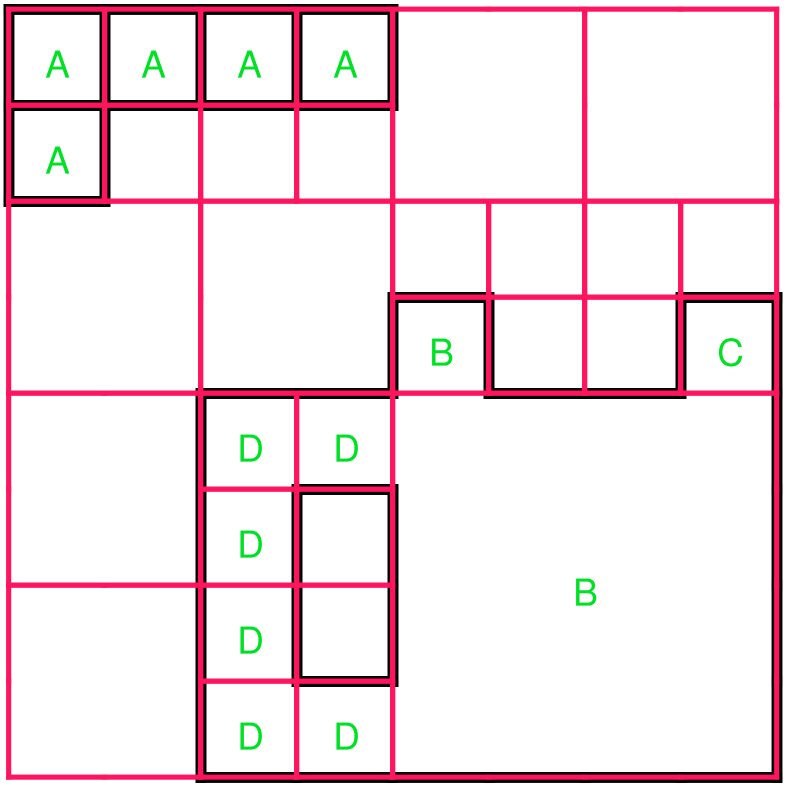
- Assume predetermined approach
- Find all black-black adjacencies
- Ex:



- For each black block, label all adjacent black blocks on the north and west sides with the same equivalence class and generate new equivalences if necessary

CONNECTED COMPONENT LABELING IN QUADTREES

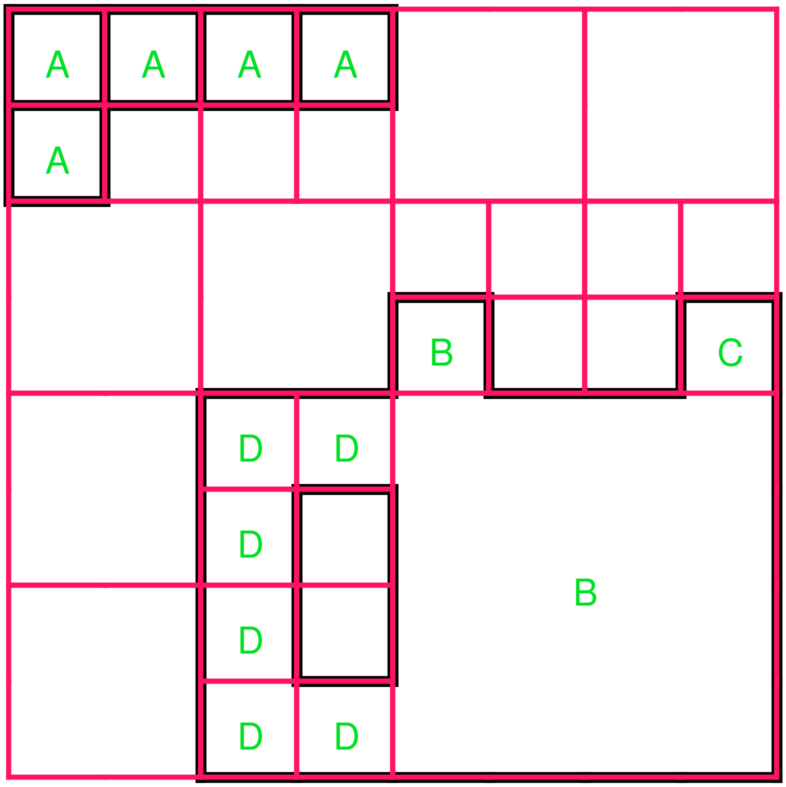
- Assume predetermined approach
- Find all black-black adjacencies
- Ex:



- For each black block, label all adjacent black blocks on the north and west sides with the same equivalence class and generate new equivalences if necessary

CONNECTED COMPONENT LABELING IN QUADTREES

- Assume predetermined approach
- Find all black-black adjacencies
- Ex:



- For each black block, label all adjacent black blocks on the north and west sides with the same equivalence class and generate new equivalences if necessary
- Execution time is almost linear in the number of blocks (i.e., multiplied by the inverse of Ackerman's function)

PERIMETER

- Equivalent to first step of connected component labeling except that look for black/white adjacencies instead of black/black adjacencies
- Many definitions
- Algorithms for crack perimeter

ALG1: for each black block examine all four adjacent sides

ALG2: 2 step procedure

1. for each black block look for adjacent white north and west neighbors
2. for each white block look for adjacent black north and west neighbors
3. can avoid special cases of blocks along the border by embedding the image in a white region

ALG3: perform one of steps 1 or 2 of ALG2 and double the result as the sum of the contribution of east (north) sides of black blocks is identical to the sum of the contribution of west (south) sides of black blocks since the sides are all orthogonal

image →

← image

PERIMETER

- Equivalent to first step of connected component labeling except that look for black/white adjacencies instead of black/black adjacencies
- Many definitions

  1. crack perimeter (16)

- Algorithms for crack perimeter

  ALG1: for each black block examine all four adjacent sides

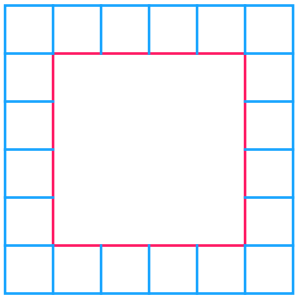
  ALG2: 2 step procedure

  1. for each black block look for adjacent white north and west neighbors
  2. for each white block look for adjacent black north and west neighbors
  3. can avoid special cases of blocks along the border by embedding the image in a white region

  ALG3: perform one of steps 1 or 2 of ALG2 and double the result as the sum of the contribution of east (north) sides of black blocks is identical to the sum of the contribution of west (south) sides of black blocks since the sides are all orthogonal

PERIMETER

- Equivalent to first step of connected component labeling except that look for black/white adjacencies instead of black/black adjacencies
- Many definitions
  1. crack perimeter (16)
  2. outer perimeter (20)



- Algorithms for crack perimeter

  ALG1: for each black block examine all four adjacent sides

  ALG2: 2 step procedure
  1. for each black block look for adjacent white north and west neighbors
  2. for each white block look for adjacent black north and west neighbors
  3. can avoid special cases of blocks along the border by embedding the image in a white region

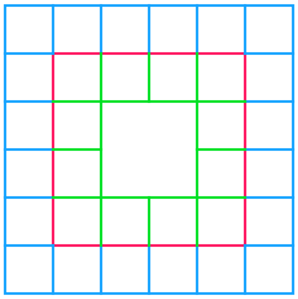
  ALG3: perform one of steps 1 or 2 of ALG2 and double the result as the sum of the contribution of east (north) sides of black blocks is identical to the sum of the contribution of west (south) sides of black blocks since the sides are all orthogonal

image

image

PERIMETER

- Equivalent to first step of connected component labeling except that look for black/white adjacencies instead of black/black adjacencies
- Many definitions
  1. crack perimeter (16)
  2. outer perimeter (20)
  3. inner perimeter (12)



- Algorithms for crack perimeter

ALG1: for each black block examine all four adjacent sides

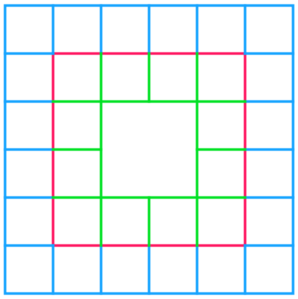
ALG2: 2 step procedure

1. for each black block look for adjacent white north and west neighbors
2. for each white block look for adjacent black north and west neighbors
3. can avoid special cases of blocks along the border by embedding the image in a white region

ALG3: perform one of steps 1 or 2 of ALG2 and double the result as the sum of the contribution of east (north) sides of black blocks is identical to the sum of the contribution of west (south) sides of black blocks since the sides are all orthogonal

PERIMETER

- Equivalent to first step of connected component labeling except that look for black/white adjacencies instead of black/black adjacencies
- Many definitions
  1. crack perimeter (16)
  2. outer perimeter (20)
  3. inner perimeter (12)



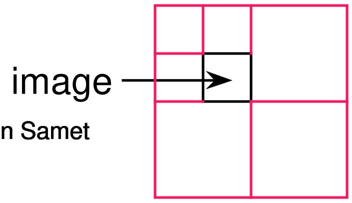
- Algorithms for crack perimeter

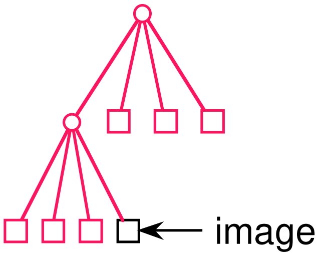
ALG1: for each black block examine all four adjacent sides

ALG2: 2 step procedure

1. for each black block look for adjacent white north and west neighbors
2. for each white block look for adjacent black north and west neighbors
3. can avoid special cases of blocks along the border by embedding the image in a white region

ALG3: perform one of steps 1 or 2 of ALG2 and double the result as the sum of the contribution of east (north) sides of black blocks is identical to the sum of the contribution of west (south) sides of black blocks since the sides are all orthogonal





EULER NUMBER (GENUS)

- Graph-theoretic definition: G = vertices – edges + faces
- For a binary image: number of components minus number of holes
- Can show that the same result holds for a quadtree representation of a binary image using an appropriate analogy between vertices and blocks and adjacencies involving blocks of differing sizes

Binary image:

Quadtree

EULER NUMBER (GENUS)

- Graph-theoretic definition: G = vertices – edges + faces
- For a binary image: number of components minus number of holes
- Can show that the same result holds for a quadtree representation of a binary image using an appropriate analogy between vertices and blocks and adjacencies involving blocks of differing sizes

Binary image:

V = number of 1’s in the image

Quadtree

V = number of black blocks

EULER NUMBER (GENUS)

- Graph-theoretic definition: G = vertices – edges + faces
- For a binary image: number of components minus number of holes
- Can show that the same result holds for a quadtree representation of a binary image using an appropriate analogy between vertices and blocks and adjacencies involving blocks of differing sizes

Binary image:

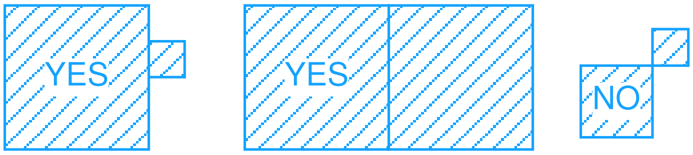
V = number of 1’s in the image

E = number of pairs of 11 or $\begin{matrix}1\\1\end{matrix}$

Quadtree

V = number of black blocks

E = number of pairs of adjacent black blocks



EULER NUMBER (GENUS)

- Graph-theoretic definition: G = vertices – edges + faces
- For a binary image: number of components minus number of holes
- Can show that the same result holds for a quadtree representation of a binary image using an appropriate analogy between vertices and blocks and adjacencies involving blocks of differing sizes

Binary image:

V = number of 1's in the image

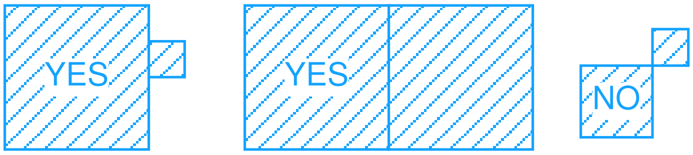
E = number of pairs of 11 or $\begin{matrix}1\\1\end{matrix}$

F = number of $2 \cdot 2$ blocks of 1's

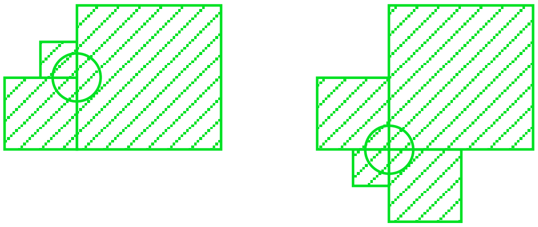
Quadtree

V = number of black blocks

E = number of pairs of adjacent black blocks



F = number of cases where three or four black blocks touch at a common point



EULER NUMBER (GENUS)

- Graph-theoretic definition: G = vertices – edges + faces
- For a binary image: number of components minus number of holes
- Can show that the same result holds for a quadtree representation of a binary image using an appropriate analogy between vertices and blocks and adjacencies involving blocks of differing sizes

Binary image:

V = number of 1's in the image

E = number of pairs of 11 or $\begin{matrix}1\\1\end{matrix}$

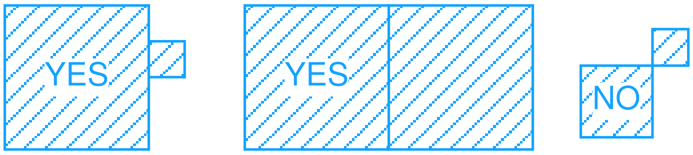
F = number of 2 · 2 blocks of 1's

G = V – E + F (Minsky and Papert)

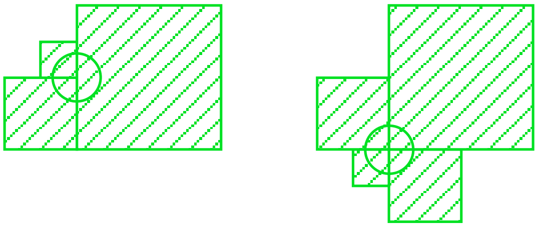
Quadtree

V = number of black blocks

E = number of pairs of adjacent black blocks



F = number of cases where three or four black blocks touch at a common point



G = V – E + F (Dyer)

- Reinforces observation that the important concept in analyzing quadtree algorithms is the number of blocks and not their sizes

ALGORITHM FOR THE COMPUTATION OF THE EULER NUMBER

- Postorder tree traversal
- Analogous to step 1 of connected component labeling $O(B+W)$

Algorithm:

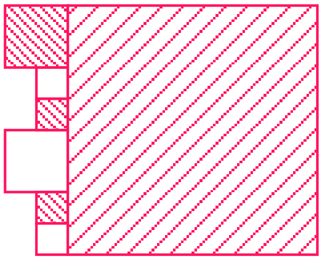
- For each BLACK node:
  1. $B \leftarrow B+1$
  2. check all leaf nodes adjacent on the W and N sides
  3. check leaf nodes surrounding the NW corner

ALGORITHM FOR THE COMPUTATION OF THE EULER NUMBER

- Postorder tree traversal
- Analogous to step 1 of connected component labeling O(B+W)

Algorithm:

- For each BLACK node:
  1. B←B+1
  2. check all leaf nodes adjacent on the W and N sides
     a. A←A+1 for each BLACK neighbor



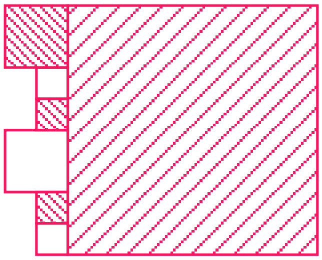
  3. check leaf nodes surrounding the NW corner

ALGORITHM FOR THE COMPUTATION OF THE EULER NUMBER

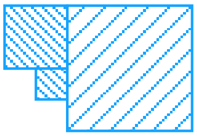
- Postorder tree traversal
- Analogous to step 1 of connected component labeling *O*(B+W)

Algorithm:

- For each BLACK node:
  1. B←B+1
  2. check all leaf nodes adjacent on the W and N sides
     a. A←A+1 for each BLACK neighbor



     b. S←S+1 for each successive pair along a side



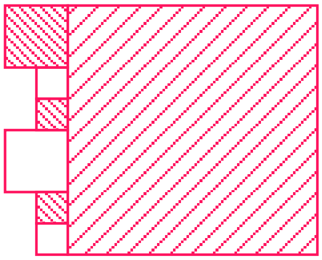
  3. check leaf nodes surrounding the NW corner

ALGORITHM FOR THE COMPUTATION OF THE EULER NUMBER

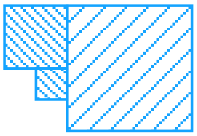
- Postorder tree traversal
- Analogous to step 1 of connected component labeling
  O(B+W)

Algorithm:

- For each BLACK node:
  1. B←B+1
  2. check all leaf nodes adjacent on the W and N sides
     a. A←A+1 for each BLACK neighbor

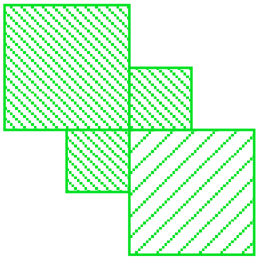


     b. S←S+1 for each successive pair along a side



  3. check leaf nodes surrounding the NW corner

     S←S+1 if all are BLACK

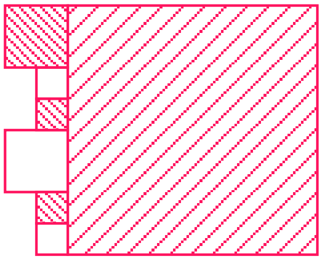


ALGORITHM FOR THE COMPUTATION OF THE EULER NUMBER

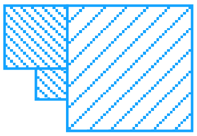
- Postorder tree traversal
- Analogous to step 1 of connected component labeling *O*(B+W)

Algorithm:

- For each BLACK node:
  1. B←B+1
  2. check all leaf nodes adjacent on the W and N sides
     a. A←A+1 for each BLACK neighbor

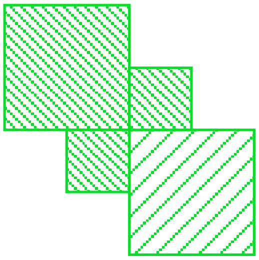


     b. S←S+1 for each successive pair along a side



  3. check leaf nodes surrounding the NW corner

     S←S+1 if all are BLACK



  4. G = B − A + S

TRANSFORMATIONS ON 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

SET OPERATIONS ON QUADTREES

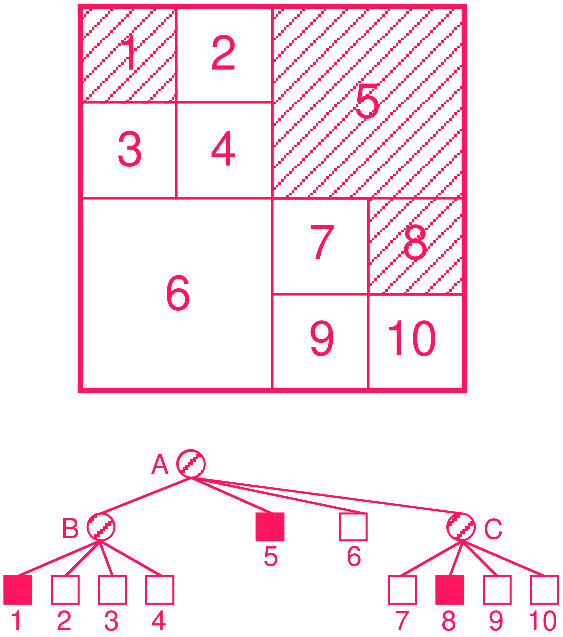
- UNION(S,T) : traverse S and T in tandem

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

SET OPERATIONS ON QUADTREES

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

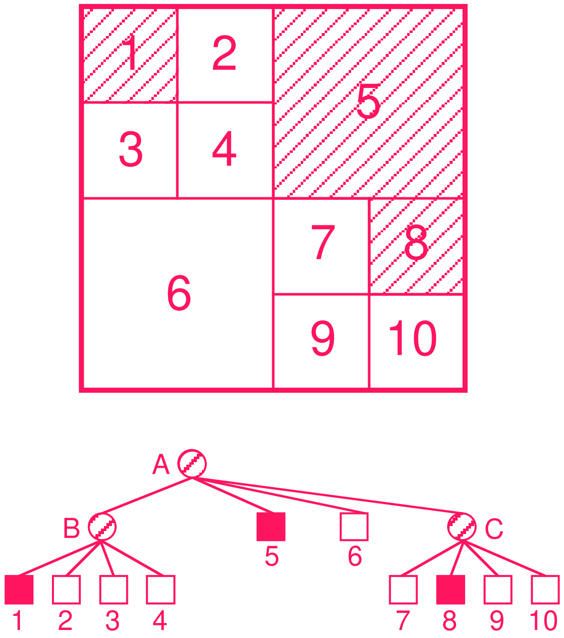
  1. GRAY(S) ⊘ :
     - GRAY(T) ⊘ : recursively process subtrees and merge if all resulting sons are BLACK
     - BLACK(T) ● : result is T
     - WHITE(T) ○ : result is S
  2. BLACK(S) ● : result is S
  3. WHITE(S) ○ : result is T

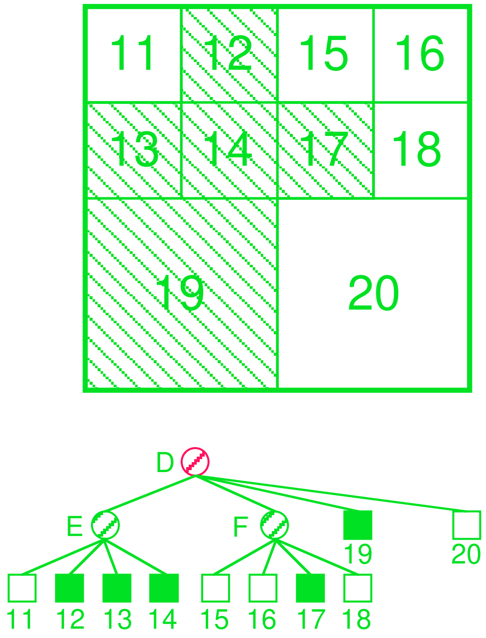


SET OPERATIONS ON QUADTREES

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

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



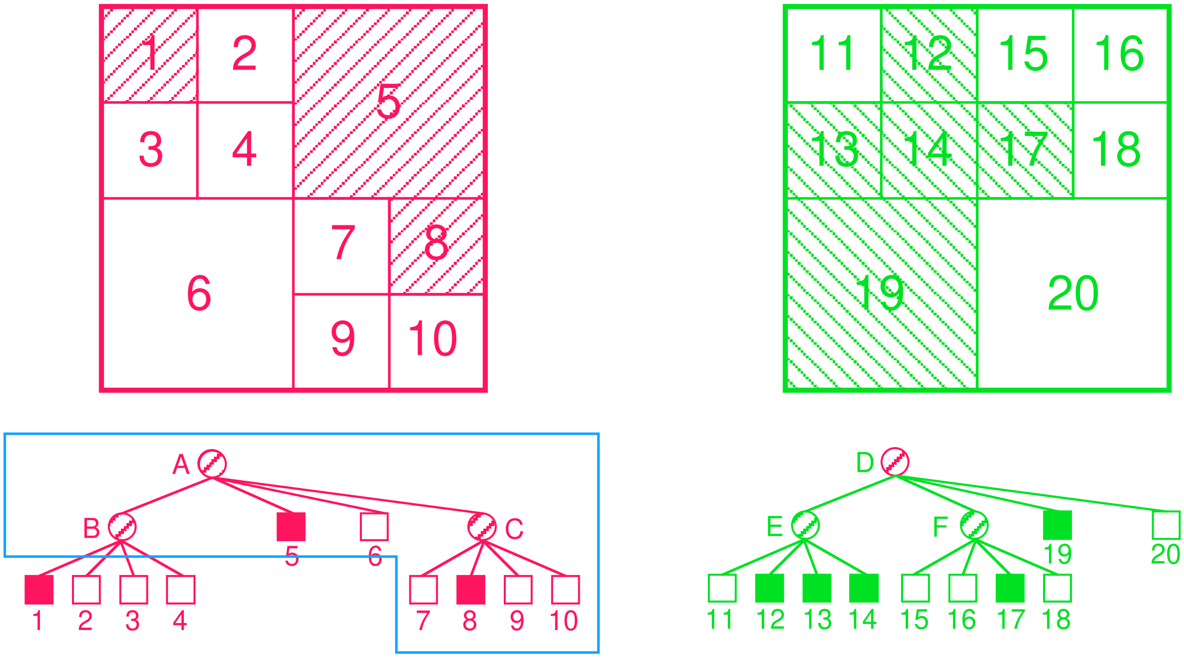


4	3	2	1
z	g	r	b

tf1

SET OPERATIONS ON QUADTREES

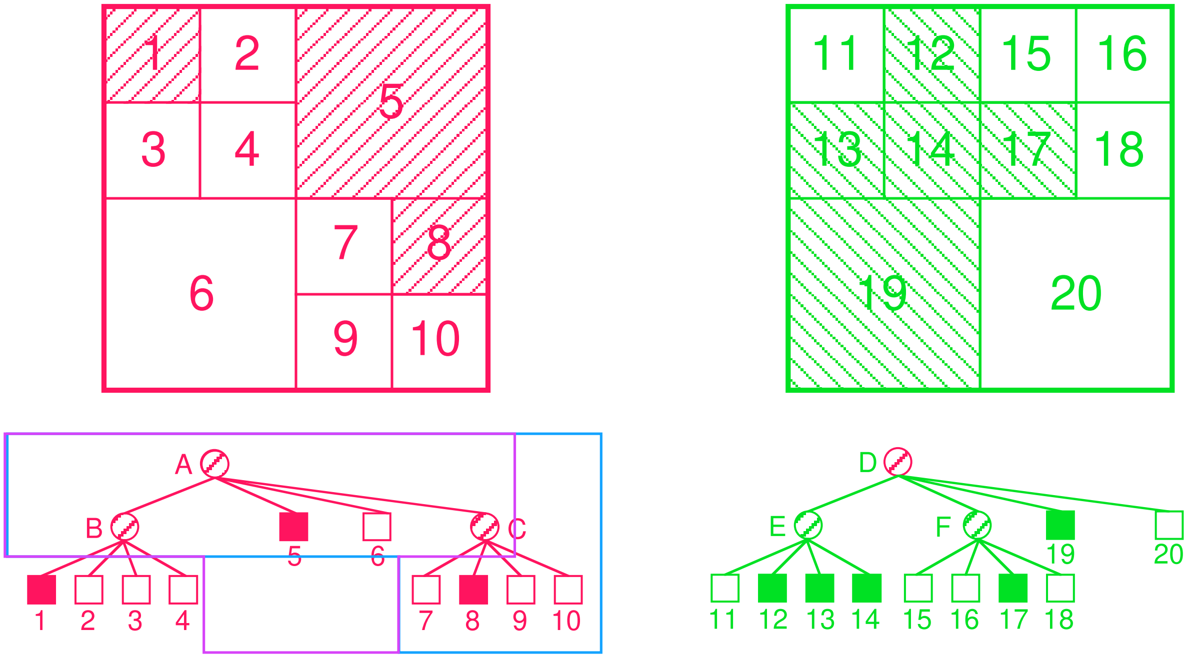
- UNION(S,T) : traverse S and T in tandem
  1. GRAY(S) ⊘ :
     - GRAY(T) ⊘ : recursively process subtrees and merge if all resulting sons are BLACK
     - BLACK(T) ● : result is T
     - WHITE(T) ○ : result is S
  2. BLACK(S) ● : result is S
  3. WHITE(S) ○ : result is T



SET OPERATIONS ON QUADTREES

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

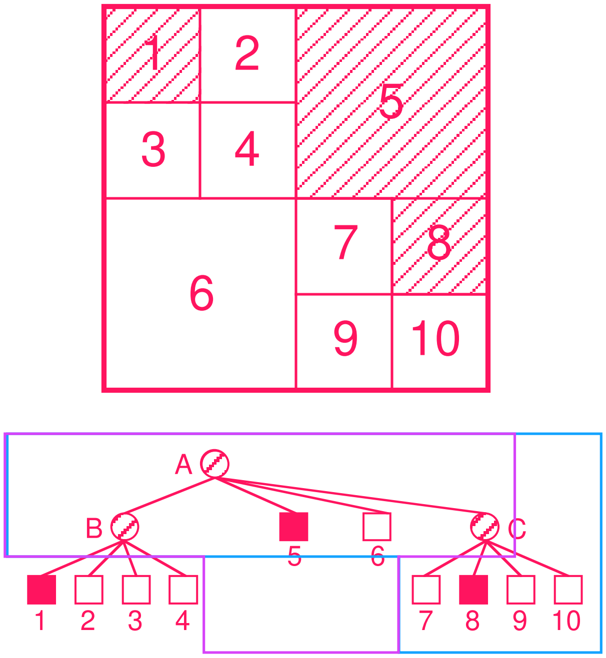
1. GRAY(S) ▨ :
   - GRAY(T) ▨ : recursively process subtrees and merge if all resulting sons are BLACK
   - BLACK(T) ● : result is T
   - WHITE(T) ○ : result is S
2. BLACK(S) ● : result is S
3. WHITE(S) ○ : result is T

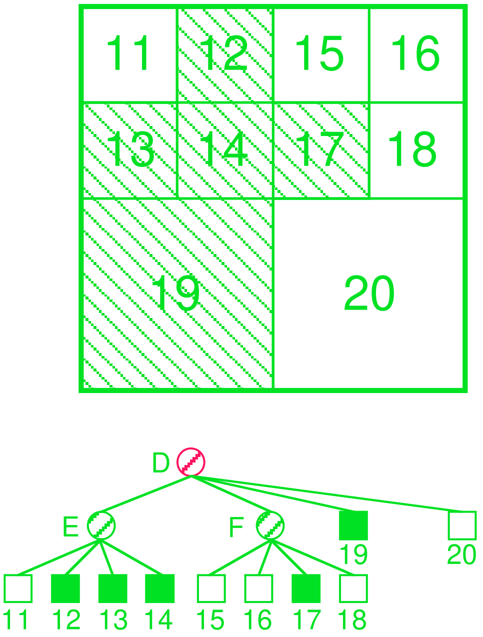


- INTERSECTION: interchange roles of BLACK and WHITE in UNION

SET OPERATIONS ON QUADTREES

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





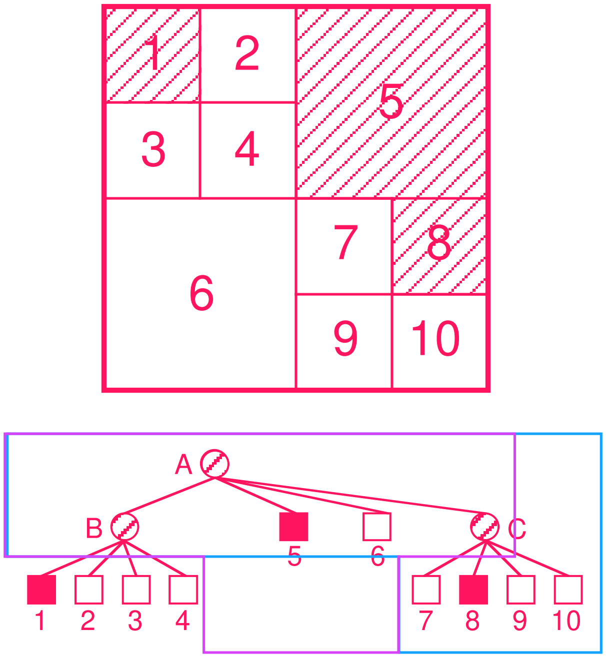
- INTERSECTION: interchange roles of BLACK and WHITE in UNION
- Execution time is bounded by sum of nodes in two input trees but may be less if don't create a new copy as really just the sum of the minimum of the number of nodes at corresponding levels of the two quadtrees

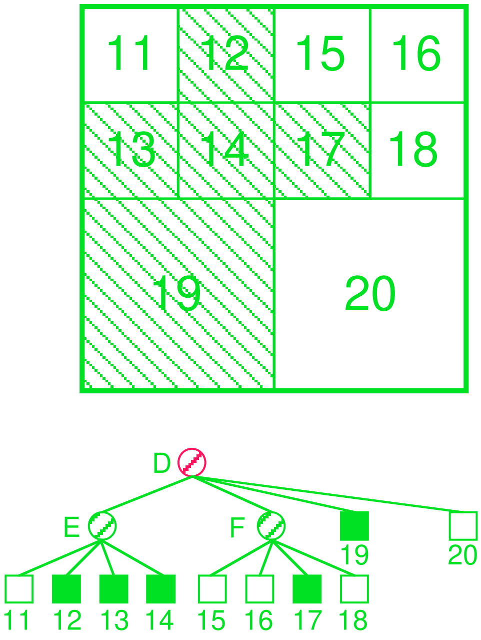
SET OPERATIONS ON QUADTREES

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

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

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

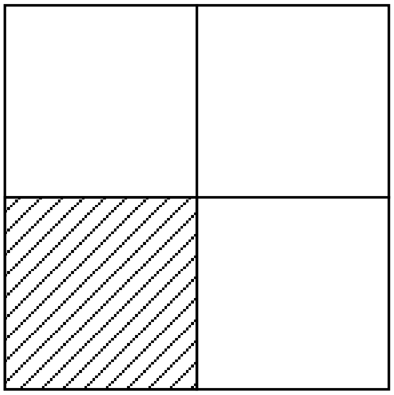




- INTERSECTION: interchange roles of BLACK and WHITE in UNION
- Execution time is bounded by sum of nodes in two input trees but may be less if don't create a new copy as really just the sum of the minimum of the number of nodes at corresponding levels of the two quadtrees
- More efficient than vectors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

WINDOWING

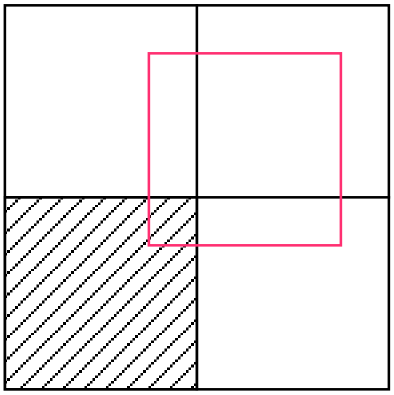
- General algorithm: check if a node in the target tree is completely contained by a node in the source tree; otherwise, decompose the target tree node and repeat the process and possibly merge later
  1. find smallest subtree of input quadtree containing window
  2. if subtree is a leaf, then done
  3. otherwise, subdivide window and apply procedure to each quadrant
  4. attempt to merge the sons of the current node
- Ex:



An 8 by 8 image with 4 nodes

WINDOWING

- General algorithm: check if a node in the target tree is completely contained by a node in the source tree; otherwise, decompose the target tree node and repeat the process and possibly merge later
  1. find smallest subtree of input quadtree containing window
  2. if subtree is a leaf, then done
  3. otherwise, subdivide window and apply procedure to each quadrant
  4. attempt to merge the sons of the current node
- Ex:

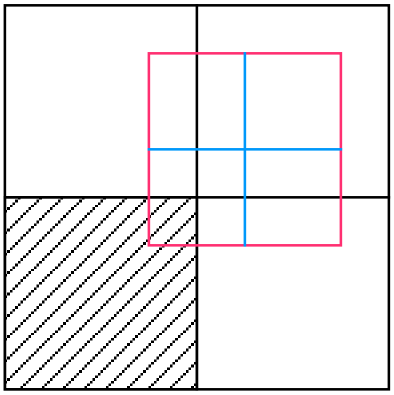


An 8 by 8 image with 4 nodes

A 4 by 4 window initially represented by quadtree of one node

WINDOWING

- General algorithm: check if a node in the target tree is completely contained by a node in the source tree; otherwise, decompose the target tree node and repeat the process and possibly merge later
  1. find smallest subtree of input quadtree containing window
  2. if subtree is a leaf, then done
  3. otherwise, subdivide window and apply procedure to each quadrant
  4. attempt to merge the sons of the current node
- Ex:



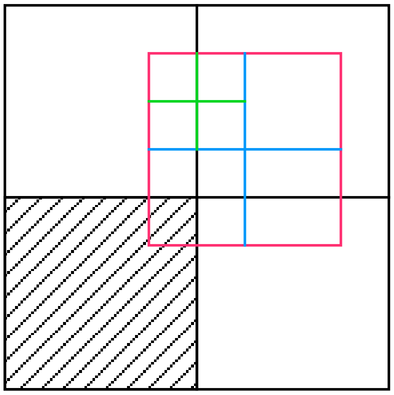
An 8 by 8 image with 4 nodes

A 4 by 4 window initially represented by quadtree of one node

The window is not contained in a single input tree node
⇒ subdivide the window

WINDOWING

- General algorithm: check if a node in the target tree is completely contained by a node in the source tree; otherwise, decompose the target tree node and repeat the process and possibly merge later
  1. find smallest subtree of input quadtree containing window
  2. if subtree is a leaf, then done
  3. otherwise, subdivide window and apply procedure to each quadrant
  4. attempt to merge the sons of the current node
- Ex:



An 8 by 8 image with 4 nodes

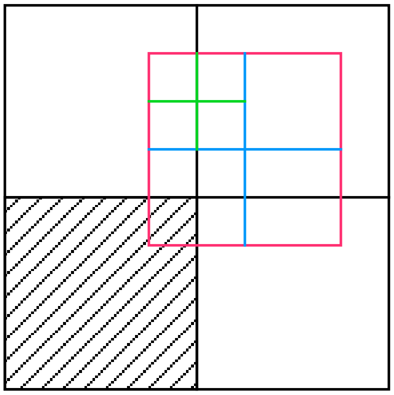
A 4 by 4 window initially represented by quadtree of one node

The window is not contained in a single input tree node
⇒ subdivide the window

NW quadrant is not contained in a single input tree node
⇒ subdivide (these sons are subsequently merged)

WINDOWING

- General algorithm: check if a node in the target tree is completely contained by a node in the source tree; otherwise, decompose the target tree node and repeat the process and possibly merge later
  1. find smallest subtree of input quadtree containing window
  2. if subtree is a leaf, then done
  3. otherwise, subdivide window and apply procedure to each quadrant
  4. attempt to merge the sons of the current node
- Ex:



An 8 by 8 image with 4 nodes

A 4 by 4 window initially represented by quadtree of one node

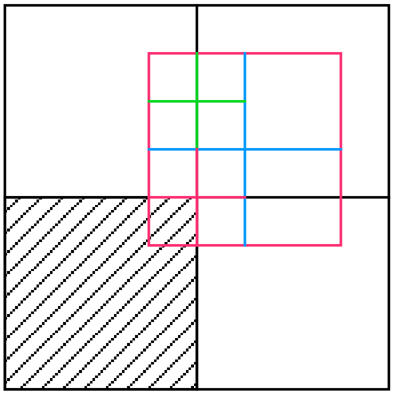
The window is not contained in a single input tree node
⇒ subdivide the window

NW quadrant is not contained in a single input tree node
⇒ subdivide (these sons are subsequently merged)

NE quadrant is contained in a single input tree node
⇒ a leaf

WINDOWING

- General algorithm: check if a node in the target tree is completely contained by a node in the source tree; otherwise, decompose the target tree node and repeat the process and possibly merge later
  1. find smallest subtree of input quadtree containing window
  2. if subtree is a leaf, then done
  3. otherwise, subdivide window and apply procedure to each quadrant
  4. attempt to merge the sons of the current node
- Ex:



An 8 by 8 image with 4 nodes

A 4 by 4 window initially represented by quadtree of one node

The window is not contained in a single input tree node
⇒ subdivide the window

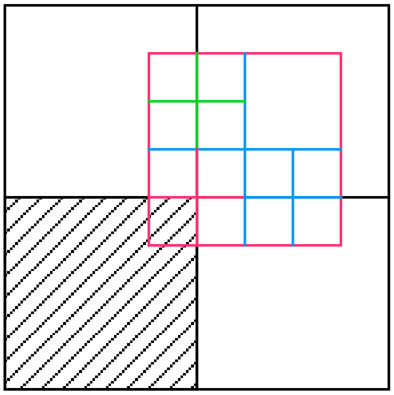
NW quadrant is not contained in a single input tree node
⇒ subdivide (these sons are subsequently merged)

NE quadrant is contained in a single input tree node
⇒ a leaf

SW quadrant is not contained in a single input tree node
⇒ subdivide

WINDOWING

- General algorithm: check if a node in the target tree is completely contained by a node in the source tree; otherwise, decompose the target tree node and repeat the process and possibly merge later
  1. find smallest subtree of input quadtree containing window
  2. if subtree is a leaf, then done
  3. otherwise, subdivide window and apply procedure to each quadrant
  4. attempt to merge the sons of the current node
- Ex:



An 8 by 8 image with 4 nodes

A 4 by 4 window initially represented by quadtree of one node

The window is not contained in a single input tree node
⇒ subdivide the window

NW quadrant is not contained in a single input tree node
⇒ subdivide (these sons are subsequently merged)

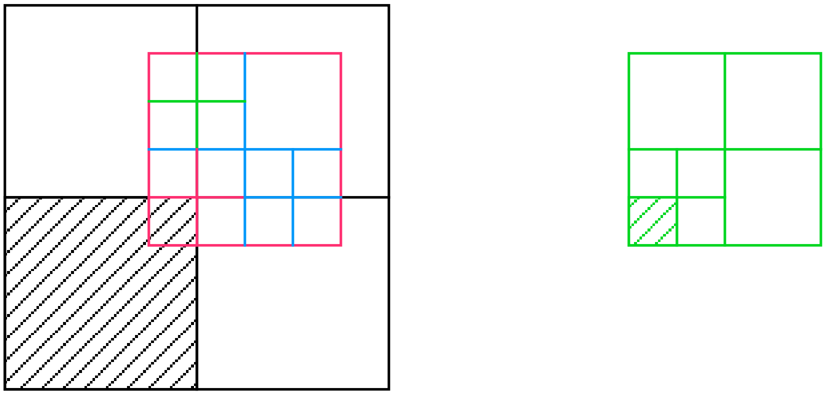
NE quadrant is contained in a single input tree node
⇒ a leaf

SW quadrant is not contained in a single input tree node
⇒ subdivide

SE quadrant is not contained in a single input tree node
⇒ subdivide (these sons are subsequently merged)

WINDOWING

- General algorithm: check if a node in the target tree is completely contained by a node in the source tree; otherwise, decompose the target tree node and repeat the process and possibly merge later
  1. find smallest subtree of input quadtree containing window
  2. if subtree is a leaf, then done
  3. otherwise, subdivide window and apply procedure to each quadrant
  4. attempt to merge the sons of the current node
- Ex:



An 8 by 8 image with 4 nodes

A 4 by 4 window initially represented by quadtree of one node

The window is not contained in a single input tree node
⇒ subdivide the window

NW quadrant is not contained in a single input tree node
⇒ subdivide (these sons are subsequently merged)

NE quadrant is contained in a single input tree node
⇒ a leaf

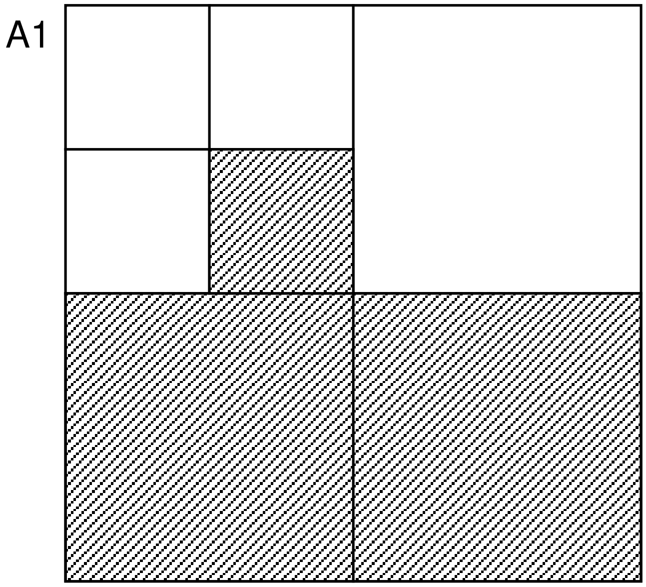
SW quadrant is not contained in a single input tree node
⇒ subdivide

SE quadrant is not contained in a single input tree node
⇒ subdivide (these sons are subsequently merged)

Final output tree

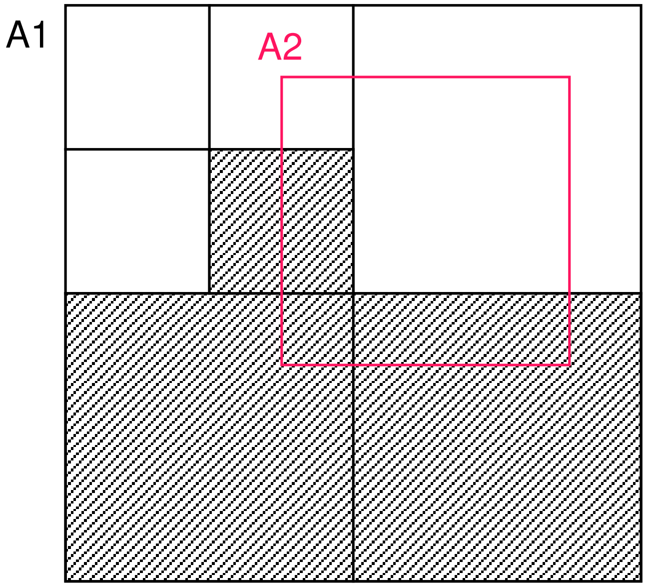
WINDOWING (AN ALTERNATIVE VIEW)

- Def: extract a rectangular subsection of an image and build its quadtree
- Algorithm: analogous to intersecting two unregistered images
  1. A1 is the image from which the window is being extracted
  2. A2 is a BLACK block with the same size and origin as the window
  3. resulting quadtree has the size and position of A2 with the value of the corresponding pixel of A1 in each position
  4. equivalence of the two operations should be clear
- Ex:



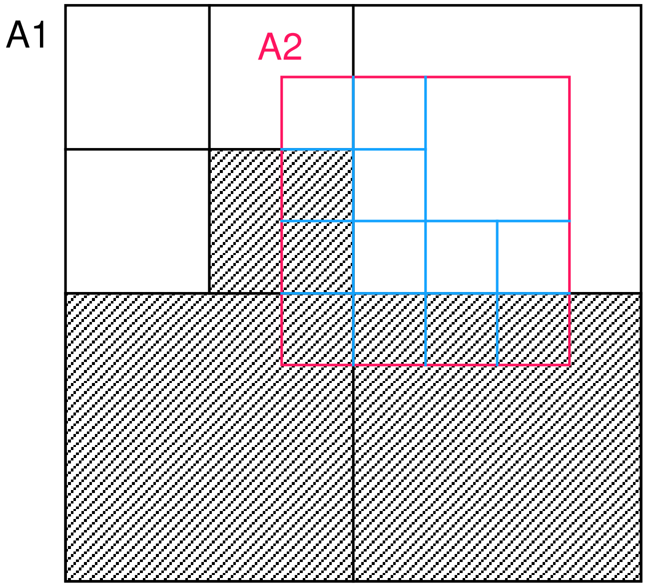
WINDOWING (AN ALTERNATIVE VIEW)

- Def: extract a rectangular subsection of an image and build its quadtree
- Algorithm: analogous to intersecting two unregistered images
  1. A1 is the image from which the window is being extracted
  2. A2 is a BLACK block with the same size and origin as the window
  3. resulting quadtree has the size and position of A2 with the value of the corresponding pixel of A1 in each position
  4. equivalence of the two operations should be clear
- Ex:



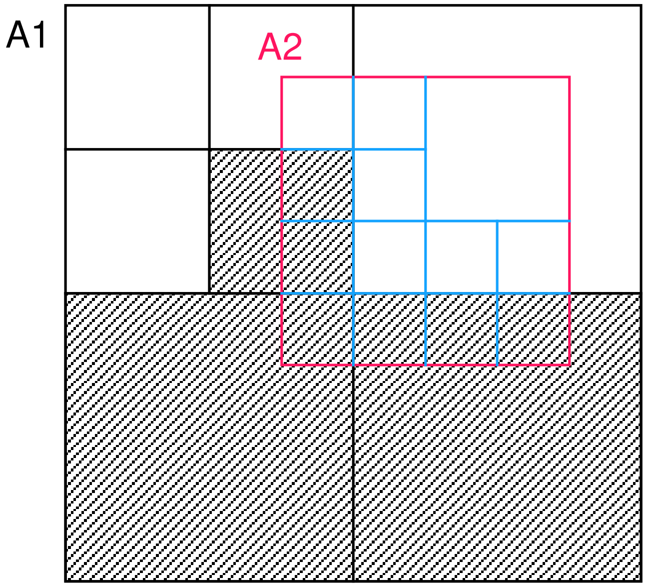
WINDOWING (AN ALTERNATIVE VIEW)

- Def: extract a rectangular subsection of an image and build its quadtree
- Algorithm: analogous to intersecting two unregistered images
  1. A1 is the image from which the window is being extracted
  2. A2 is a BLACK block with the same size and origin as the window
  3. resulting quadtree has the size and position of A2 with the value of the corresponding pixel of A1 in each position
  4. equivalence of the two operations should be clear
- Ex:



WINDOWING (AN ALTERNATIVE VIEW)

- Def: extract a rectangular subsection of an image and build its quadtree
- Algorithm: analogous to intersecting two unregistered images
  1. A1 is the image from which the window is being extracted
  2. A2 is a BLACK block with the same size and origin as the window
  3. resulting quadtree has the size and position of A2 with the value of the corresponding pixel of A1 in each position
  4. equivalence of the two operations should be clear
- Ex:



- The same result can be obtained by taking the union of two unregistered images where A2 is a WHITE block with the same size and origin as the window

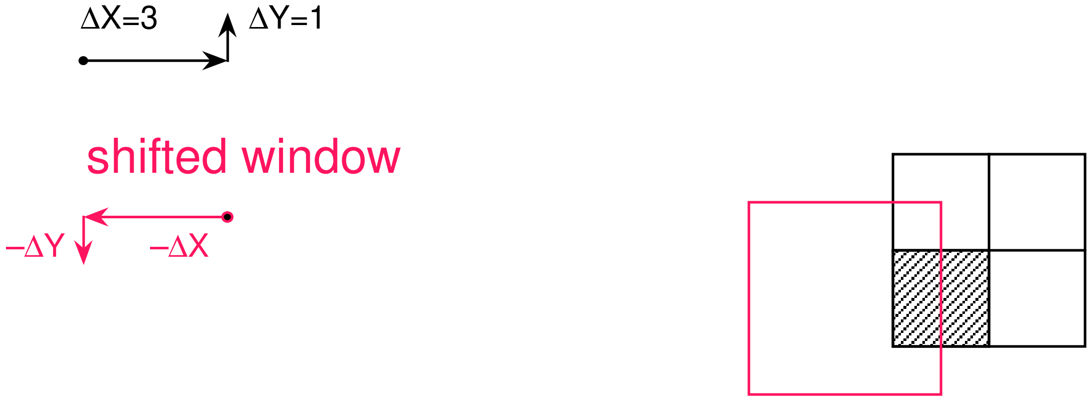
SHIFTING A QUADTREE IMAGE REPRESENTATION

- Equivalent to extracting a window larger than the input image with a different origin
- Shifting an input image by $(\Delta X,\Delta Y)$ implies that the origin of the window is at $(-\Delta X,-\Delta Y)$ from the origin of input image
- Ex: shift by (3,1)

- Alternative algorithm transforms the individual components of the image (Hunter and Steiglitz)
  1. treat components as polygons
  2. trace their boundaries
  3. transform their vertices
  4. construct new polygon for each component by a fill operation
  5. overlay the polygons

SHIFTING A QUADTREE IMAGE REPRESENTATION

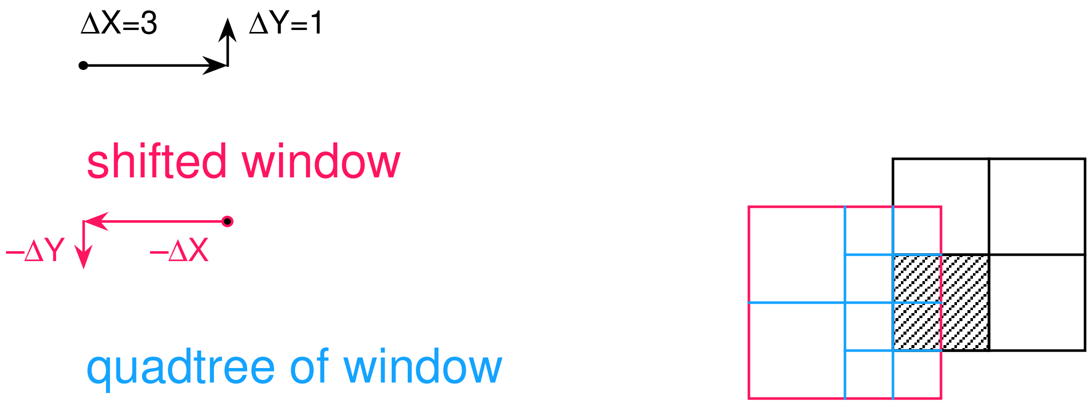
- Equivalent to extracting a window larger than the input image with a different origin
- Shifting an input image by (ΔX,ΔY) implies that the origin of the window is at (–ΔX,–ΔY) from the origin of input image
- Ex: shift by (3,1)



- Alternative algorithm transforms the individual components of the image (Hunter and Steiglitz)
  1. treat components as polygons
  2. trace their boundaries
  3. transform their vertices
  4. construct new polygon for each component by a fill operation
  5. overlay the polygons

SHIFTING A QUADTREE IMAGE REPRESENTATION

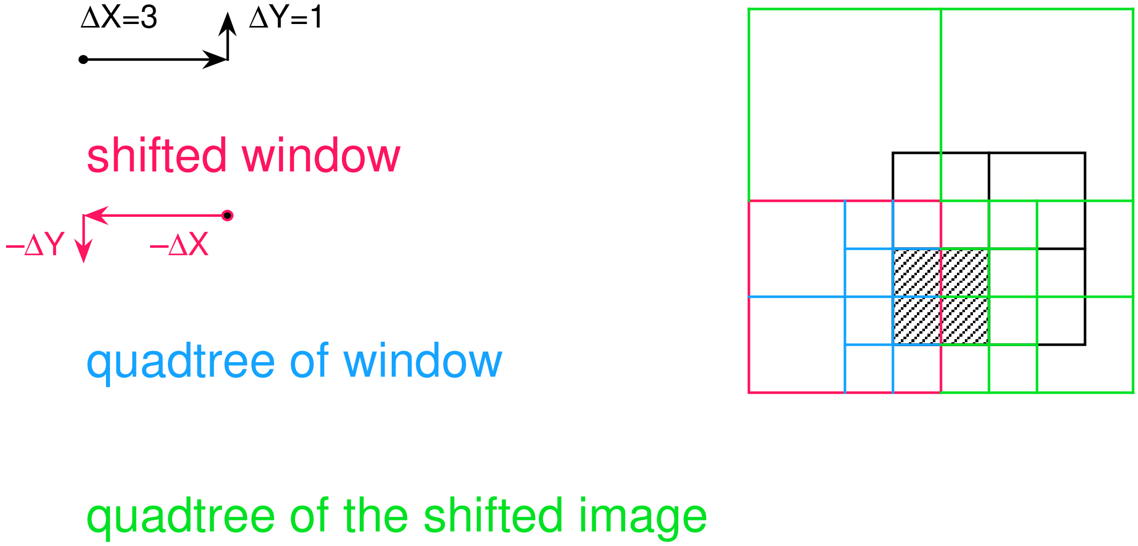
- Equivalent to extracting a window larger than the input image with a different origin
- Shifting an input image by (ΔX,ΔY) implies that the origin of the window is at (–ΔX,–ΔY) from the origin of input image
- Ex: shift by (3,1)



- Alternative algorithm transforms the individual components of the image (Hunter and Steiglitz)
  1. treat components as polygons
  2. trace their boundaries
  3. transform their vertices
  4. construct new polygon for each component by a fill operation
  5. overlay the polygons

SHIFTING A QUADTREE IMAGE REPRESENTATION

- Equivalent to extracting a window larger than the input image with a different origin
- Shifting an input image by (ΔX,ΔY) implies that the origin of the window is at (–ΔX,–ΔY) from the origin of input image
- Ex: shift by (3,1)



shifted window

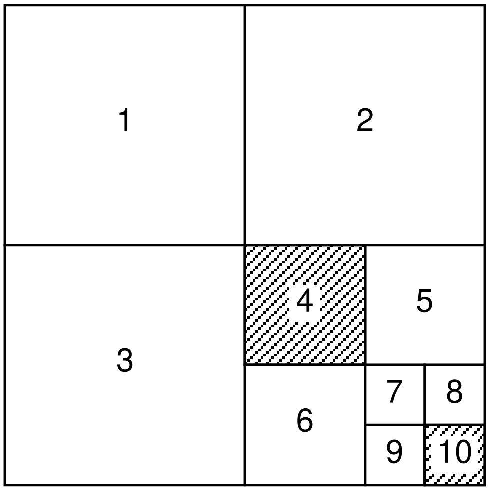
quadtree of window

quadtree of the shifted image

- Alternative algorithm transforms the individual components of the image (Hunter and Steiglitz)
  1. treat components as polygons
  2. trace their boundaries
  3. transform their vertices
  4. construct new polygon for each component by a fill operation
  5. overlay the polygons

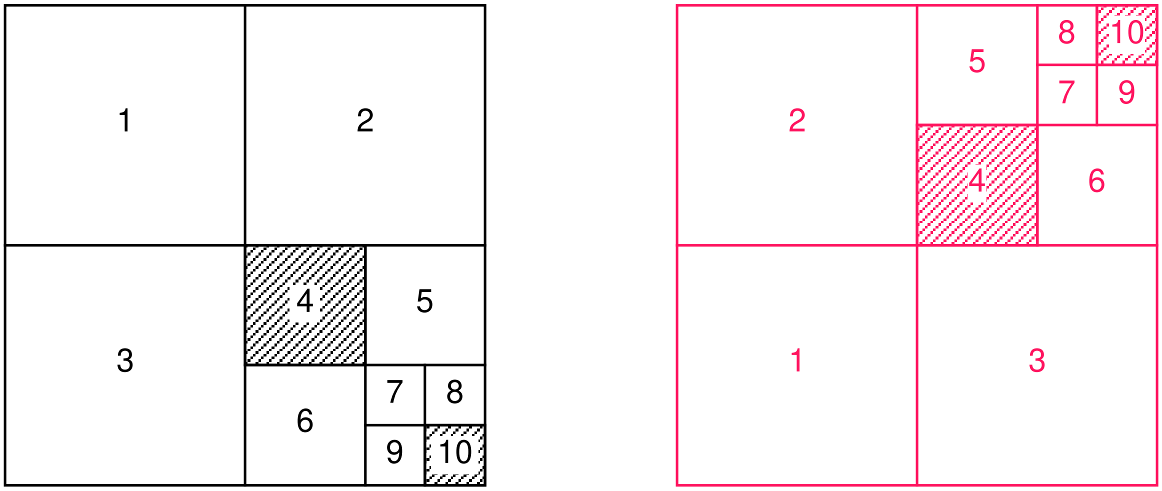
ROTATING A QUADTREE IMAGE REPRESENTATION

- Easy for multiples of 90 degrees
- Harder for others because result is an approximation — use a more general linear transformation algorithm
- Ex: Rotate by 90° counterclockwise



ROTATING A QUADTREE IMAGE REPRESENTATION

- Easy for multiples of 90 degrees
- Harder for others because result is an approximation — use a more general linear transformation algorithm
- Ex: Rotate by 90° counterclockwise

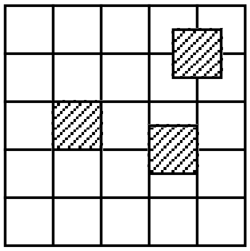


- Correspondence of sons:

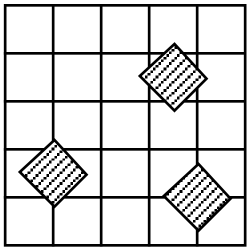
OLD	→	NEW
NW		SW
NE		NW
SW		SE
SE		NE

LINEAR IMAGE TRANSFORMATIONS

- Source tree and target tree
- General window algorithm checks if a node in the target tree is completely contained by a node in the source tree; otherwise, it decomposes the target tree node and repeats the process and possibly merges later
- Merges can be avoided by checking if the target tree node is completely covered by black nodes in the source tree
- If transformation only involves translation, then a quadtree (octree) block can be covered by at most 4 (8) blocks of greater than equal size



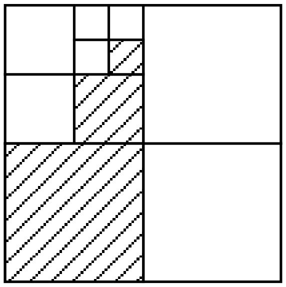
- If transformation involves rotation as well as translation, then a quadtree (octree) block can be covered by at most 6 (18) blocks of greater than equal size



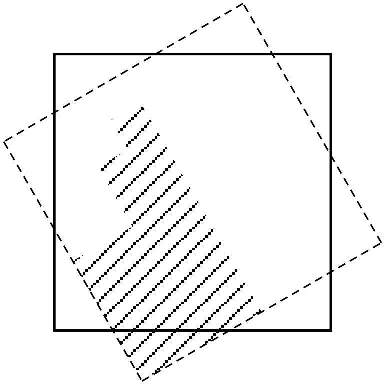
- If transformation involves scaling, then the number of covering blocks depends on the scale factor

EXAMPLE OF A LINEAR IMAGE TRANSFORMATION

- Ex:

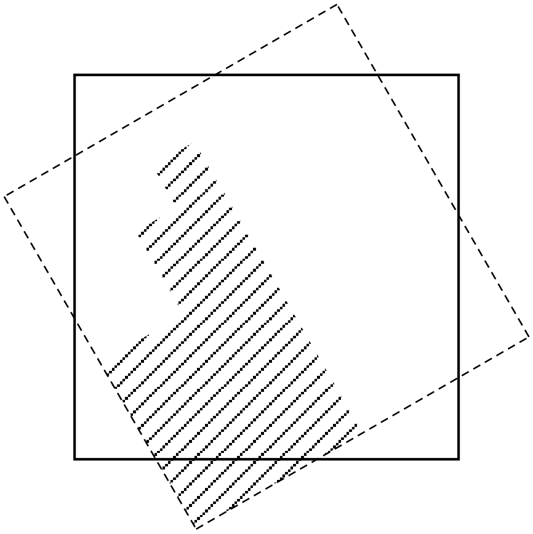


source tree



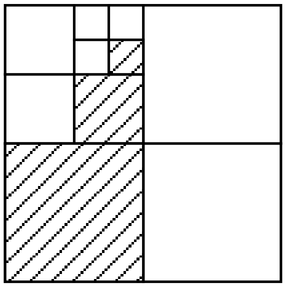
result of rotating by 30 degrees
counterclockwise about the origin

- Source tree space is shown with broken lines
- Target tree space is shown with solid lines
- Figure shows the result of the rotation in the original space
- ? denotes target tree node not completely covered by WHITE or BLACK nodes in the source tree; the image is assumed to be bordered by WHITE nodes

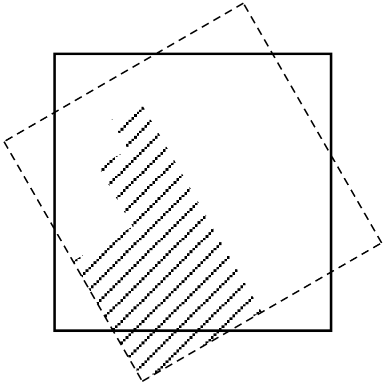


EXAMPLE OF A LINEAR IMAGE TRANSFORMATION

- Ex:

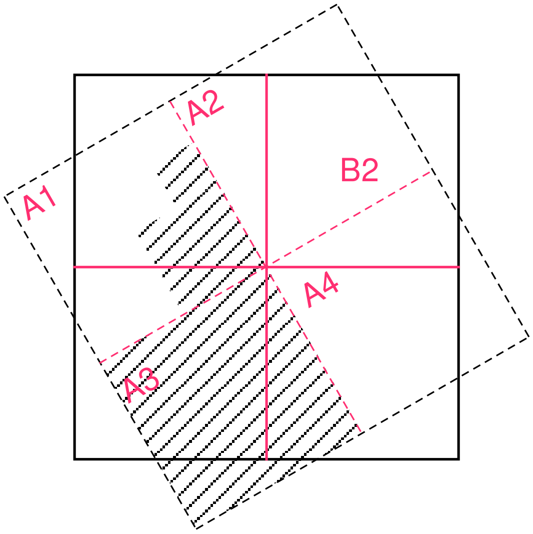


source tree



result of rotating by 30 degrees counterclockwise about the origin

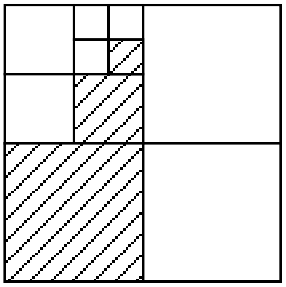
- Source tree space is shown with broken lines
- Target tree space is shown with solid lines
- Figure shows the result of the rotation in the original space
- ? denotes target tree node not completely covered by WHITE or BLACK nodes in the source tree; the image is assumed to be bordered by WHITE nodes



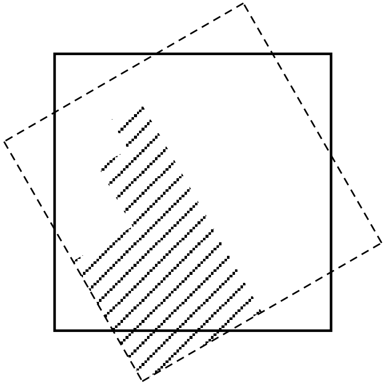
- Decompose the source and the target trees since the target is not covered by a BLACK or a WHITE node in source

EXAMPLE OF A LINEAR IMAGE TRANSFORMATION

- Ex:

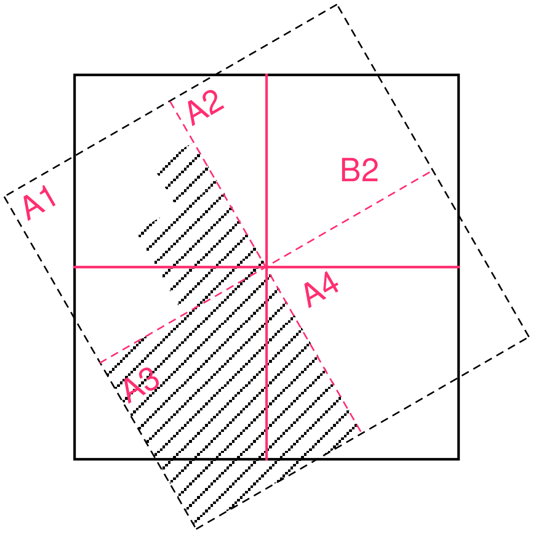


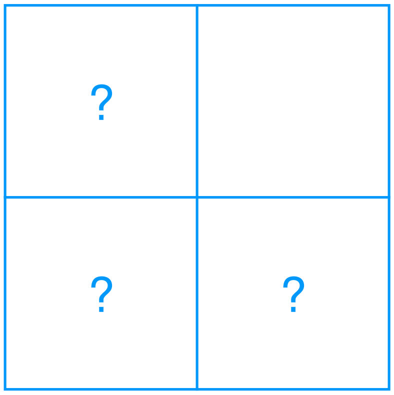
source tree



result of rotating by 30 degrees counterclockwise about the origin

- Source tree space is shown with broken lines
- Target tree space is shown with solid lines
- Figure shows the result of the rotation in the original space
- ? denotes target tree node not completely covered by WHITE or BLACK nodes in the source tree; the image is assumed to be bordered by WHITE nodes

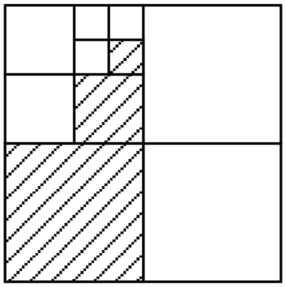




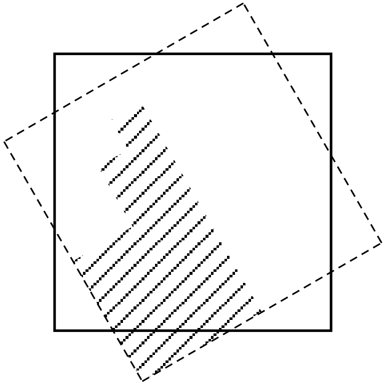
- Decompose the source and the target trees since the target is not covered by a BLACK or a WHITE node in source
- B2 is WHITE because blocks A2 and A4 that cover it are WHITE

EXAMPLE OF A LINEAR IMAGE TRANSFORMATION

- Ex:

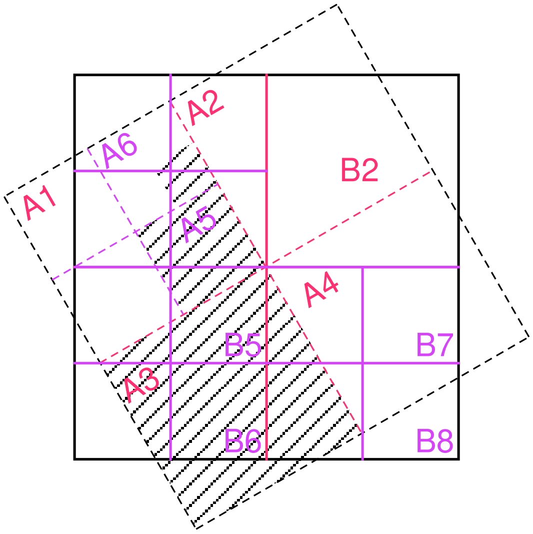


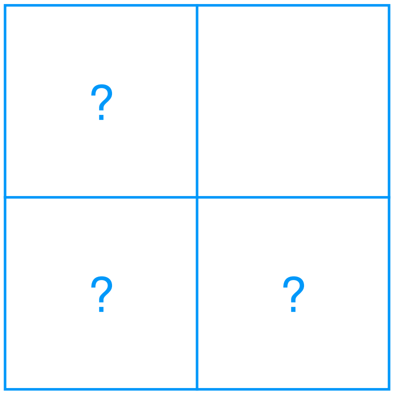
source tree



result of rotating by 30 degrees counterclockwise about the origin

- Source tree space is shown with broken lines
- Target tree space is shown with solid lines
- Figure shows the result of the rotation in the original space
- ? denotes target tree node not completely covered by WHITE or BLACK nodes in the source tree; the image is assumed to be bordered by WHITE nodes

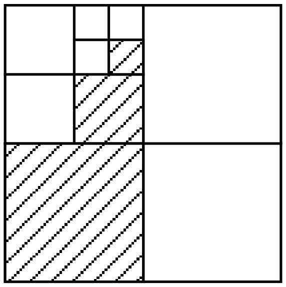




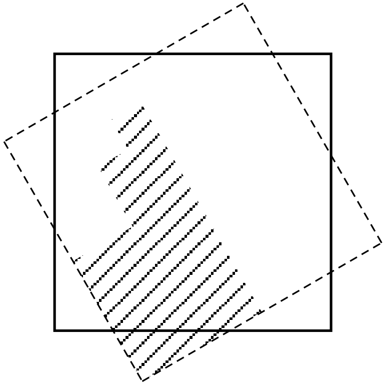
- Decompose the source and the target trees since the target is not covered by a BLACK or a WHITE node in source
- B2 is WHITE because blocks A2 and A4 that cover it are WHITE
- Decompose A1 in the source and ? in the target trees

EXAMPLE OF A LINEAR IMAGE TRANSFORMATION

- Ex:

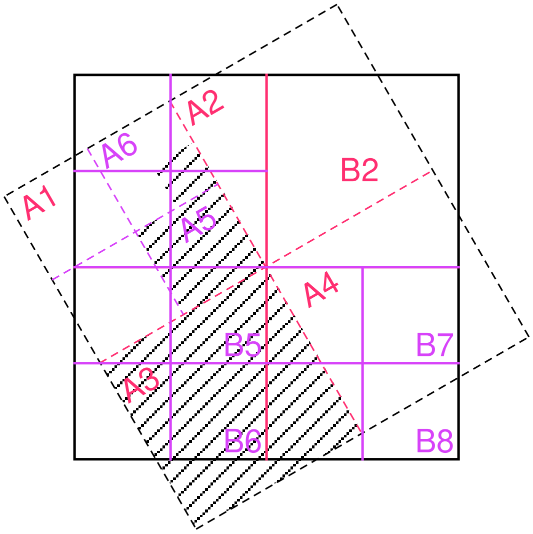


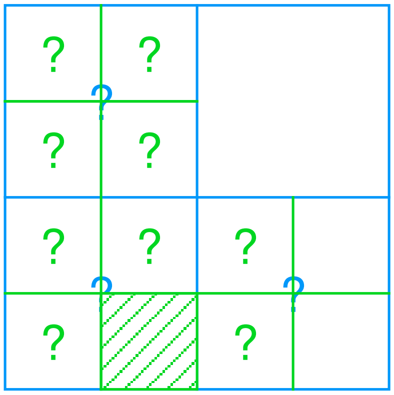
source tree



result of rotating by 30 degrees counterclockwise about the origin

- Source tree space is shown with broken lines
- Target tree space is shown with solid lines
- Figure shows the result of the rotation in the original space
- ? denotes target tree node not completely covered by WHITE or BLACK nodes in the source tree; the image is assumed to be bordered by WHITE nodes

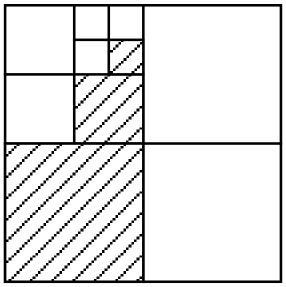




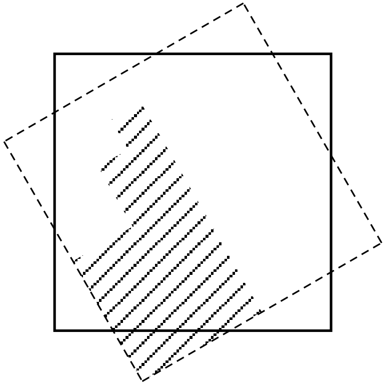
- Decompose the source and the target trees since the target is not covered by a BLACK or a WHITE node in source
- B2 is WHITE because blocks A2 and A4 that cover it are WHITE
- Decompose A1 in the source and ? in the target trees
- B6 (B7 and B8) is BLACK (WHITE) because block A3 (A4) that cover it is BLACK (WHITE) while B5 is not completely BLACK

EXAMPLE OF A LINEAR IMAGE TRANSFORMATION

- Ex:

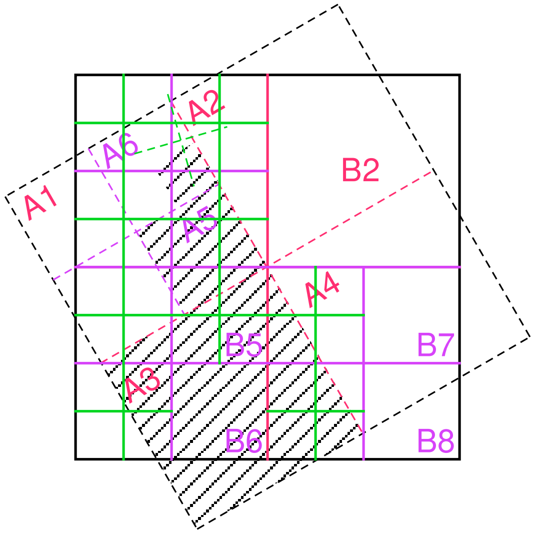


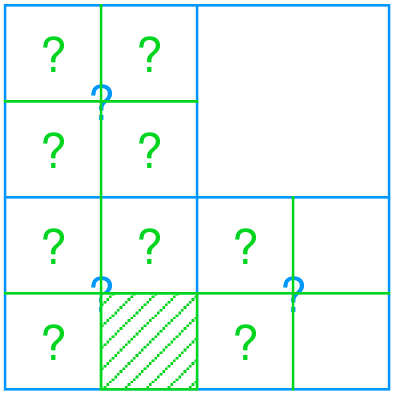
source tree



result of rotating by 30 degrees counterclockwise about the origin

- Source tree space is shown with broken lines
- Target tree space is shown with solid lines
- Figure shows the result of the rotation in the original space
- ? denotes target tree node not completely covered by WHITE or BLACK nodes in the source tree; the image is assumed to be bordered by WHITE nodes

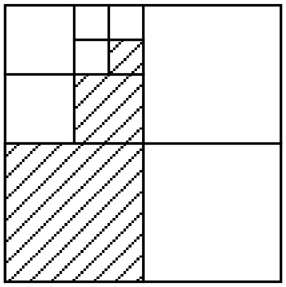




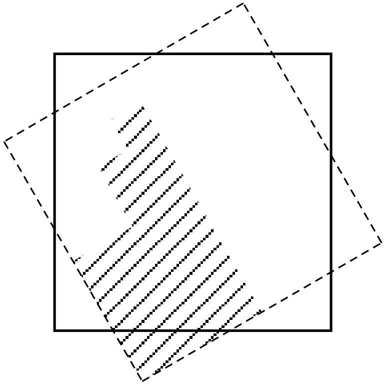
- Decompose the source and the target trees since the target is not covered by a BLACK or a WHITE node in source
- B2 is WHITE because blocks A2 and A4 that cover it are WHITE
- Decompose A1 in the source and ? in the target trees
- B6 (B7 and B8) is BLACK (WHITE) because block A3 (A4) that cover it is BLACK (WHITE) while B5 is not completely BLACK
- Decompose A6 in the source and ? in the target trees

EXAMPLE OF A LINEAR IMAGE TRANSFORMATION

- Ex:

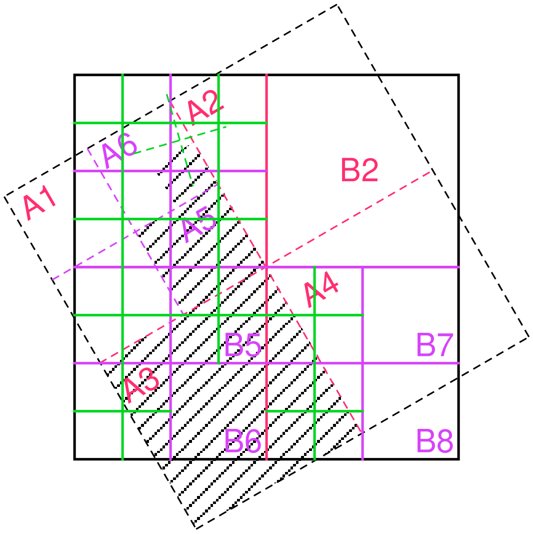


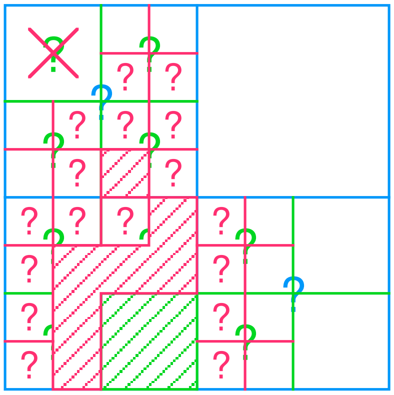
source tree



result of rotating by 30 degrees counterclockwise about the origin

- Source tree space is shown with broken lines
- Target tree space is shown with solid lines
- Figure shows the result of the rotation in the original space
- ? denotes target tree node not completely covered by WHITE or BLACK nodes in the source tree; the image is assumed to be bordered by WHITE nodes





- Decompose the source and the target trees since the target is not covered by a BLACK or a WHITE node in source
- B2 is WHITE because blocks A2 and A4 that cover it are WHITE
- Decompose A1 in the source and ? in the target trees
- B6 (B7 and B8) is BLACK (WHITE) because block A3 (A4) that cover it is BLACK (WHITE) while B5 is not completely BLACK
- Decompose A6 in the source and ? in the target trees
- Result after merging 4 WHITE sons in NW son of NW quadrant

REGION EXPANSION

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

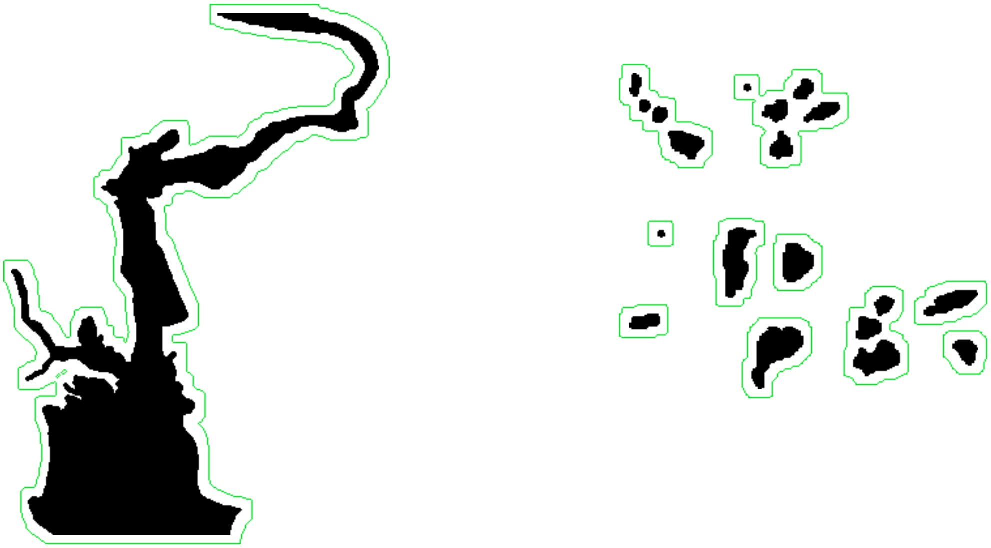
SPATIAL RANGE QUERIES

- Ex: find all objects within 10 miles of St. Louis, the Mississippi River, Lake Michigan, ...

- Analog of range query in conventional databases
  1. ex: find all people with height between 1.5 and 1.8 meters and weight between 60 and 70 kilos
  2. difference is that the range query is not rectangular
  3. instead, region is spatially defined
- Known as a buffer or a corridor in GIS, image dilation or region expansion in image processing
- Can be computed very efficiently for a quadtree by using the L_∞ (i.e., chessboard) distance metric since locus of all points within a given distance is a square
  1. $\text{dist}(a,b) = \max(|a_x - b_x|,|a_y - b_y|)$

SPATIAL RANGE QUERIES

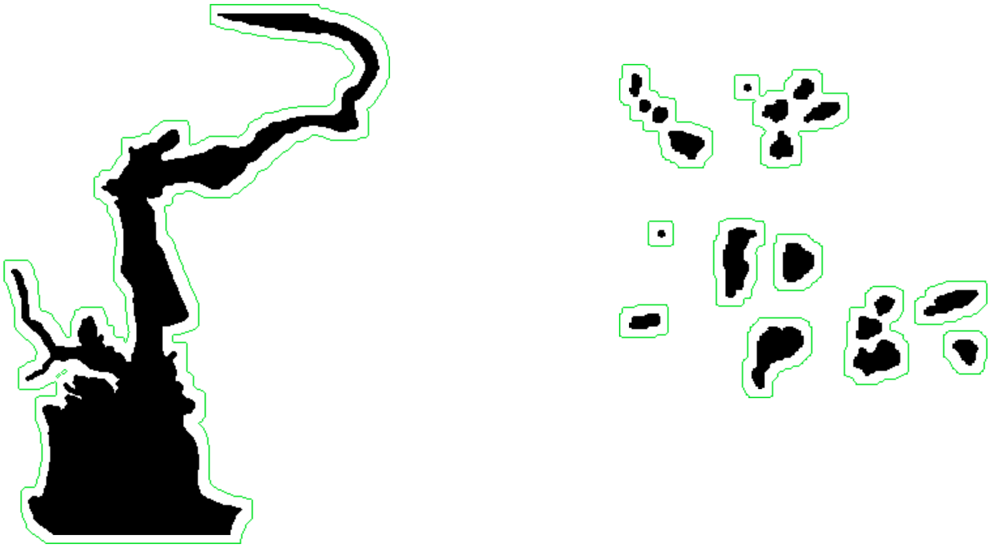
- Ex: find all objects within 10 miles of St. Louis, the Mississippi River, Lake Michigan, ...



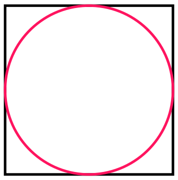
- Analog of range query in conventional databases
  1. ex: find all people with height between 1.5 and 1.8 meters and weight between 60 and 70 kilos
  2. difference is that the range query is not rectangular
  3. instead, region is spatially defined
- Known as a buffer or a corridor in GIS, image dilation or region expansion in image processing
- Can be computed very efficiently for a quadtree by using the L_∞ (i.e., chessboard) distance metric since locus of all points within a given distance is a square
  1. $\mathrm{dist}(a,b) = \max(|a_x - b_x|,|a_y - b_y|)$

SPATIAL RANGE QUERIES

- Ex: find all objects within 10 miles of St. Louis, the Mississippi River, Lake Michigan, ...



- Analog of range query in conventional databases
  1. ex: find all people with height between 1.5 and 1.8 meters and weight between 60 and 70 kilos
  2. difference is that the range query is not rectangular
  3. instead, region is spatially defined
- Known as a buffer or a corridor in GIS, image dilation or region expansion in image processing
- Can be computed very efficiently for a quadtree by using the L_∞ (i.e., chessboard) distance metric since locus of all points within a given distance is a square
  1. $\text{dist}(a,b) = \max(|a_x - b_x|,|a_y - b_y|)$
  2. an overapproximation to L_2 (Euclidean) distance metric — i.e., $\text{dist}(a,b) = \sqrt{(a_x - b_x)^2 + (a_y - b_y)^2}$

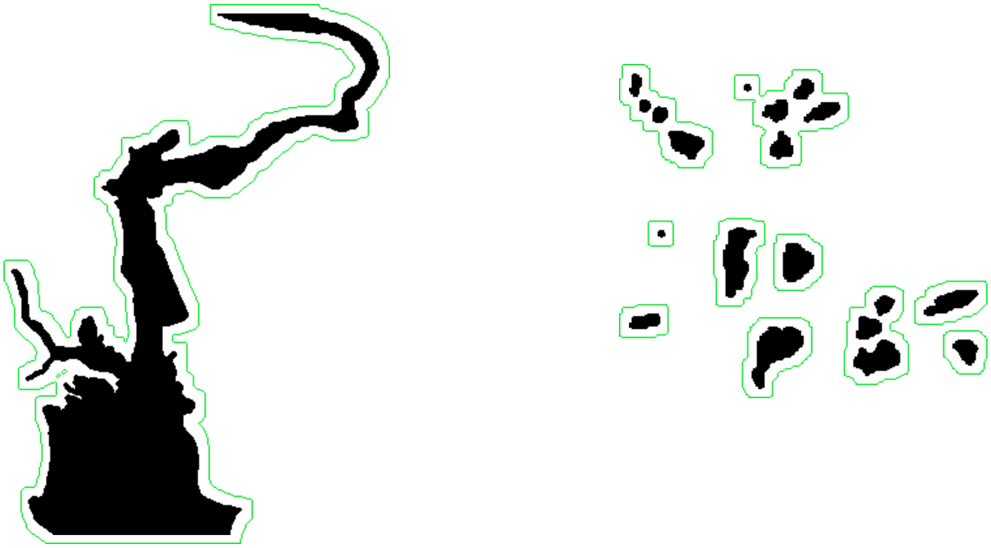


4	3	2	1
z	r	g	b

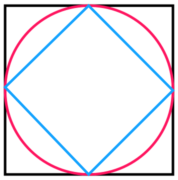
bf1

SPATIAL RANGE QUERIES

- Ex: find all objects within 10 miles of St. Louis, the Mississippi River, Lake Michigan, ...

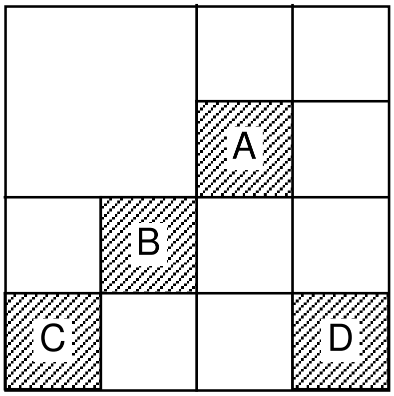


- Analog of range query in conventional databases
  1. ex: find all people with height between 1.5 and 1.8 meters and weight between 60 and 70 kilos
  2. difference is that the range query is not rectangular
  3. instead, region is spatially defined
- Known as a buffer or a corridor in GIS, image dilation or region expansion in image processing
- Can be computed very efficiently for a quadtree by using the L_∞ (i.e., chessboard) distance metric since locus of all points within a given distance is a square
  1. $\mathrm{dist}(a,b) = \max(|a_x - b_x|, |a_y - b_y|)$
  2. an overapproximation to L_2 (Euclidean) distance metric — i.e., $\mathrm{dist}(a,b) = \sqrt{(a_x - b_x)^2 + (a_y - b_y)^2}$
  3. compare with the L_1 (i.e., city block) distance metric which underapproximates the Euclidean distance — i.e., $\mathrm{dist}(a,b) = |a_x - b_x| + |a_y - b_y|$



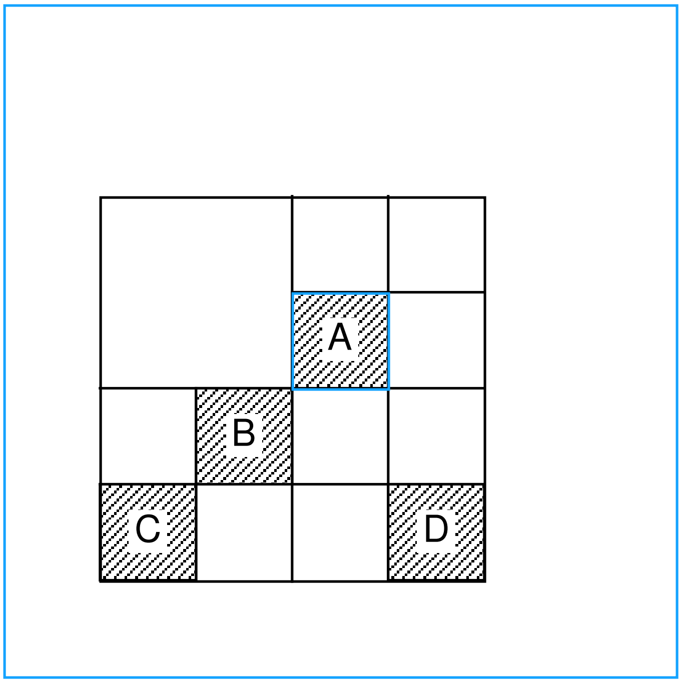
BUFFER1

- Algorithm:
  1. expand each BLACK B block by radius *R*
  2. decompose the resulting square into quadtree blocks
  3. insert the blocks one-by-one into the tree
- Drawbacks
  1. excessive redundant insertion
  2. excessive merging
- Ex: expand using *R*=3



BUFFER1

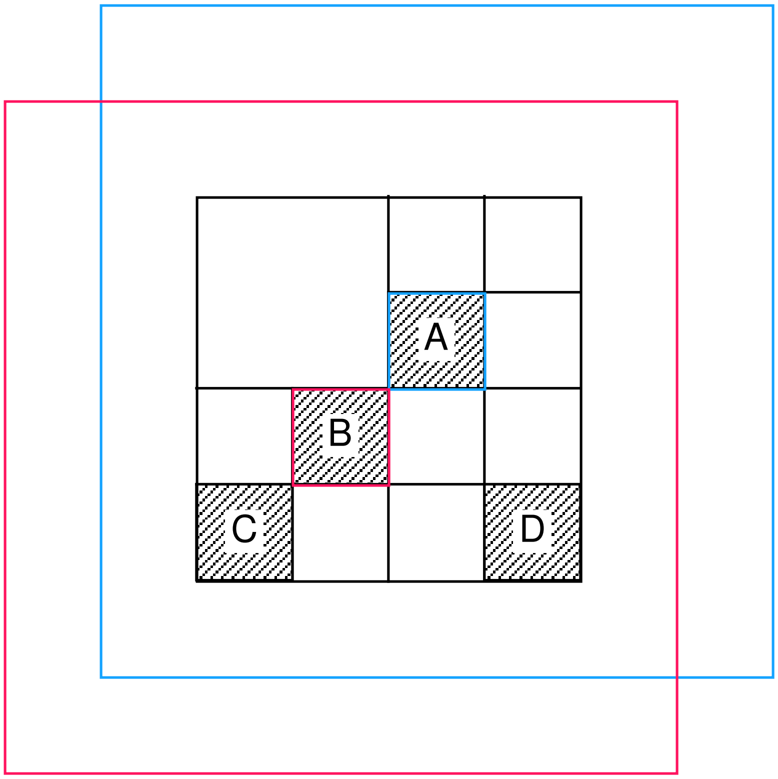
- Algorithm:
  1. expand each BLACK B block by radius *R*
  2. decompose the resulting square into quadtree blocks
  3. insert the blocks one-by-one into the tree
- Drawbacks
  1. excessive redundant insertion
  2. excessive merging
- Ex: expand using *R*=3



expanding A

BUFFER1

- Algorithm:
  1. expand each BLACK B block by radius R
  2. decompose the resulting square into quadtree blocks
  3. insert the blocks one-by-one into the tree
- Drawbacks
  1. excessive redundant insertion
  2. excessive merging
- Ex: expand using $R=3$

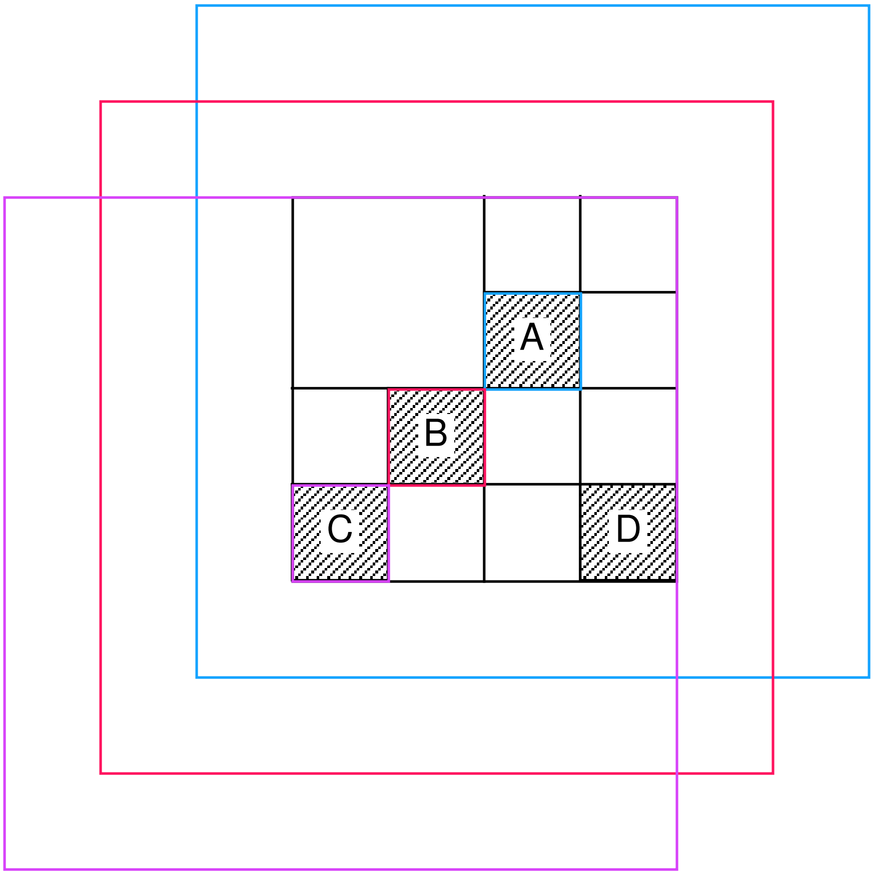


expanding A

expanding B

BUFFER1

- Algorithm:
  1. expand each BLACK B block by radius *R*
  2. decompose the resulting square into quadtree blocks
  3. insert the blocks one-by-one into the tree
- Drawbacks
  1. excessive redundant insertion
  2. excessive merging
- Ex: expand using *R*=3

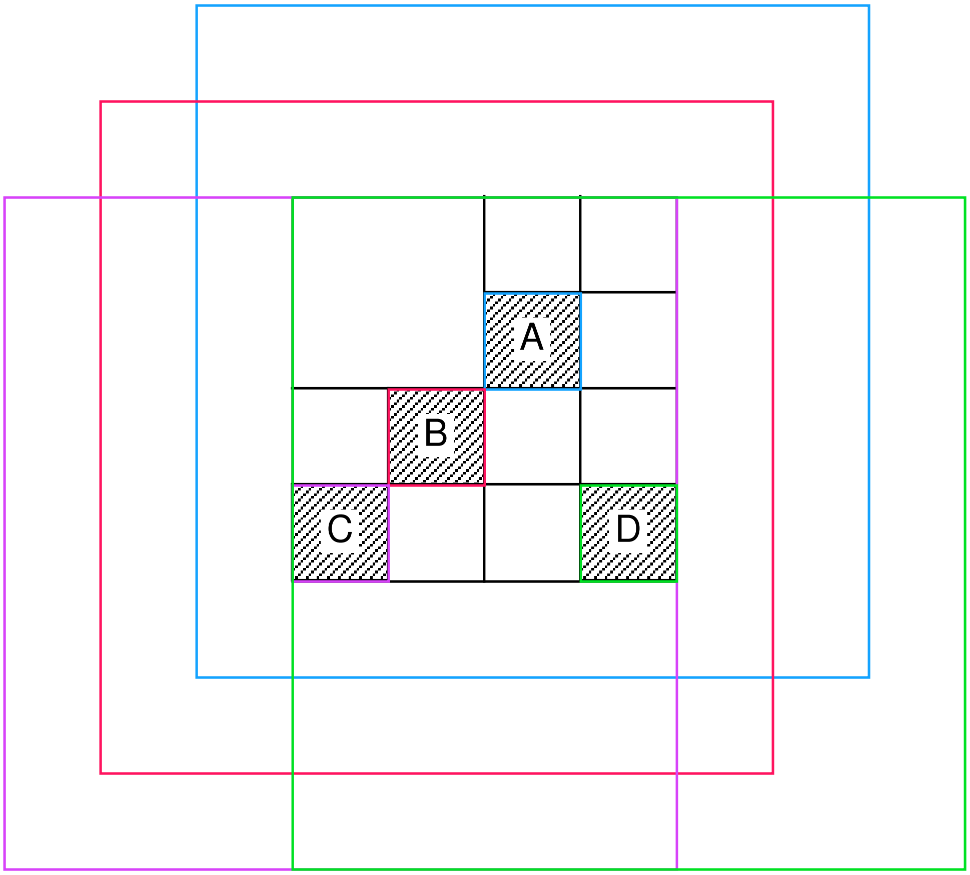


expanding A
expanding B
expanding C

5	4	3	2	1
g	v	r	z	b

BUFFER1

- Algorithm:
  1. expand each BLACK B block by radius *R*
  2. decompose the resulting square into quadtree blocks
  3. insert the blocks one-by-one into the tree
- Drawbacks
  1. excessive redundant insertion
  2. excessive merging
- Ex: expand using *R*=3



expanding A

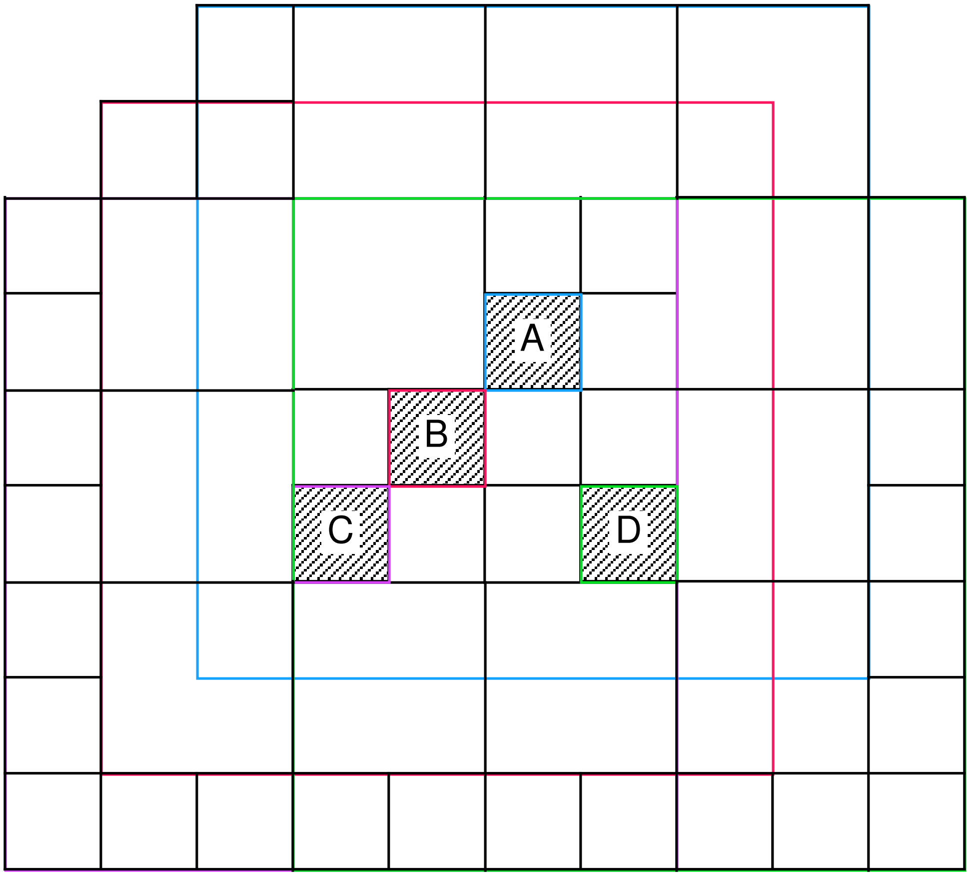
expanding B

expanding C

expanding D

BUFFER1

- Algorithm:
  1. expand each BLACK B block by radius *R*
  2. decompose the resulting square into quadtree blocks
  3. insert the blocks one-by-one into the tree
- Drawbacks
  1. excessive redundant insertion
  2. excessive merging
- Ex: expand using *R*=3



expanding A

expanding B

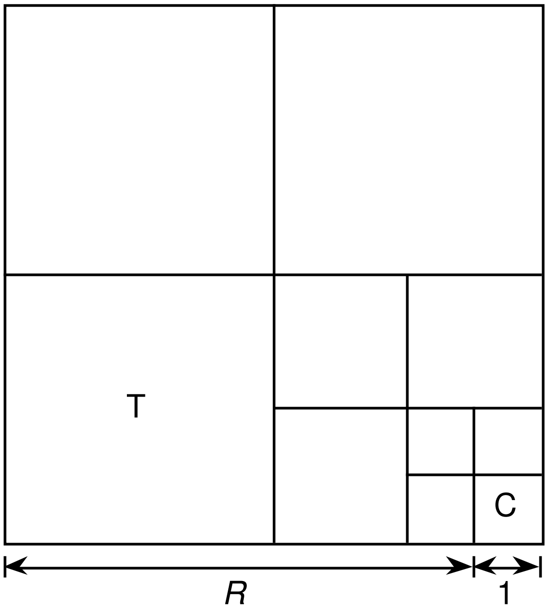
expanding C

expanding D

result is the union of the individual expansions

OBSERVATION

- Any WHITE block T of width ≤ $(R+1)/2$ will be BLACK as a result of expansion by radius R
- Rationale:
  1. if block T has width ≤ $(R+1)/2$, then at least one of T's brothers will have a non-WHITE descendant C such that all of T is within R of a point on C's boundary
  2. therefore, T and all of its brothers must be BLACK
  3. can simply replace T's father by a BLACK block



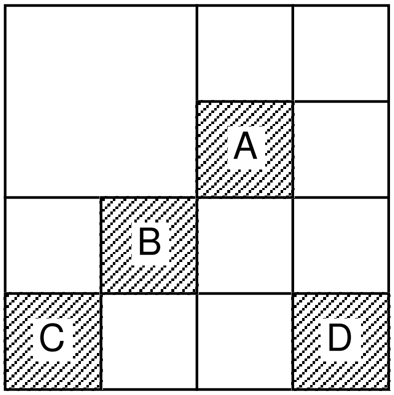
- Analogous property holds for any nonleaf node of width ≤ $(R+1)/2$ — i.e., it will be BLACK as a result of expansion by radius R

IMPROVING ON BUFFER1 TO YIELD BUFFER2

1. Reduce number of blocks that must be inserted as a result of expansion
   - introduce concept of merging cluster
     a. a nonleaf node of width ≤ R+1
     b. such nodes can be replaced by BLACK blocks
2. reduce number of BLACK blocks that need to be considered for expansion
   - introduce concept of vertex set
     a. only a subset of each merging cluster affect the expansion
     b. indicates which of the 8 possible expansion directions of the relevant elements of the merging cluster need expansion

MERGING CLUSTER

- Merging clusters are blocks that correspond to non-leaf nodes of width W(R) where W(R) = 2^r, ($2^r \leq R+1 < 2^{r+1}$)
- Expansion by radius R implies that all merging clusters become BLACK blocks
- Rationale:
  1. blocks corresponding to merging clusters are nonleaf nodes
  2. they must have at least one son that is WHITE or a nonleaf node with width ≤ $(R+1)/2$
- Ex: merging cluster for R=3



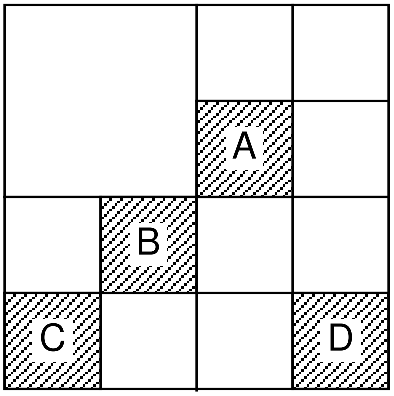
W(R) = 4

9 WHITE blocks

4 BLACK blocks

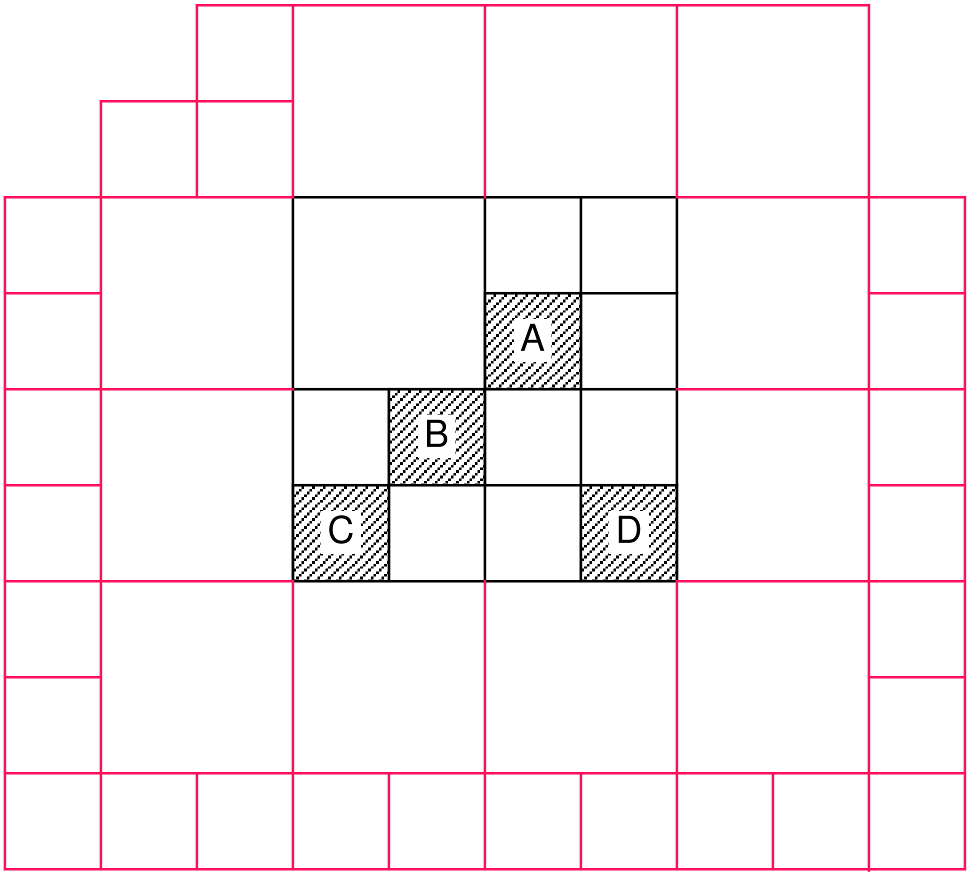
VERTEX SET

- Vertex set of merging cluster *M* consists of four subsets (one for each of the diagonal directions) of vertices of blocks which must be expanded to yield the area spanned by the merging cluster
- Ex: merging cluster with R=3 and the result of its expansion



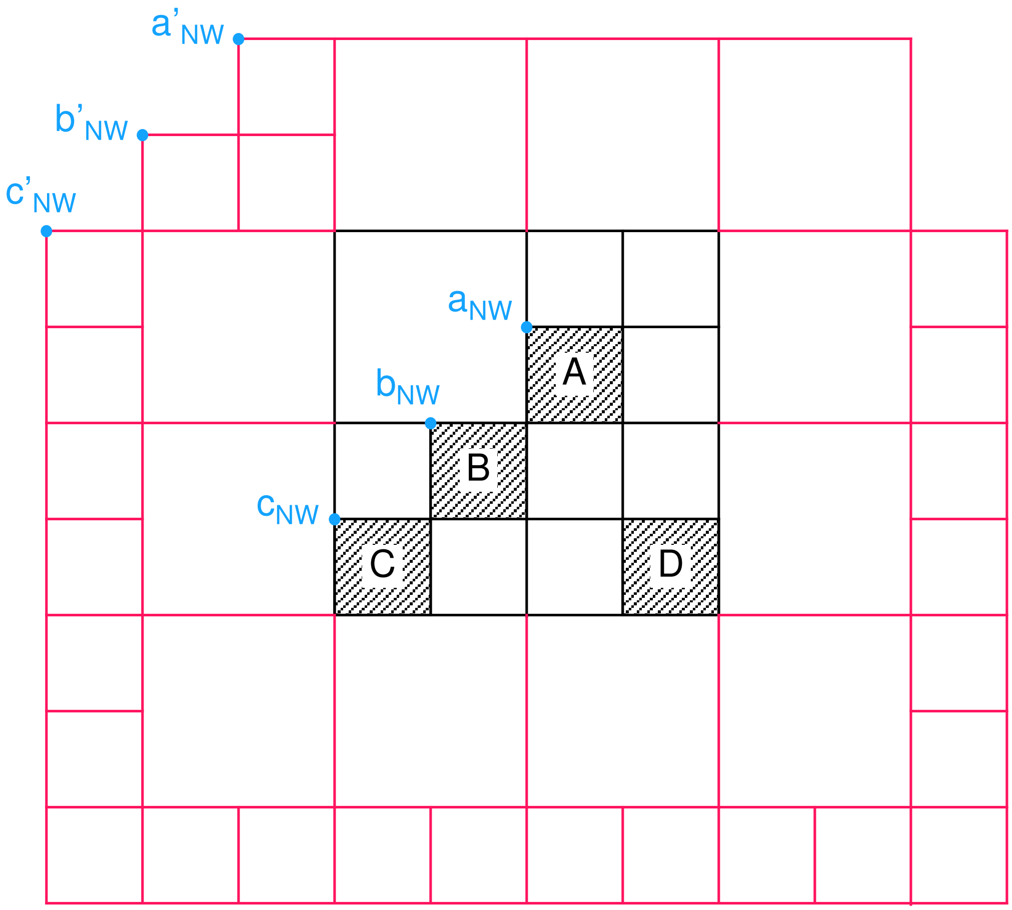
VERTEX SET

- Vertex set of merging cluster *M* consists of four subsets (one for each of the diagonal directions) of vertices of blocks which must be expanded to yield the area spanned by the merging cluster
- Ex: merging cluster with *R*=3 and the result of its expansion



VERTEX SET

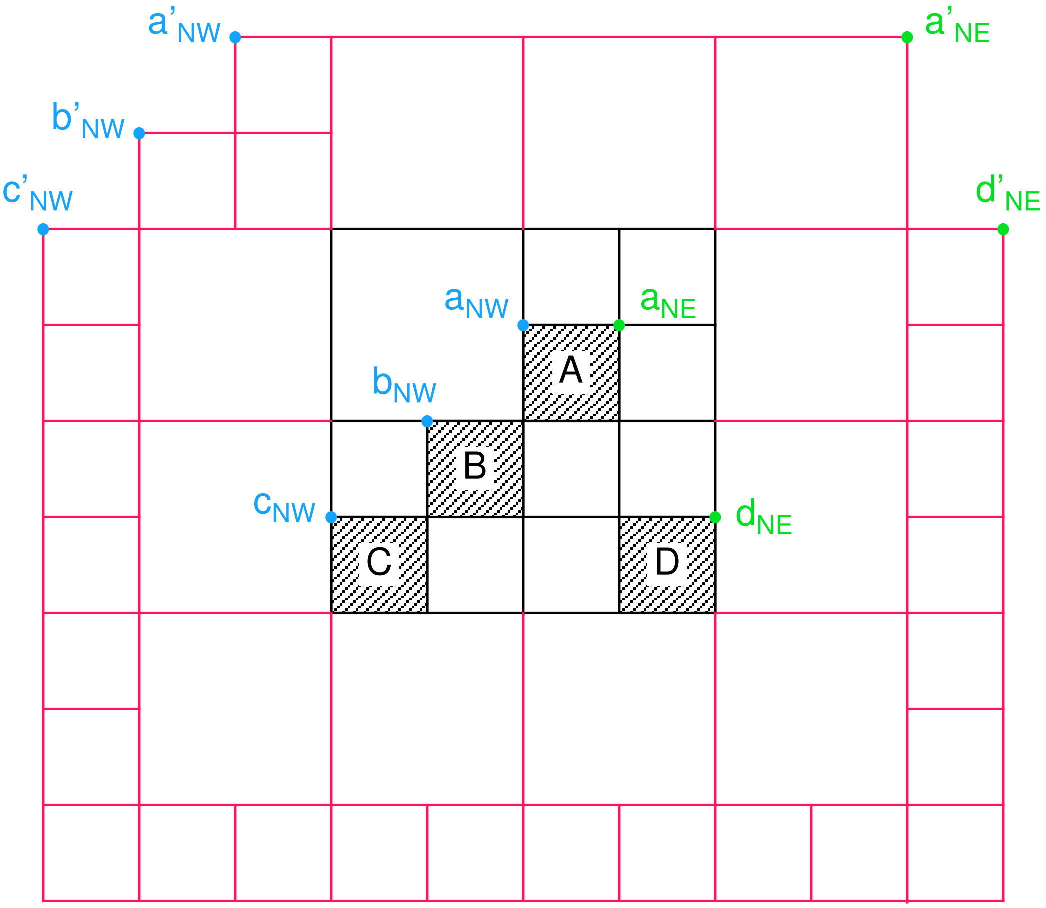
- Vertex set of merging cluster *M* consists of four subsets (one for each of the diagonal directions) of vertices of blocks which must be expanded to yield the area spanned by the merging cluster
- Ex: merging cluster with *R*=3 and the result of its expansion



$VS_{NW} = \{ a_{NW} , b_{NW} , c_{NW} \}$

VERTEX SET

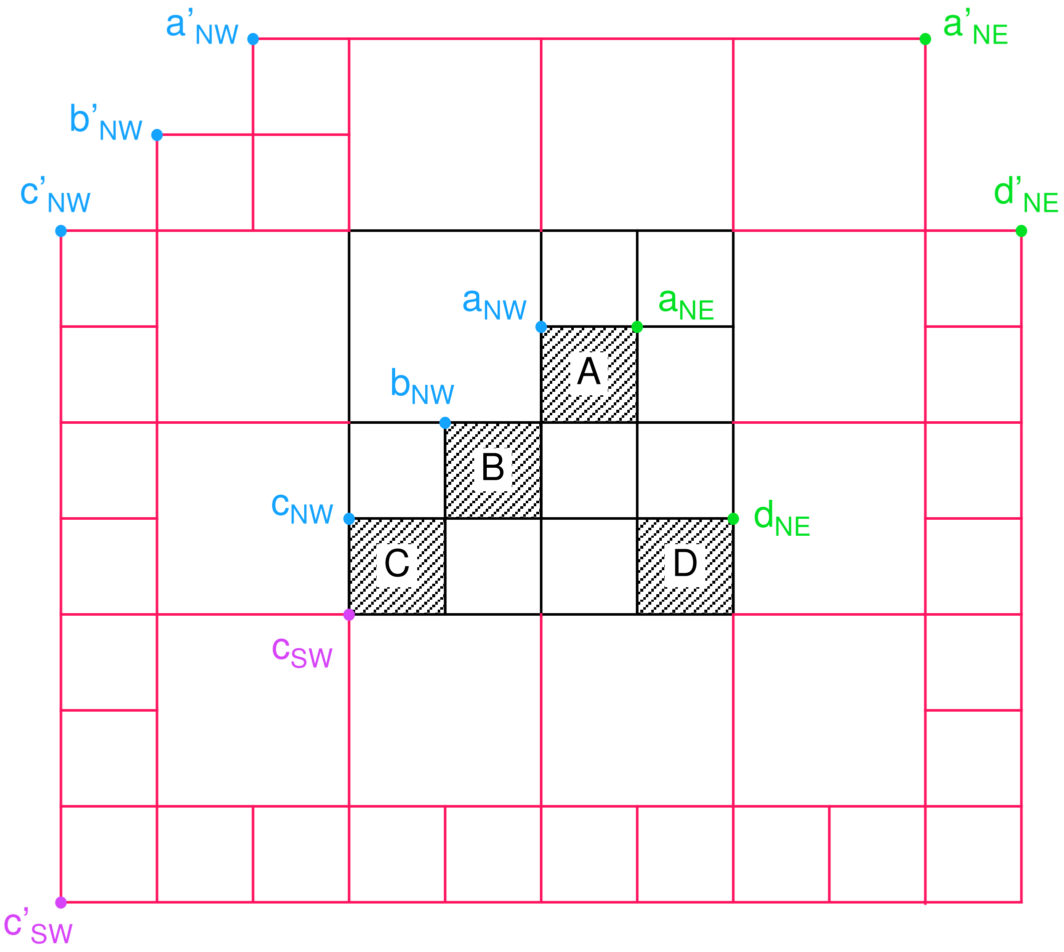
- Vertex set of merging cluster *M* consists of four subsets (one for each of the diagonal directions) of vertices of blocks which must be expanded to yield the area spanned by the merging cluster
- Ex: merging cluster with *R*=3 and the result of its expansion



$VS_{NW} = \{ a_{NW} , b_{NW} , c_{NW} \}$
$VS_{NE} = \{ a_{NE} , d_{NE} \}$

VERTEX SET

- Vertex set of merging cluster *M* consists of four subsets (one for each of the diagonal directions) of vertices of blocks which must be expanded to yield the area spanned by the merging cluster
- Ex: merging cluster with *R*=3 and the result of its expansion



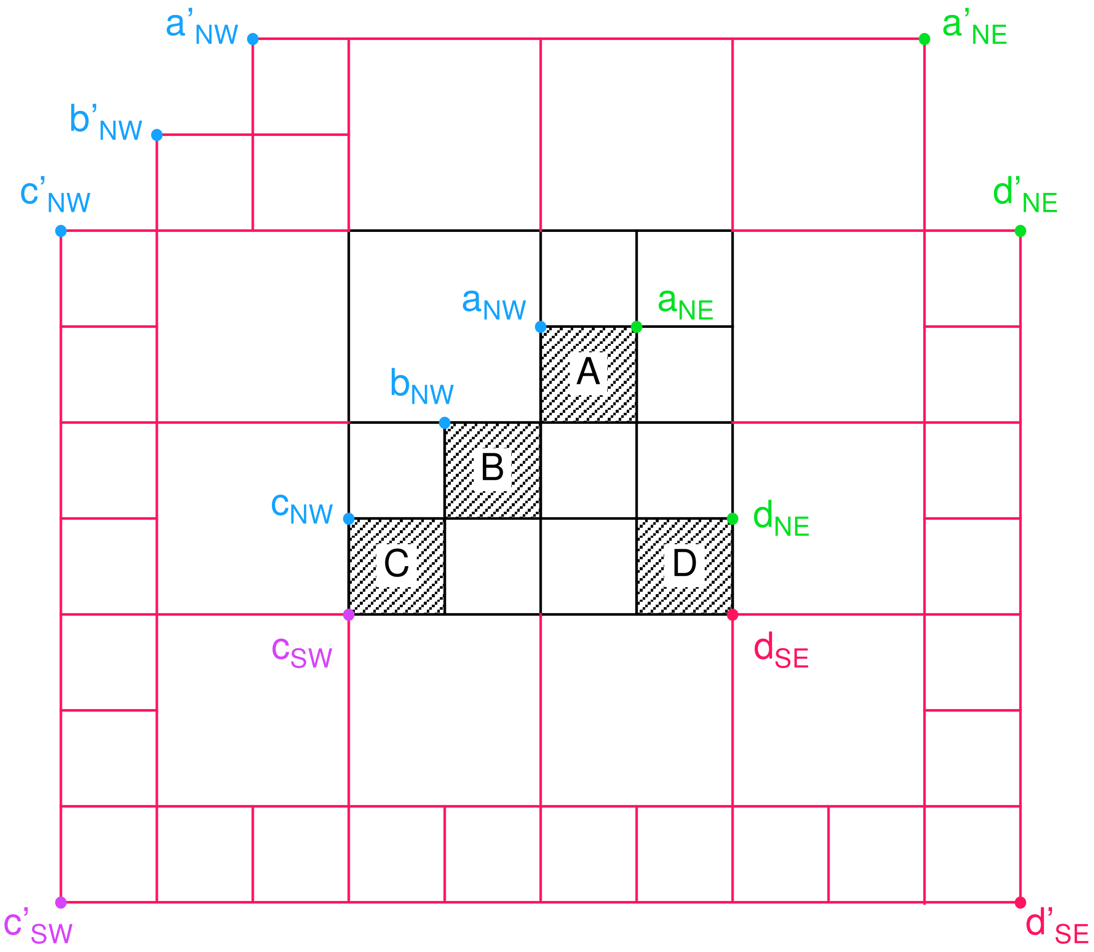
$VS_{NW} = \{ a_{NW} , b_{NW} , c_{NW} \}$

$VS_{NE} = \{ a_{NE} , d_{NE} \}$

$VS_{SW} = \{ c_{SW} \}$

VERTEX SET

- Vertex set of merging cluster *M* consists of four subsets (one for each of the diagonal directions) of vertices of blocks which must be expanded to yield the area spanned by the merging cluster
- Ex: merging cluster with *R*=3 and the result of its expansion



$VS_{NW} = \{ a_{NW} , b_{NW} , c_{NW} \}$

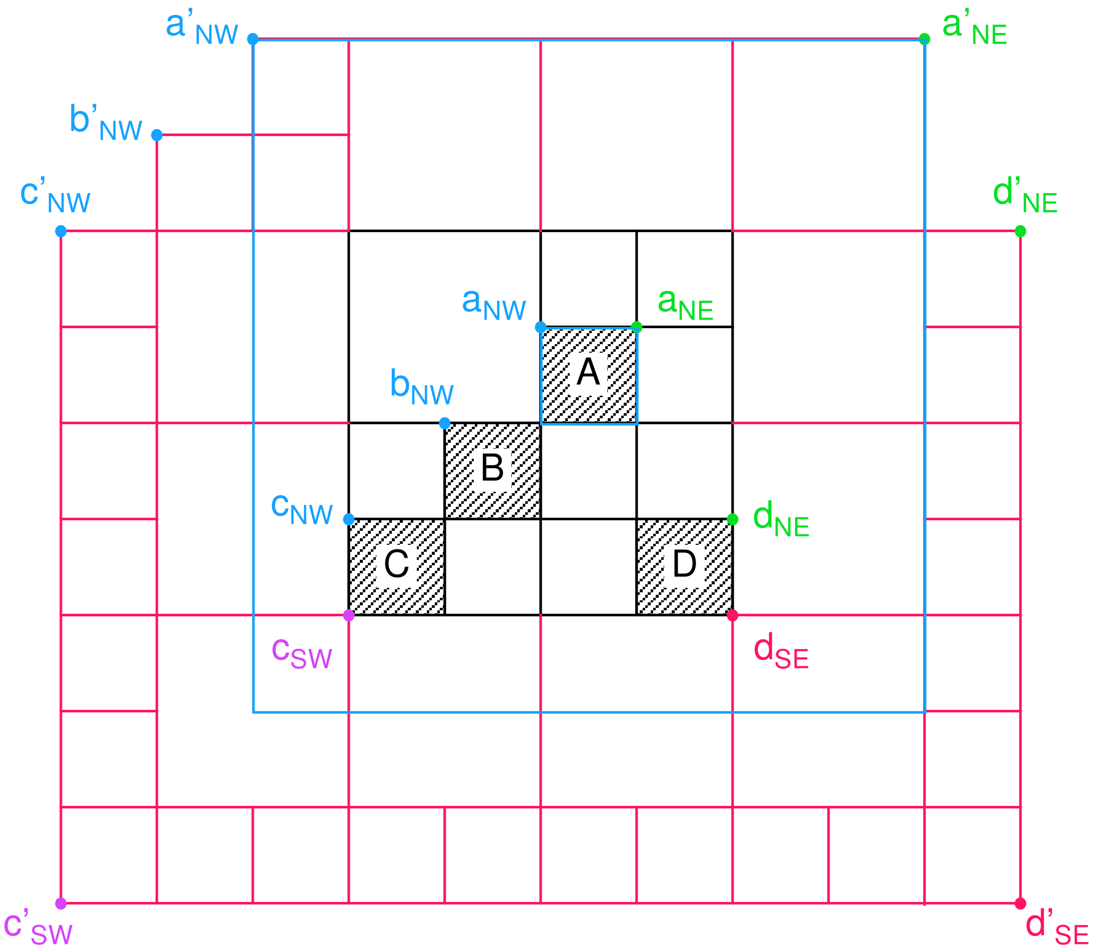
$VS_{NE} = \{ a_{NE} , d_{NE} \}$

$VS_{SW} = \{ c_{SW} \}$

$VS_{SE} = \{ d_{SE} \}$

VERTEX SET

- Vertex set of merging cluster *M* consists of four subsets (one for each of the diagonal directions) of vertices of blocks which must be expanded to yield the area spanned by the merging cluster
- Ex: merging cluster with R=3 and the result of its expansion

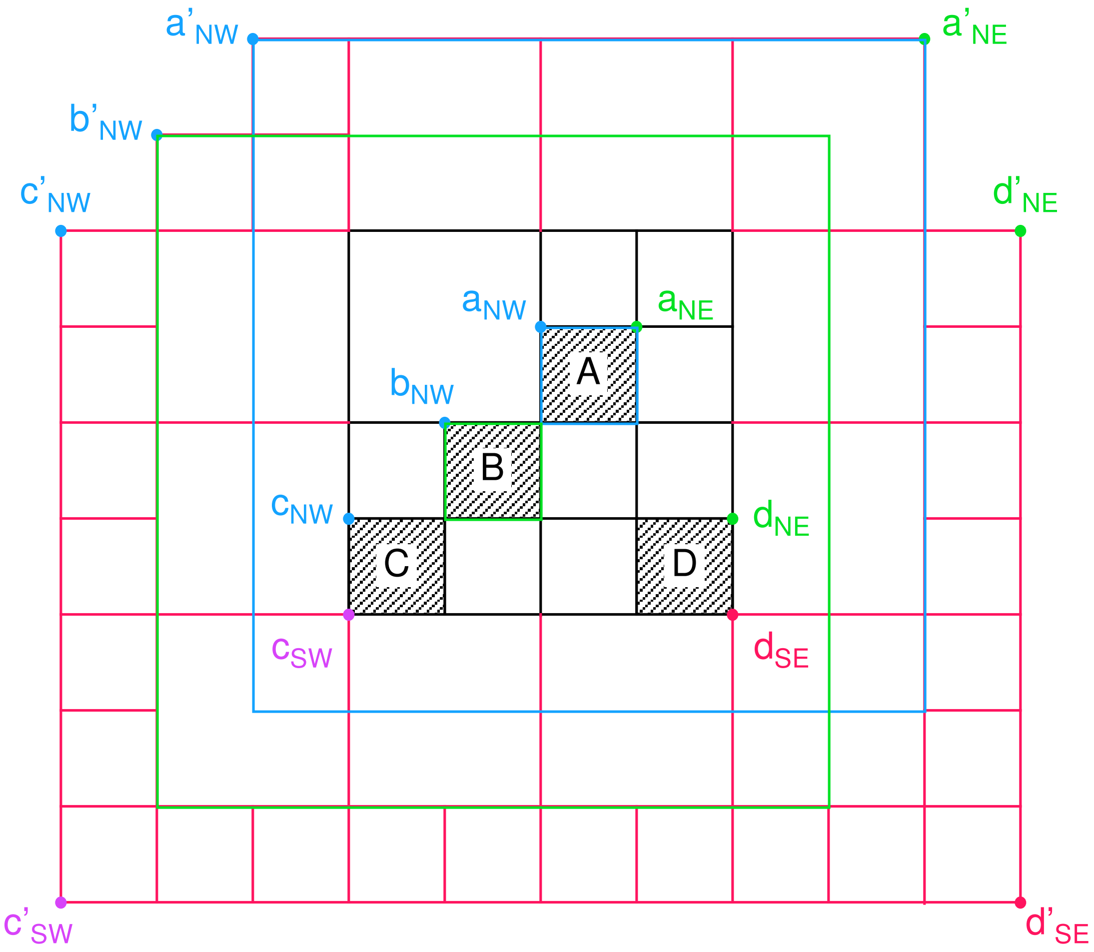


$VS_{NW} = \{ a_{NW} , b_{NW} , c_{NW} \}$
$VS_{NE} = \{ a_{NE} , d_{NE} \}$
$VS_{SW} = \{ c_{SW} \}$
$VS_{SE} = \{ d_{SE} \}$

- Compare with BUFFER1 which expands the elements of M individually and takes their union

VERTEX SET

- Vertex set of merging cluster *M* consists of four subsets (one for each of the diagonal directions) of vertices of blocks which must be expanded to yield the area spanned by the merging cluster
- Ex: merging cluster with *R*=3 and the result of its expansion

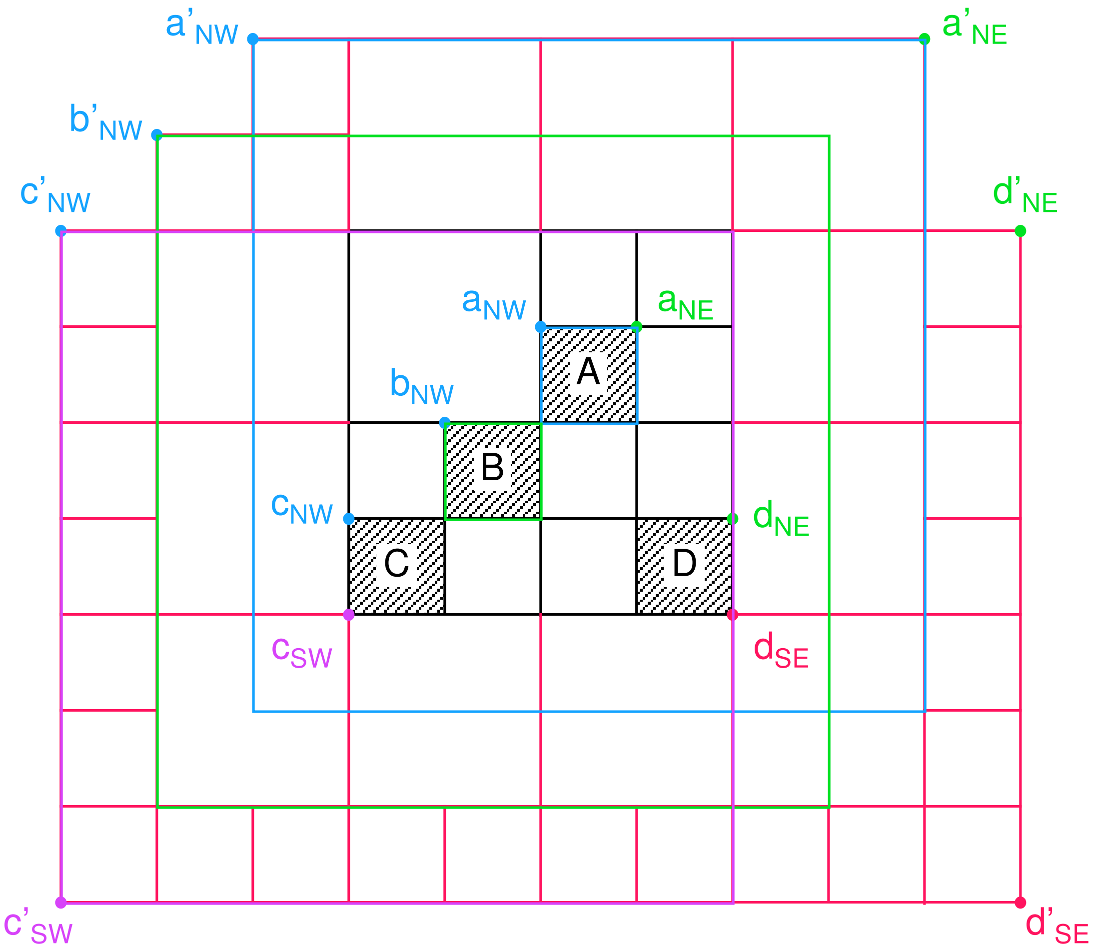


$VS_{NW} = \{ a_{NW}, b_{NW}, c_{NW} \}$
$VS_{NE} = \{ a_{NE}, d_{NE} \}$
$VS_{SW} = \{ c_{SW} \}$
$VS_{SE} = \{ d_{SE} \}$

- Compare with BUFFER1 which expands the elements of M individually and takes their union

VERTEX SET

- Vertex set of merging cluster *M* consists of four subsets (one for each of the diagonal directions) of vertices of blocks which must be expanded to yield the area spanned by the merging cluster
- Ex: merging cluster with *R*=3 and the result of its expansion



$VS_{NW} = \{ a_{NW} , b_{NW} , c_{NW} \}$

$VS_{NE} = \{ a_{NE} , d_{NE} \}$

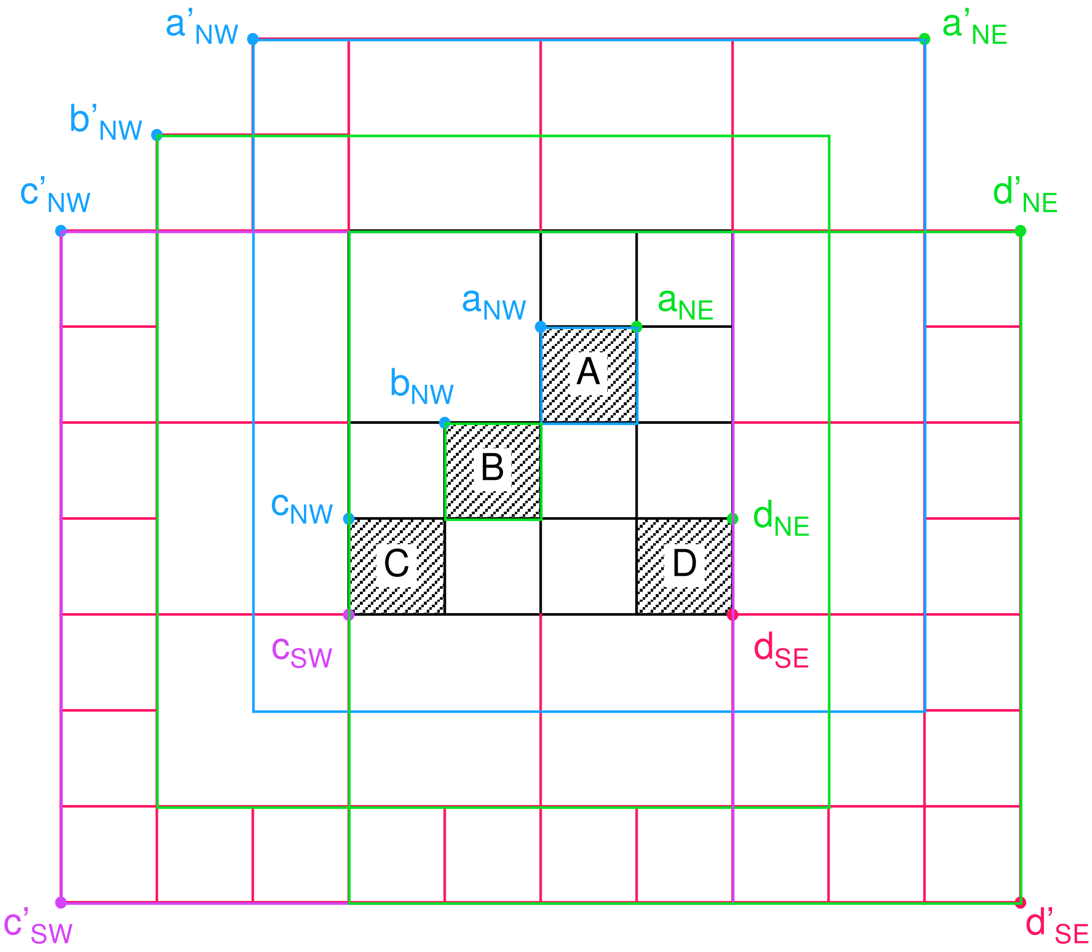
$VS_{SW} = \{ c_{SW} \}$

$VS_{SE} = \{ d_{SE} \}$

- Compare with BUFFER1 which expands the elements of M individually and takes their union

VERTEX SET

- Vertex set of merging cluster *M* consists of four subsets (one for each of the diagonal directions) of vertices of blocks which must be expanded to yield the area spanned by the merging cluster
- Ex: merging cluster with *R*=3 and the result of its expansion



$VS_{NW} = \{ a_{NW}, b_{NW}, c_{NW} \}$

$VS_{NE} = \{ a_{NE}, d_{NE} \}$

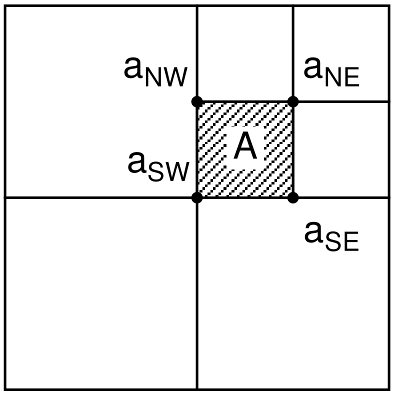
$VS_{SW} = \{ c_{SW} \}$

$VS_{SE} = \{ d_{SE} \}$

- Compare with BUFFER1 which expands the elements of M individually and takes their union

CONSTRUCTION OF THE VERTEX SET

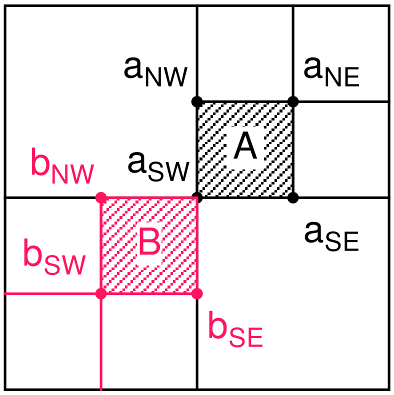
- For a BLACK node *P* in merging cluster *M*, vertex *v* of *P* is in VS_d if
  1. *v* is the *d* vertex of *P*, and
  2. *v* is not in the closed OPQUAD(*d*) quadrant of a quadtree-like subdivision rooted at a *d* vertex of any other BLACK node in *M*
- The vertex subsets are disjoint by definition and thus they can be constructed independently
- Ex:



Construction of Vertex Subsets				
Node	VS_{NW}	VS_{NE}	VS_{SW}	VS_{SE}
A	$\{a_{NW}\}$	$\{a_{NE}\}$	$\{a_{SW}\}$	$\{a_{SE}\}$

CONSTRUCTION OF THE VERTEX SET

- For a BLACK node *P* in merging cluster *M*, vertex *v* of *P* is in VS_d if
  1. *v* is the *d* vertex of *P*, and
  2. *v* is not in the closed OPQUAD(*d*) quadrant of a quadtree-like subdivision rooted at a *d* vertex of any other BLACK node in *M*
- The vertex subsets are disjoint by definition and thus they can be constructed independently
- Ex:



Construction of Vertex Subsets

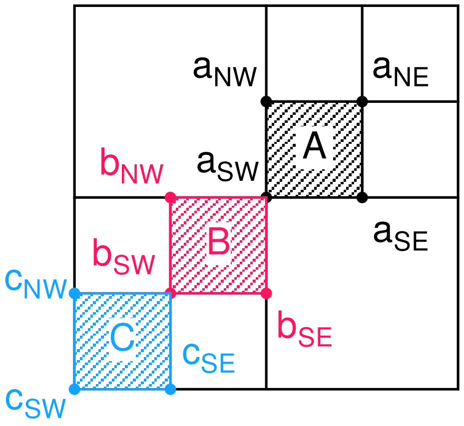
Node	VS_{NW}	VS_{NE}	VS_{SW}	VS_{SE}
A	$\{a_{NW}\}$	$\{a_{NE}\}$	$\{a_{SW}\}$	$\{a_{SE}\}$
B	$\{a_{NW}, b_{NW}\}$	$\{a_{NE}\}$	$\{b_{SW}\}$	$\{a_{SE}, b_{SE}\}$

a_{SW} is eliminated by being in OPQUAD(SW) rooted at b_{SW}

b_{NE} is eliminated by being in OPQUAD(NE) rooted at a_{NE}

CONSTRUCTION OF THE VERTEX SET

- For a BLACK node *P* in merging cluster *M*, vertex *v* of *P* is in VS_d if
  1. *v* is the *d* vertex of *P*, and
  2. *v* is not in the closed OPQUAD(*d*) quadrant of a quadtree-like subdivision rooted at a *d* vertex of any other BLACK node in *M*
- The vertex subsets are disjoint by definition and thus they can be constructed independently
- Ex:



Construction of Vertex Subsets				
Node	VS_{NW}	VS_{NE}	VS_{SW}	VS_{SE}
A	$\{a_{NW}\}$	$\{a_{NE}\}$	$\{a_{SW}\}$	$\{a_{SE}\}$
B	$\{a_{NW}, b_{NW}\}$	$\{a_{NE}\}$	$\{b_{SW}\}$	$\{a_{SE}, b_{SE}\}$
C	$\{a_{NW}, b_{NW}, c_{NW}\}$	$\{a_{NE}\}$	$\{c_{SW}\}$	$\{a_{SE}, b_{SE}, c_{SE}\}$

a_{SW} is eliminated by being in OPQUAD(SW) rooted at b_{SW}

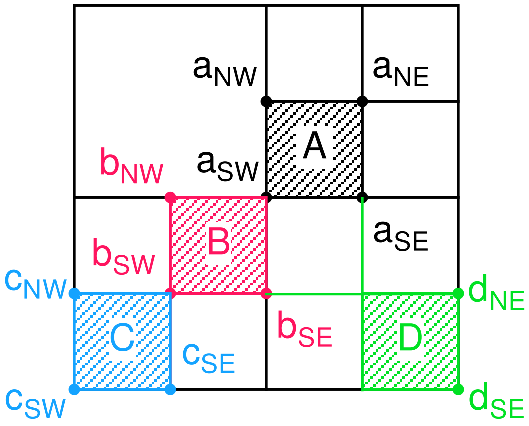
b_{NE} is eliminated by being in OPQUAD(NE) rooted at a_{NE}

b_{SW} is eliminated by being in OPQUAD(SW) rooted at c_{SW}

c_{NE} is eliminated by being in OPQUAD(NE) rooted at a_{NE}

CONSTRUCTION OF THE VERTEX SET

- For a BLACK node *P* in merging cluster *M*, vertex *v* of *P* is in VS$_d$ if
  1. *v* is the *d* vertex of *P*, and
  2. *v* is not in the closed OPQUAD(*d*) quadrant of a quadtree-like subdivision rooted at a *d* vertex of any other BLACK node in *M*
- The vertex subsets are disjoint by definition and thus they can be constructed independently
- Ex:



Construction of Vertex Subsets

Node	VS$_{NW}$	VS$_{NE}$	VS$_{SW}$	VS$_{SE}$
A	$\{a_{NW}\}$	$\{a_{NE}\}$	$\{a_{SW}\}$	$\{a_{SE}\}$
B	$\{a_{NW}, b_{NW}\}$	$\{a_{NE}\}$	$\{b_{SW}\}$	$\{a_{SE}, b_{SE}\}$
C	$\{a_{NW}, b_{NW}, c_{NW}\}$	$\{a_{NE}\}$	$\{c_{SW}\}$	$\{a_{SE}, b_{SE}, c_{SE}\}$
D	$\{a_{NW}, b_{NW}, c_{NW}\}$	$\{a_{NE}, d_{NE}\}$	$\{c_{SW}\}$	$\{d_{SE}\}$

a_{SW} is eliminated by being in OPQUAD(SW) rooted at b_{SW}

b_{NE} is eliminated by being in OPQUAD(NE) rooted at a_{NE}

b_{SW} is eliminated by being in OPQUAD(SW) rooted at c_{SW}

c_{NE} is eliminated by being in OPQUAD(NE) rooted at a_{NE}

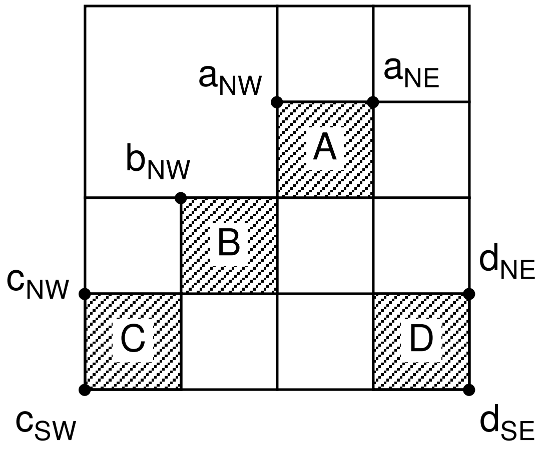
$\{a_{SE}, b_{SE}, c_{SE}\}$ are eliminated by being in OPQUAD(SE) rooted at d_{SE}

EXPANDING THE VERTEX SET

1. Vertex sets guide the expansion in the 8 directions
2. Merging cluster can be inserted as one BLACK node
3. Areas of expansion in the diagonal directions always form staircase-like regions
    - extreme points of the staircase in direction *d* are images of the translations of the elements of VS_d
4. Areas of expansion in lateral directions are rectangles
    - for lateral direction *d* use the vertex in $\mathrm{VS}_{d?}$ or $\mathrm{VS}_{?d}$ that is closest to the *d* boundary of the merging cluster
5. Expansion in direction *d* can be avoided if the neighboring node in direction *d* is BLACK or a merging cluster

    Ex: expansion by 3

Direction Use

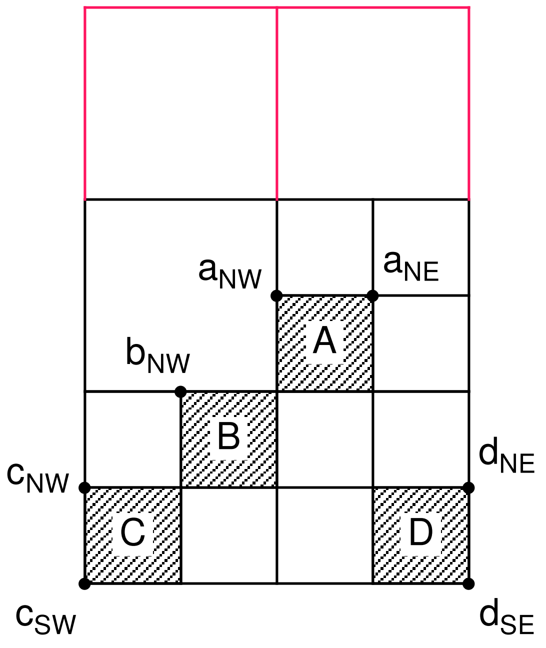


EXPANDING THE VERTEX SET

1. Vertex sets guide the expansion in the 8 directions
2. Merging cluster can be inserted as one BLACK node
3. Areas of expansion in the diagonal directions always form staircase-like regions
    - extreme points of the staircase in direction *d* are images of the translations of the elements of VS_d
4. Areas of expansion in lateral directions are rectangles
    - for lateral direction *d* use the vertex in $\mathrm{VS}_{d?}$ or $\mathrm{VS}_{?d}$ that is closest to the *d* boundary of the merging cluster
5. Expansion in direction *d* can be avoided if the neighboring node in direction *d* is BLACK or a merging cluster

Ex: expansion by 3

Direction	Use
N	a_{NW}

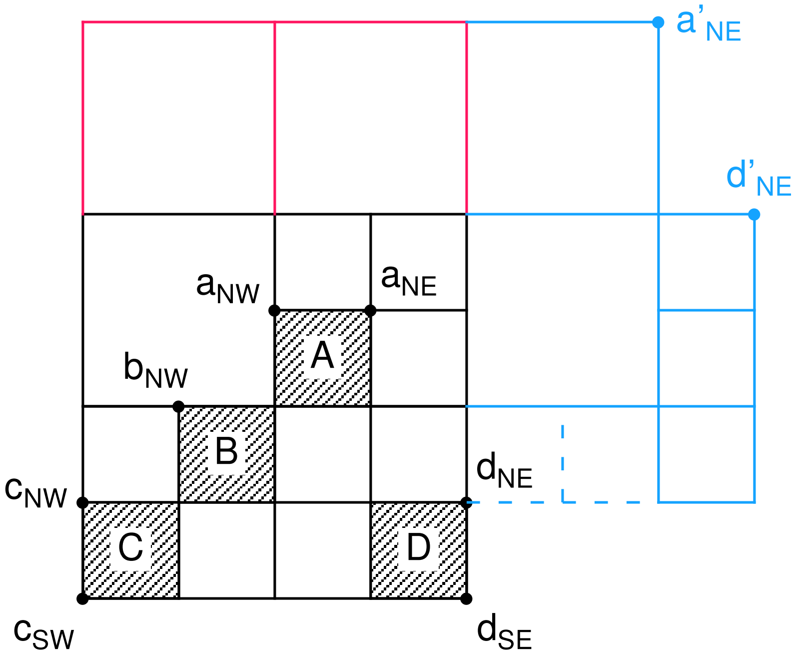


EXPANDING THE VERTEX SET

1. Vertex sets guide the expansion in the 8 directions
2. Merging cluster can be inserted as one BLACK node
3. Areas of expansion in the diagonal directions always form staircase-like regions
   - extreme points of the staircase in direction *d* are images of the translations of the elements of VS_d
4. Areas of expansion in lateral directions are rectangles
   - for lateral direction *d* use the vertex in $\mathrm{VS}_{d?}$ or $\mathrm{VS}_{?d}$ that is closest to the *d* boundary of the merging cluster
5. Expansion in direction *d* can be avoided if the neighboring node in direction *d* is BLACK or a merging cluster

Ex: expansion by 3

Direction	Use
N	a_{NW}
NE	a_{NE} , d_{NE}

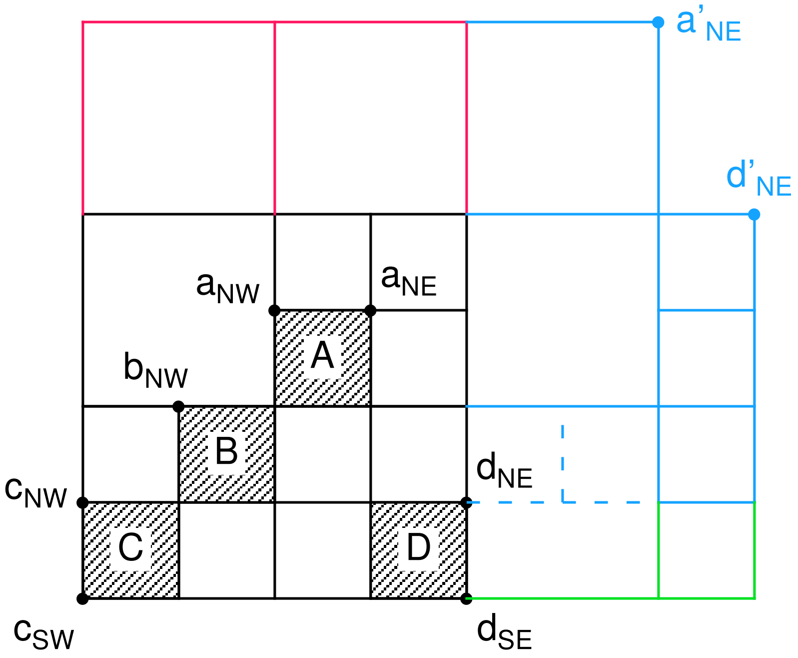


EXPANDING THE VERTEX SET

1. Vertex sets guide the expansion in the 8 directions
2. Merging cluster can be inserted as one BLACK node
3. Areas of expansion in the diagonal directions always form staircase-like regions
   - extreme points of the staircase in direction *d* are images of the translations of the elements of VS_d
4. Areas of expansion in lateral directions are rectangles
   - for lateral direction *d* use the vertex in $\mathrm{VS}_{d?}$ or $\mathrm{VS}_{?d}$ that is closest to the *d* boundary of the merging cluster
5. Expansion in direction *d* can be avoided if the neighboring node in direction *d* is BLACK or a merging cluster

Ex: expansion by 3

Direction	Use
N	a_{NW}
NE	a_{NE} , d_{NE}
E	d_{NE}

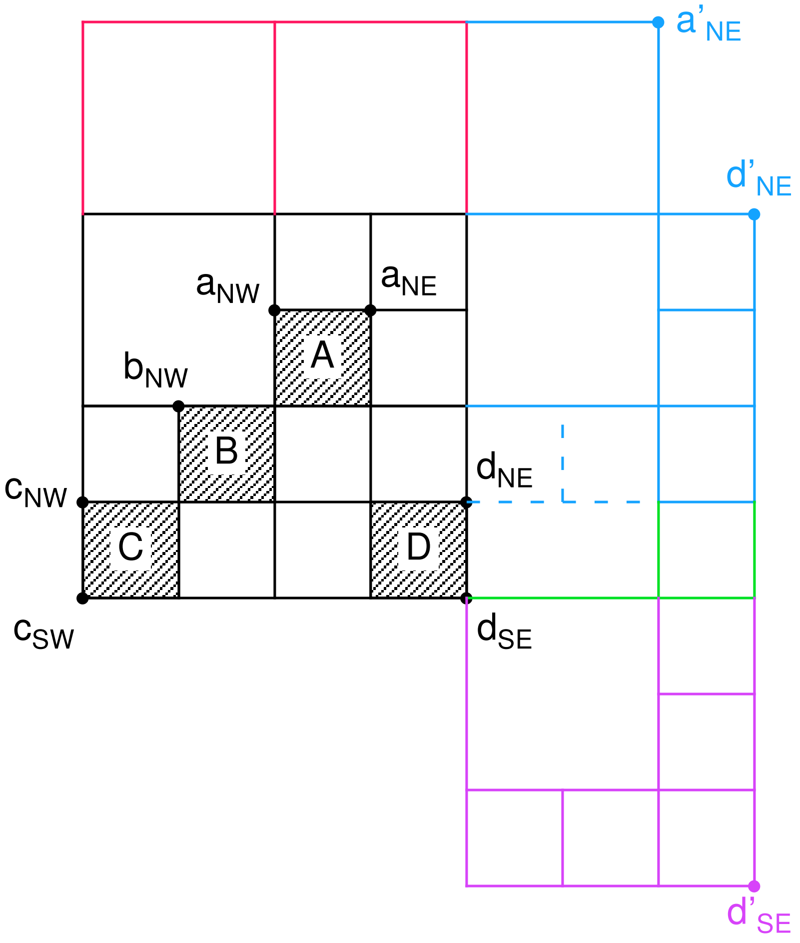


EXPANDING THE VERTEX SET

1. Vertex sets guide the expansion in the 8 directions
2. Merging cluster can be inserted as one BLACK node
3. Areas of expansion in the diagonal directions always form staircase-like regions
    - extreme points of the staircase in direction *d* are images of the translations of the elements of VS_d
4. Areas of expansion in lateral directions are rectangles
    - for lateral direction *d* use the vertex in $\mathrm{VS}_{d?}$ or $\mathrm{VS}_{?d}$ that is closest to the *d* boundary of the merging cluster
5. Expansion in direction *d* can be avoided if the neighboring node in direction *d* is BLACK or a merging cluster

Ex: expansion by 3

Direction	Use
N	a_{NW}
NE	a_{NE}, d_{NE}
E	d_{NE}
SE	d_{SE}

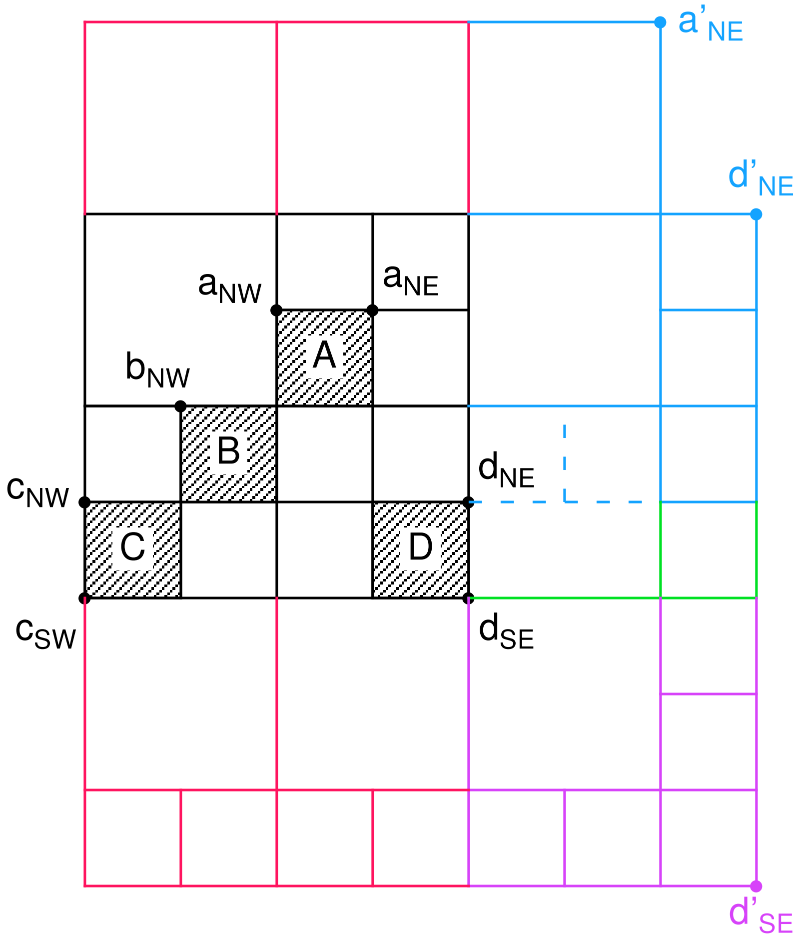


EXPANDING THE VERTEX SET

1. Vertex sets guide the expansion in the 8 directions
2. Merging cluster can be inserted as one BLACK node
3. Areas of expansion in the diagonal directions always form staircase-like regions
   - extreme points of the staircase in direction *d* are images of the translations of the elements of VS_d
4. Areas of expansion in lateral directions are rectangles
   - for lateral direction *d* use the vertex in $\mathrm{VS}_{d?}$ or $\mathrm{VS}_{?d}$ that is closest to the *d* boundary of the merging cluster
5. Expansion in direction *d* can be avoided if the neighboring node in direction *d* is BLACK or a merging cluster

Ex: expansion by 3

Direction	Use
N	a_{NW}
NE	a_{NE} , d_{NE}
E	d_{NE}
SE	d_{SE}
S	c_{SW}

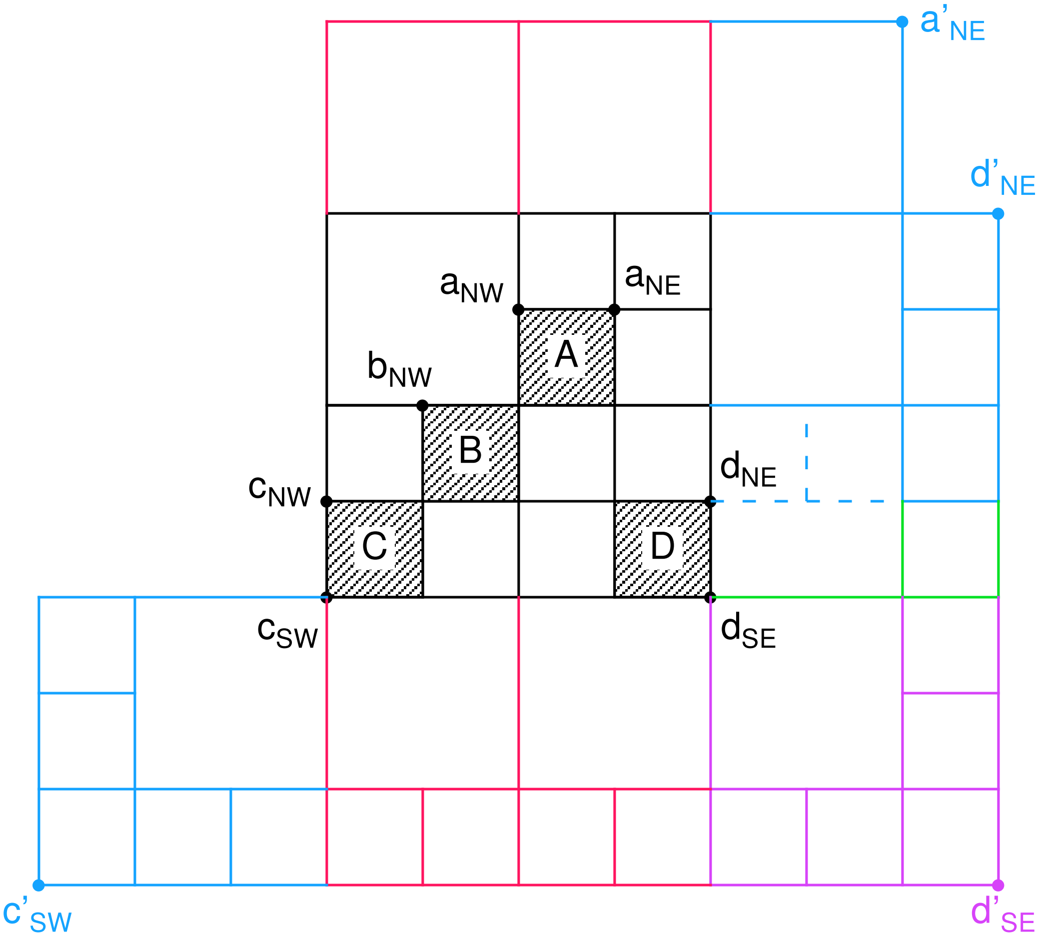


EXPANDING THE VERTEX SET

1. Vertex sets guide the expansion in the 8 directions
2. Merging cluster can be inserted as one BLACK node
3. Areas of expansion in the diagonal directions always form staircase-like regions
   - extreme points of the staircase in direction *d* are images of the translations of the elements of VS_d
4. Areas of expansion in lateral directions are rectangles
   - for lateral direction *d* use the vertex in $\mathrm{VS}_{d?}$ or $\mathrm{VS}_{?d}$ that is closest to the *d* boundary of the merging cluster
5. Expansion in direction *d* can be avoided if the neighboring node in direction *d* is BLACK or a merging cluster

Ex: expansion by 3

Direction	Use
N	a_{NW}
NE	a_{NE}, d_{NE}
E	d_{NE}
SE	d_{SE}
S	c_{SW}
SW	c_{SW}

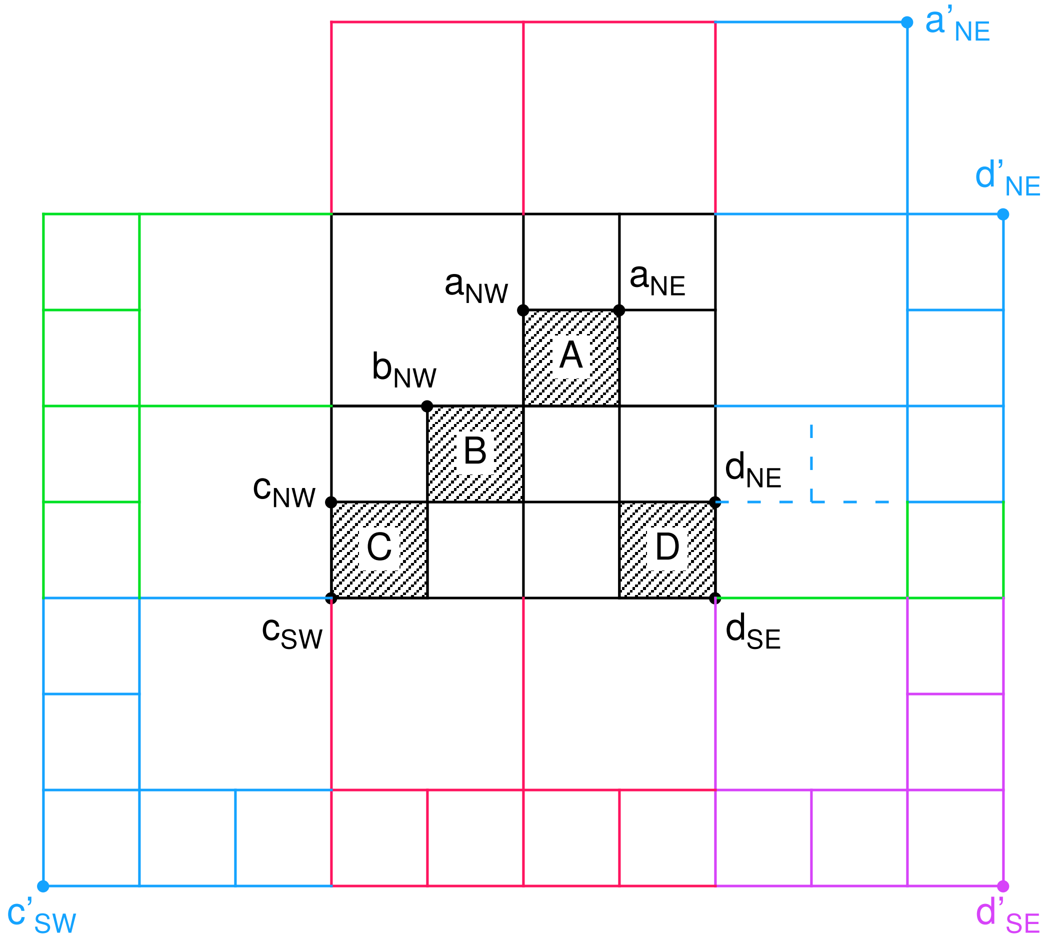


EXPANDING THE VERTEX SET

1. Vertex sets guide the expansion in the 8 directions
2. Merging cluster can be inserted as one BLACK node
3. Areas of expansion in the diagonal directions always form staircase-like regions
   - extreme points of the staircase in direction *d* are images of the translations of the elements of vs_d
4. Areas of expansion in lateral directions are rectangles
   - for lateral direction *d* use the vertex in $\mathrm{vs}_{d?}$ or $\mathrm{vs}_{?d}$ that is closest to the *d* boundary of the merging cluster
5. Expansion in direction *d* can be avoided if the neighboring node in direction *d* is BLACK or a merging cluster

Ex: expansion by 3

Direction	Use
N	a_{NW}
NE	a_{NE} , d_{NE}
E	d_{NE}
SE	d_{SE}
S	c_{SW}
SW	c_{SW}
W	c_{NW}

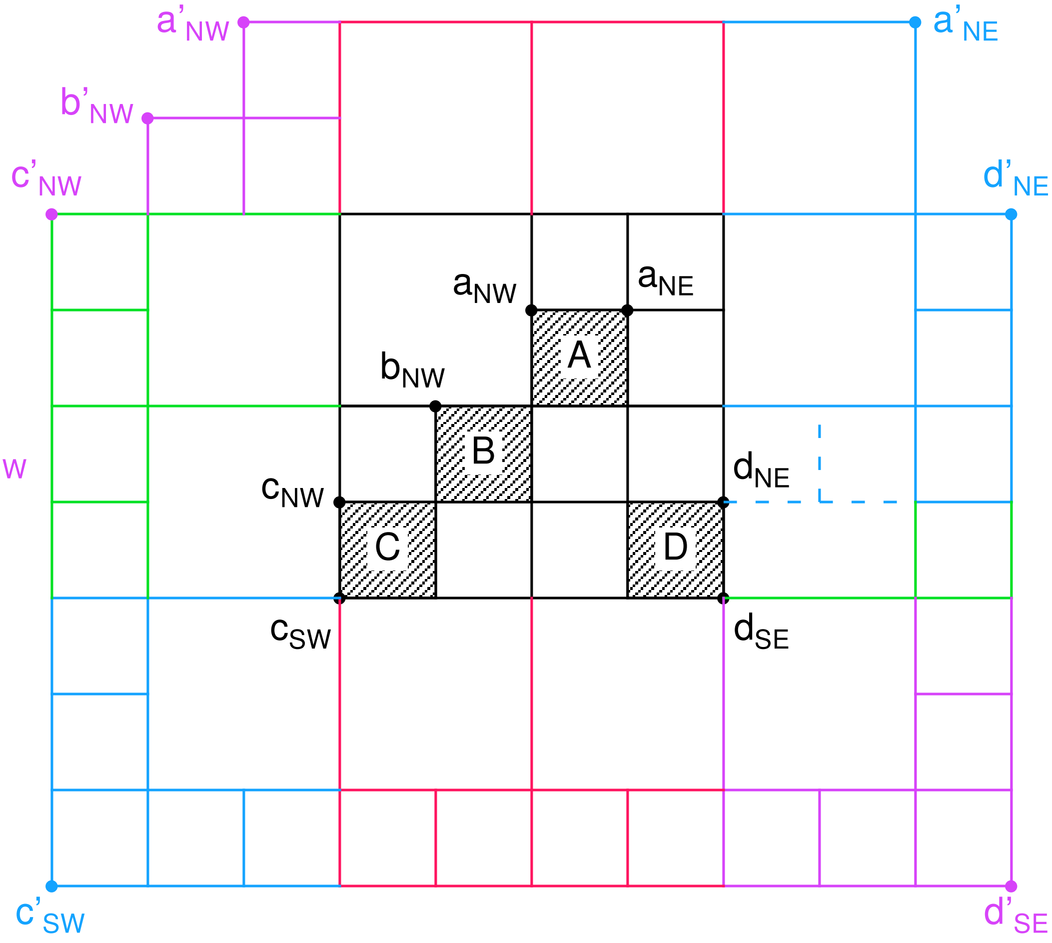


EXPANDING THE VERTEX SET

1. Vertex sets guide the expansion in the 8 directions
2. Merging cluster can be inserted as one BLACK node
3. Areas of expansion in the diagonal directions always form staircase-like regions
    - extreme points of the staircase in direction *d* are images of the translations of the elements of VS_d
4. Areas of expansion in lateral directions are rectangles
    - for lateral direction *d* use the vertex in $\mathrm{VS}_{d?}$ or $\mathrm{VS}_{?d}$ that is closest to the *d* boundary of the merging cluster
5. Expansion in direction *d* can be avoided if the neighboring node in direction *d* is BLACK or a merging cluster

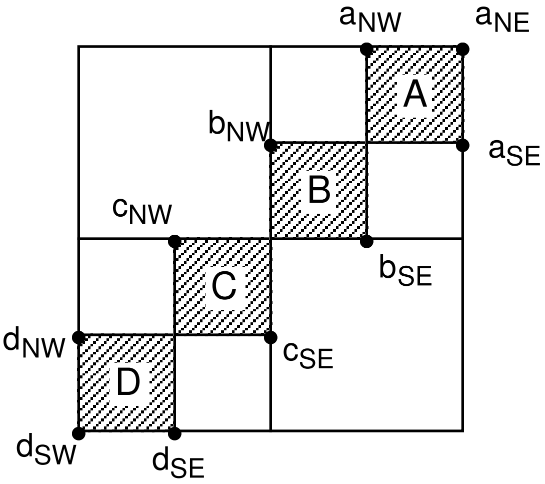
Ex: expansion by 3

Direction	Use
N	a_{NW}
NE	a_{NE}, d_{NE}
E	d_{NE}
SE	d_{SE}
S	c_{SW}
SW	c_{SW}
W	c_{NW}
NW	a_{NW}, b_{NW}, c_{NW}



AN UPPER BOUND ON THE SIZE OF THE VERTEX SET

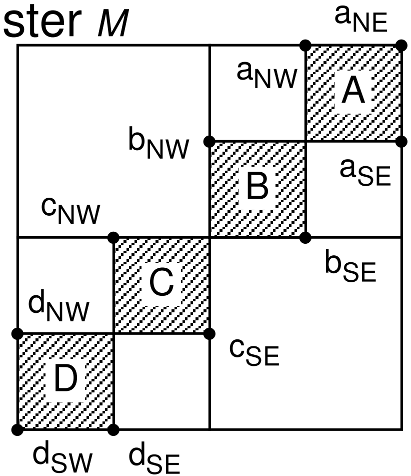
- The size of the union of any two adjacent vertex subsets is ≤ W(R)+1, and this bound is attainable
- Since the vertex subsets are disjoint, the size of the union of all the vertex subsets is ≤ 2·W(R)+2
- Example of the worst case
- Justification:
  1. assume that the elements of VS_d are sorted by their x coordinate values
  2. no 2 vertices in VS_d have the same x coordinate value
  3. number of vertices in VS_d is at most W(R)

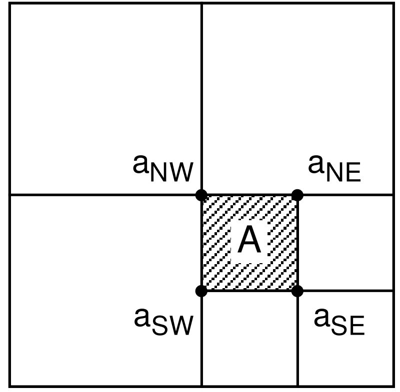


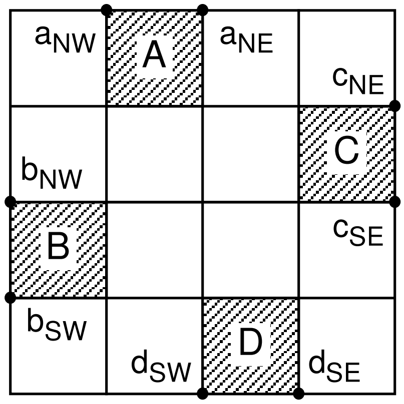
- Example using VS_{NW} and VS_{NE}
  1. no NE vertices of blocks in the SW quadrant with respect to the last (i.e., with the highest x coordinate value) vertex v in VS_{NW} can be in VS_{NE}
  2. therefore, the x coordinate values of the vertices in VS_{NE} must be greater than that of v
  3. thus all vertices in VS_{NW} and VS_{NE} have different x coordinate values and there are at most W(R)+1 such values

ALTERNATIVE BOUND ON VERTEX SET SIZE

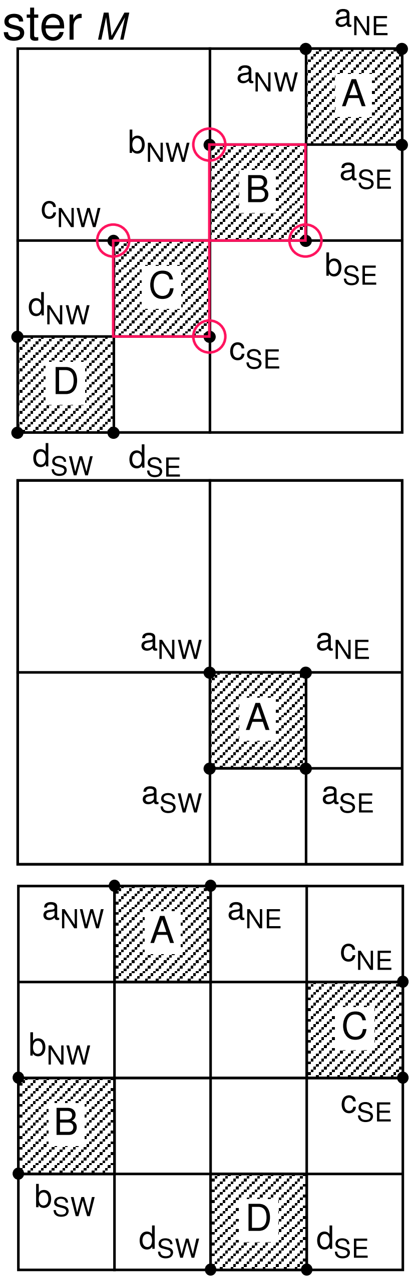
- $B(M)$ = number of BLACK blocks in merging cluster M
- The size of the union of the vertex subsets is $\leq 2 \cdot B(M)+2$, and this bound is attainable
- Examples for $R=3$
- Justification:







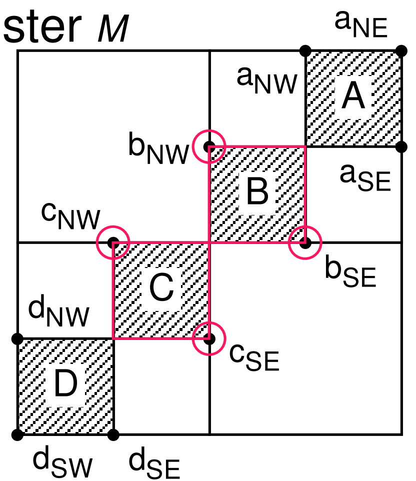
ALTERNATIVE BOUND ON VERTEX SET SIZE

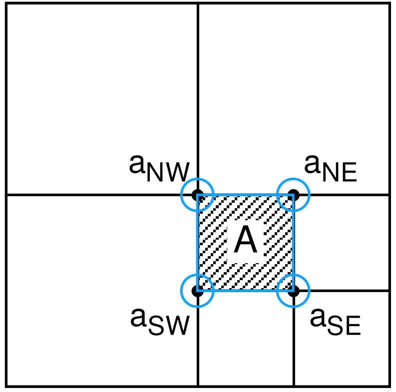


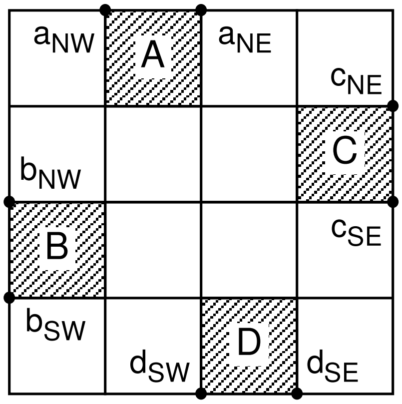
- B(*M*) = number of BLACK blocks in merging cluster *M*
- The size of the union of the vertex subsets is ≤ 2·B(*M*)+2, and this bound is attainable
- Examples for *R*=3
- Justification:
  1. block *v* that is not closest to a boundary can contribute more than 1 vertex (for a maximum of 2) only if the vertices v_e and v_f are in diagonally opposite directions *e* and *f*, respectively, and
     - v_e is not in the closed *f* quadrant of a quadtree-like subdivision rooted at an *e* vertex of any other BLACK node in *M*
     - v_f is not in the closed *e* quadrant of a quadtree-like subdivision rooted at an *f* vertex of any other BLACK node in *M*

ALTERNATIVE BOUND ON VERTEX SET SIZE

- B(M) = number of BLACK blocks in merging cluster M
- The size of the union of the vertex subsets is $\leq 2 \cdot \mathrm{B}(M)+2$, and this bound is attainable
- Examples for $R=3$
- Justification:
  1. block v that is not closest to a boundary can contribute more than 1 vertex (for a maximum of 2) only if the vertices v_e and v_f are in diagonally opposite directions e and f, respectively, and
     - v_e is not in the closed f quadrant of a quadtree-like subdivision rooted at an e vertex of any other BLACK node in M
     - v_f is not in the closed e quadrant of a quadtree-like subdivision rooted at an f vertex of any other BLACK node in M
  2. a BLACK block that is closest to a boundary of M in 3 (actually 4) directions contributes 4 vertices to the vertex set
     - there is at most one such block in M
     - ⇒ M consists of just 1 BLACK block
     - $4+2 \cdot (\mathrm{B}(M)-1) = 2 \cdot \mathrm{B}(M)+2 = 4$ vertices

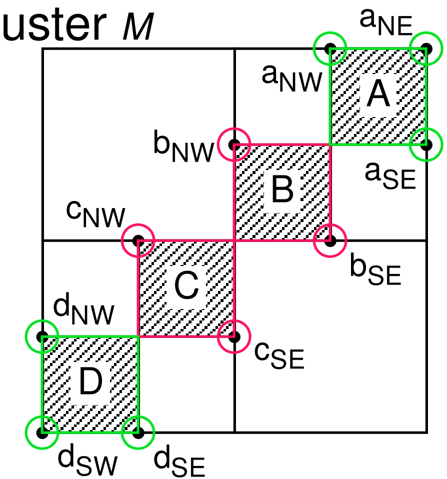


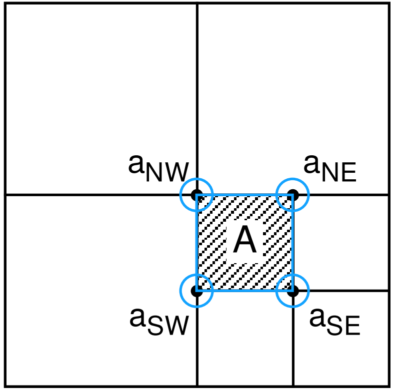


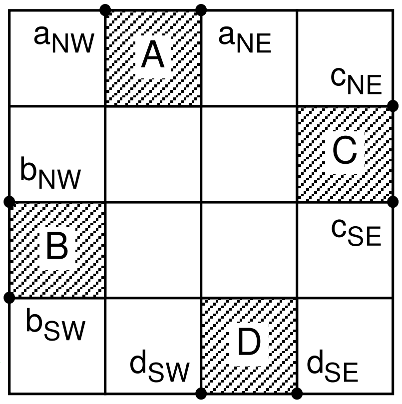


ALTERNATIVE BOUND ON VERTEX SET SIZE

- B(M) = number of BLACK blocks in merging cluster M
- The size of the union of the vertex subsets is ≤ 2·B(M)+2, and this bound is attainable
- Examples for R=3
- Justification:
  1. block v that is not closest to a boundary can contribute more than 1 vertex (for a maximum of 2) only if the vertices v_e and v_f are in diagonally opposite directions e and f, respectively, and
     - v_e is not in the closed f quadrant of a quadtree-like subdivision rooted at an e vertex of any other BLACK node in M
     - v_f is not in the closed e quadrant of a quadtree-like subdivision rooted at an f vertex of any other BLACK node in M
  2. a BLACK block that is closest to a boundary of M in 3 (actually 4) directions contributes 4 vertices to the vertex set
     - there is at most one such block in M
     - ⇒ M consists of just 1 BLACK block
     - 4+2·(B(M)–1) = 2·B(M)+2 = 4 vertices
  3. a BLACK block that is closest to a boundary of M in 2 directions contributes 3 vertices to the vertex set
     - there are at most 2 such blocks in M
     - 2·3+2·(B(M)–2) = 2·B(M)+2

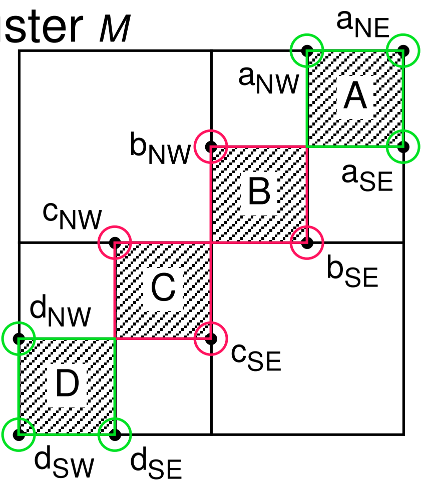


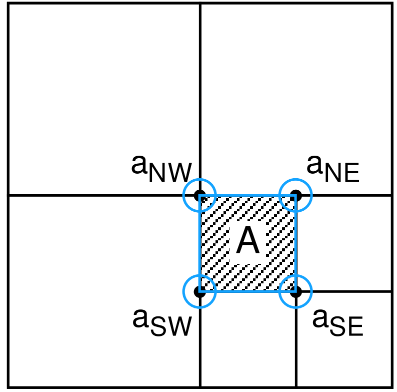


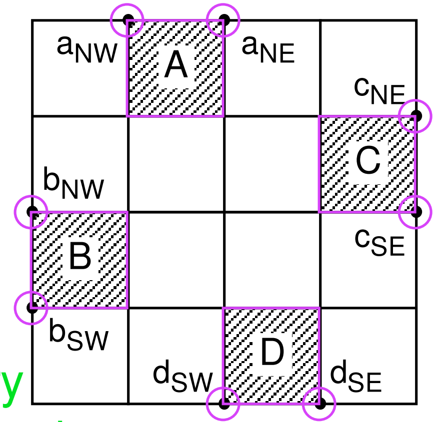


ALTERNATIVE BOUND ON VERTEX SET SIZE

- B(M) = number of BLACK blocks in merging cluster M
- The size of the union of the vertex subsets is ≤ 2·B(M)+2, and this bound is attainable
- Examples for R=3
- Justification:
  1. block v that is not closest to a boundary can contribute more than 1 vertex (for a maximum of 2) only if the vertices v_e and v_f are in diagonally opposite directions e and f, respectively, and
     - v_e is not in the closed f quadrant of a quadtree-like subdivision rooted at an e vertex of any other BLACK node in M
     - v_f is not in the closed e quadrant of a quadtree-like subdivision rooted at an f vertex of any other BLACK node in M
  2. a BLACK block that is closest to a boundary of M in 3 (actually 4) directions contributes 4 vertices to the vertex set
     - there is at most one such block in M
     - ⇒ M consists of just 1 BLACK block
     - 4+2·(B(M)–1) = 2·B(M)+2 = 4 vertices
  3. a BLACK block that is closest to a boundary of M in 2 directions contributes 3 vertices to the vertex set
     - there are at most 2 such blocks in M
     - 2·3+2·(B(M)–2) = 2·B(M)+2
  4. a BLACK block can contribute 2 vertices to the vertex set if it is closest to a boundary of M in a particular direction
     - i·2+2·(B(M)–i) = 2·B(M), where i = number of directions
     - there are at most 4 such blocks in M in which case no other block contributes any vertices

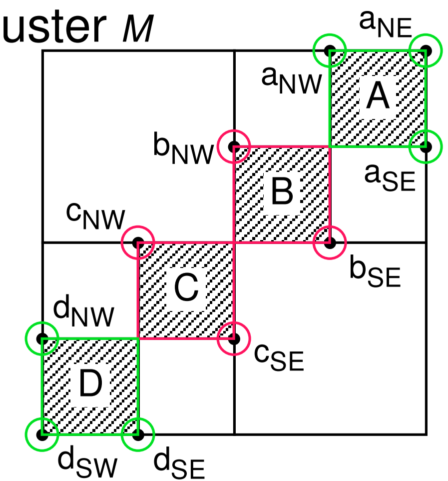


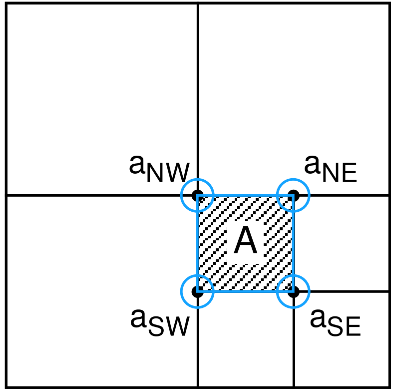


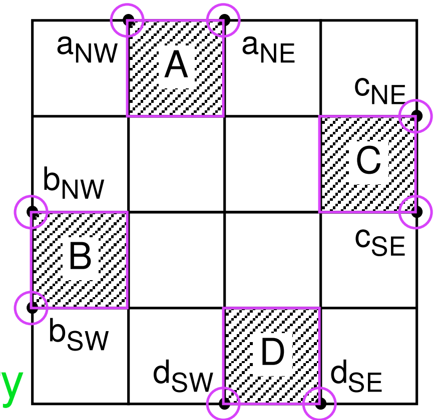


ALTERNATIVE BOUND ON VERTEX SET SIZE

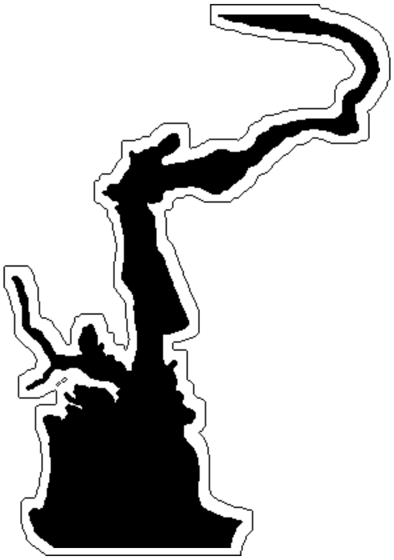
- $B(M)$ = number of BLACK blocks in merging cluster M
- The size of the union of the vertex subsets is $\leq 2 \cdot B(M)+2$, and this bound is attainable
- Examples for $R=3$
- Justification:
  1. block v that is not closest to a boundary can contribute more than 1 vertex (for a maximum of 2) only if the vertices v_e and v_f are in diagonally opposite directions e and f, respectively, and
     - v_e is not in the closed f quadrant of a quadtree-like subdivision rooted at an e vertex of any other BLACK node in M
     - v_f is not in the closed e quadrant of a quadtree-like subdivision rooted at an f vertex of any other BLACK node in M
  2. a BLACK block that is closest to a boundary of M in 3 (actually 4) directions contributes 4 vertices to the vertex set
     - there is at most one such block in M
     - $\Rightarrow$ M consists of just 1 BLACK block
     - $4+2 \cdot (B(M)-1) = 2 \cdot B(M)+2 = 4$ vertices
  3. a BLACK block that is closest to a boundary of M in 2 directions contributes 3 vertices to the vertex set
     - there are at most 2 such blocks in M
     - $2 \cdot 3+2 \cdot (B(M)-2) = 2 \cdot B(M)+2$
  4. a BLACK block can contribute 2 vertices to the vertex set if it is closest to a boundary of M in a particular direction
     - $i \cdot 2+2 \cdot (B(M)-i) = 2 \cdot B(M)$, where i = number of directions
     - there are at most 4 such blocks in M in which case no other block contributes any vertices
- Upper bound of $2 \cdot B(M)+2$ is satisfied by all four cases

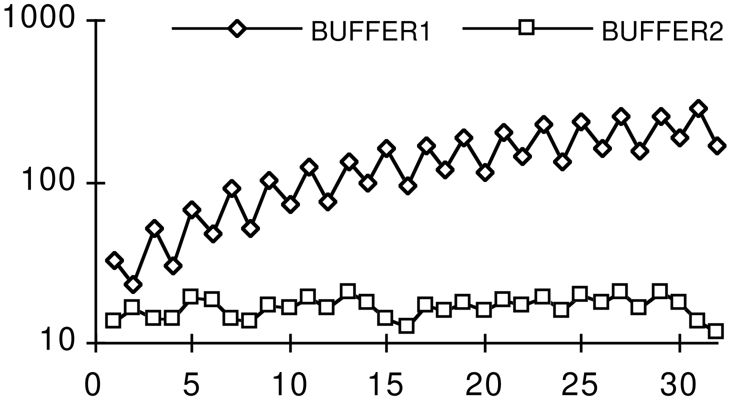






EXAMPLE PERFORMANCE





- Execution times (seconds on a VAX11/785 running UNIX)

Radius	BUFFER1	BUFFER2
1	32.6	13.5
2	23.6	16.4
3	52.4	14.0
4	30.2	14.5
5	68.9	19.0
6	48.8	18.8
7	90.5	14.2
8	51.9	13.5
9	102.2	17.5
10	74.4	16.4
11	125.4	19.3
12	75.7	16.6
13	136.4	21.2
14	97.5	18.1
15	160.7	14.3
16	93.6	12.7

Radius	BUFFER1	BUFFER2
17	167.1	17.0
18	121.3	15.9
19	192.4	17.8
20	117.1	16.0
21	200.6	18.4
22	143.8	17.0
23	226.4	19.1
24	133.3	16.1
25	234.2	20.1
26	163.7	17.8
27	256.7	20.8
28	156.0	16.6
29	259.6	21.1
30	185.6	17.7
31	287.8	13.8
32	170.1	11.9

OBSERVATIONS

- BUFFER2 is far superior to BUFFER1
- Even values of R lead to reduced execution times due to the effect of node aggregation which results in fewer nodes
- BUFFER2's execution times:
  1. remain constant or decrease slightly when $R = 2^r$
  2. increase for other values of R at a slow rate
  3. decrease as R gets very large as image becomes BLACK
- Rationale for BUFFER2 being independent of R
  1. interaction between merging clusters and vertex sets as R increases
     - larger merging clusters and larger vertex sets
     - fewer merging clusters
     - effects tend to cancel each other out
  2. execution time depends on the perimeter
     - number of blocks in the image is proportional to the perimeter of the BLACK regions and the number of BLACK blocks is approximately the same as the number of WHITE blocks
     - size of vertex set is bounded by total number of BLACK blocks in the merging clusters which is constant
  3. analogy to a fill algorithm for computing the expanded region fails
     - a fill implies increase in execution time as R increases
     - however, quadtree blocks get bigger
  4. decompose radius value into its binary representation
     - work in expansion depends on how many bits are 1 as they correspond to the blocks that must be added
     - number of 1 bits for different radii is about the same