Indexing Methods for Moving Object Databases: Games and Other Applications

Hanan Samet Jagan Sankaranarayanan* Michael Auerbach

{hjs,jagan,mikea}@cs.umd.edu

Department of Computer Science

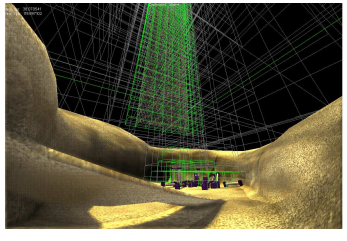
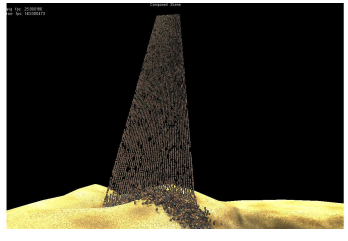
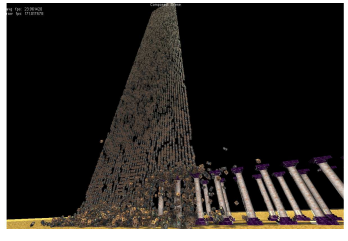
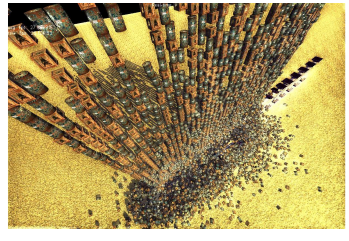
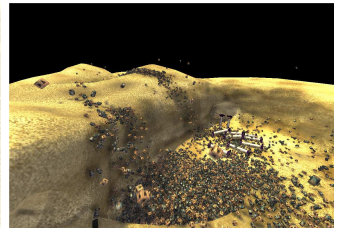
Center for Automation Research

Institute for Advanced Computer Studies

University of Maryland

College Park, MD 20742, USA

* Currently at NEC Labs America.

Overview

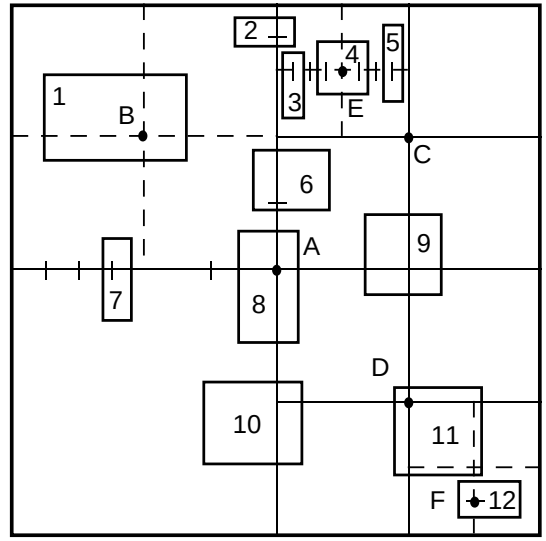
- Problem domain: Database of objects moving at unknown velocities
  - Object positions change over time, possibly obtained from sensors
  - Objects have extent (not point objects)
  - Example applications: games, traffic, interactive vlsi design, etc.
- Goal: Spatial index that supports interactive motion updates of position and of the index so queries/operations can take place at interactive rates
- Results:
  - Use a loose quadtree (cover fieldtree), associating objects with their minimum enclosing expanded quadtree cell with expansion parameter p
  - Show that for suitably chosen value of p, the width of the expanded cell is independent of the object's position
    - Depends only on p and the object's size
    - For appropriate value of p, at most 2 possible cell widths, usually 1
    - Results in constant time $O(1)$ lookup and updates
      - Optimal value of p=0.999 for queries/operations
  - Experiments show most object motions don't require index updates
    - Somewhat analogous to balance and reduced splitting/merging due to node overflow/underflow in B-tree insertions due to node being 50% full

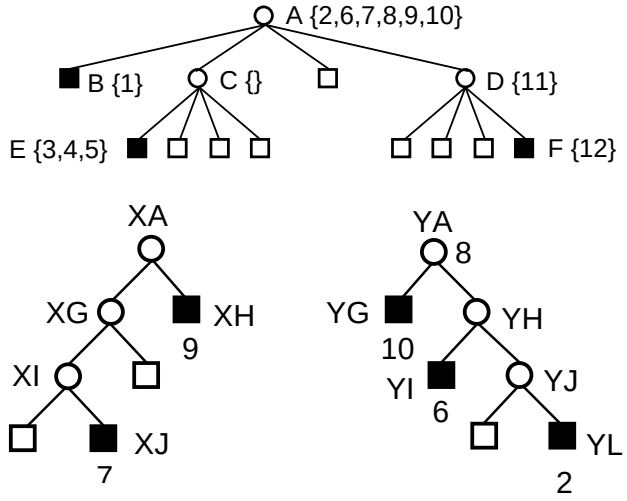
Object Representations

1. Represent objects using an object hierarchy
   - Use minimum bounding hypercube boxes (e.g., R-tree) to speed up process of detecting if objects are present or overlap other objects
   - Drawbacks:
     - Non-disjoint decomposition makes point location need multiple search paths
2. Use recursive decomposition of underlying space into cells based on a bound on the number of objects per cell
   - Drawback: May break objects into pieces
3. Use a hierarchy of congruent cells while still not decomposing the objects
   - Associate each object with its minimum enclosing quadtree cell
   - Examples include MX-CIF quadtree, multilayer grid file, R-file, filter tree, SQ histogram, loose quadtree and equivalent cover fieldtree
     - Differentiated by the access structures used to resolve collisions in sense of many objects with same minimum enclosing quadtree cell
   - Drawbacks
     - Many objects have same minimum enclosing quadtree cell
     - Size of minimum enclosing quadtree cell depends on position of object and less on its size

MX-CIF Quadtree

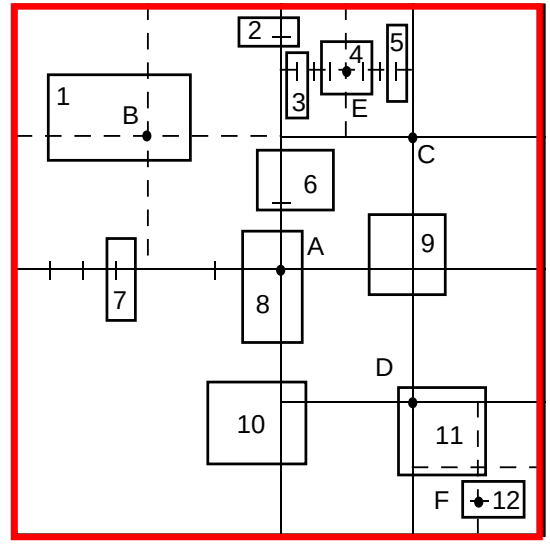
1. Collections of small rectangles for VLSI applications
2. Each rectangle is associated with its minimum enclosing quadtree block
3. Like hashing: quadtree blocks serve as hash buckets

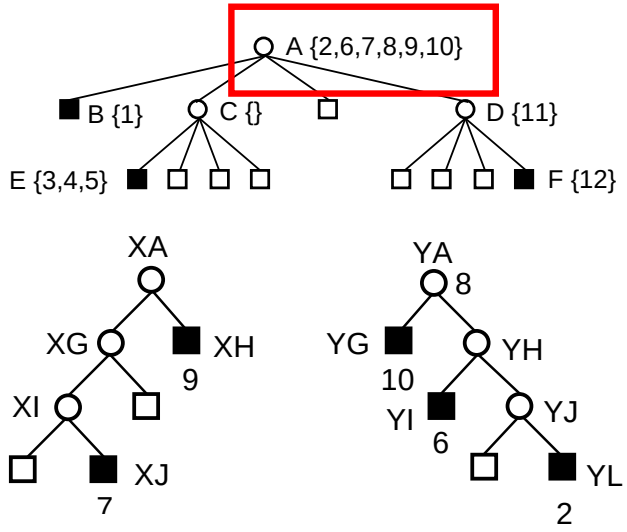




MX-CIF Quadtree

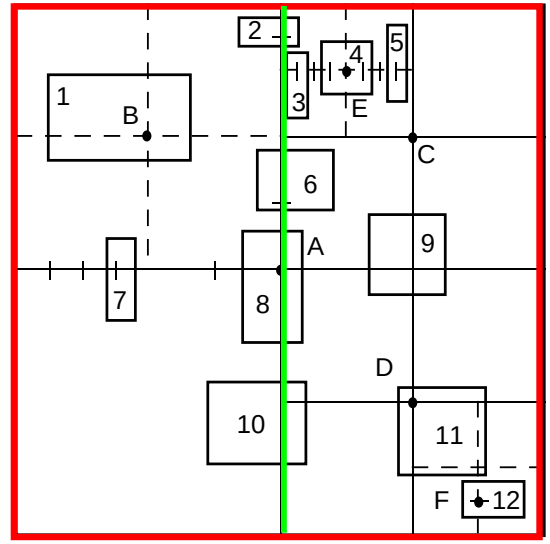
1. Collections of small rectangles for VLSI applications
2. Each rectangle is associated with its minimum enclosing quadtree block
3. Like hashing: quadtree blocks serve as hash buckets
4. Collision = more than one rectangle in a block
   - resolve by using two one-dimensional MX-CIF trees to store the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

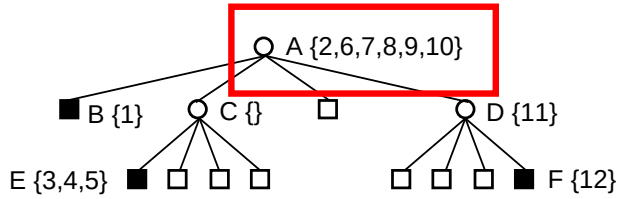


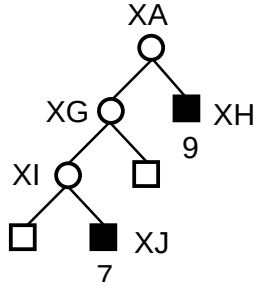


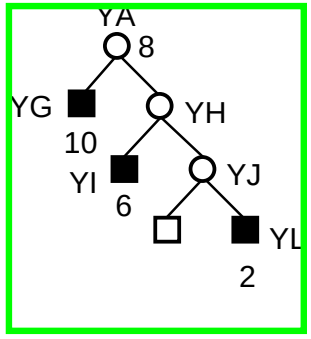
MX-CIF Quadtree

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



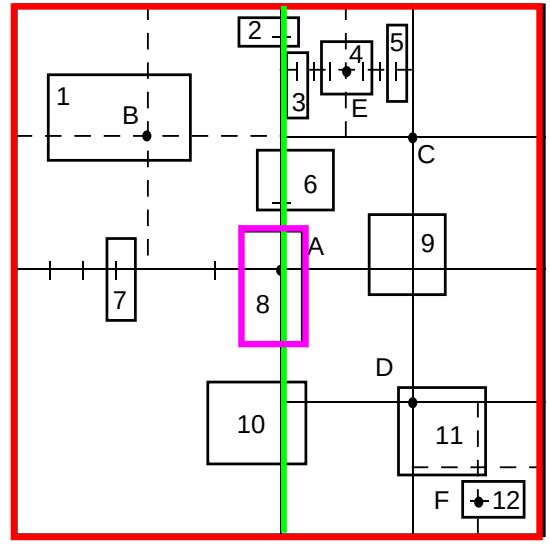


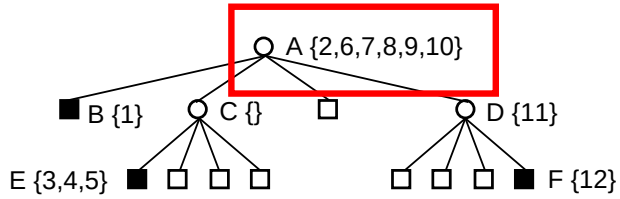


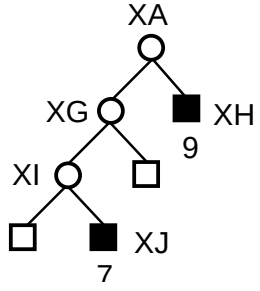


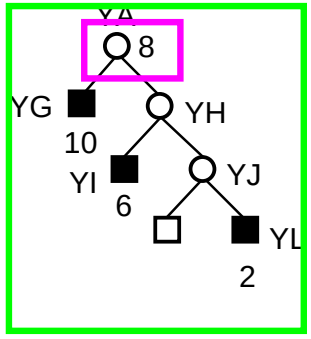
MX-CIF Quadtree

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



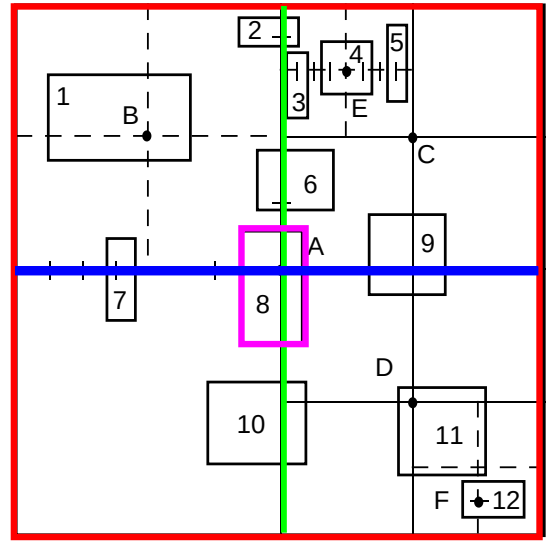


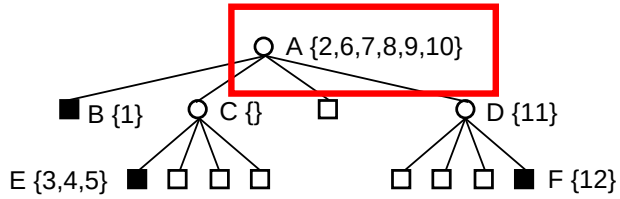


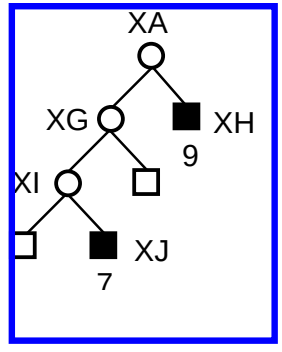


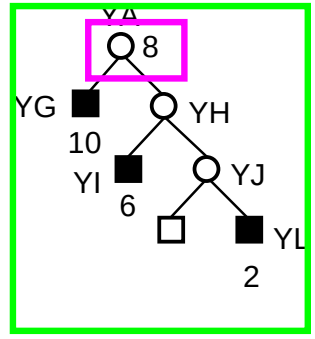
MX-CIF Quadtree

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



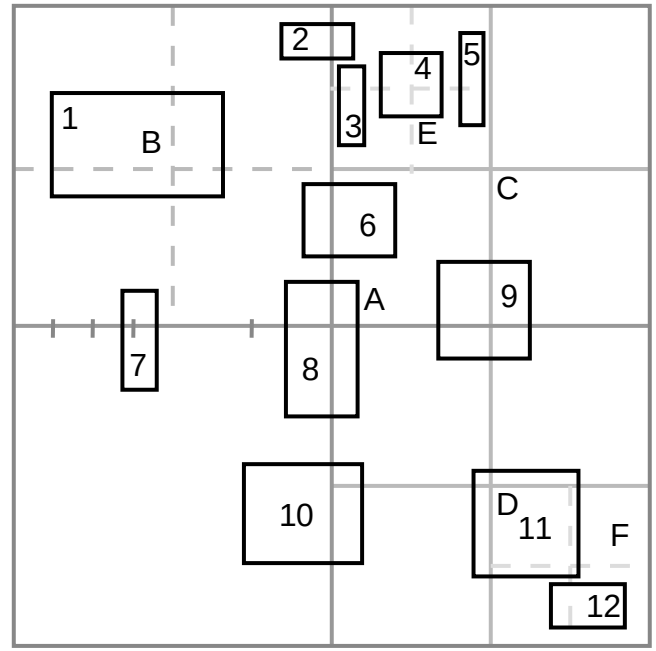


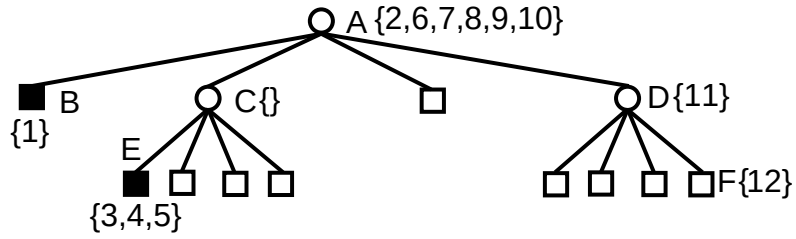




Loose Quadtree (Octree)/Cover Fieldtree

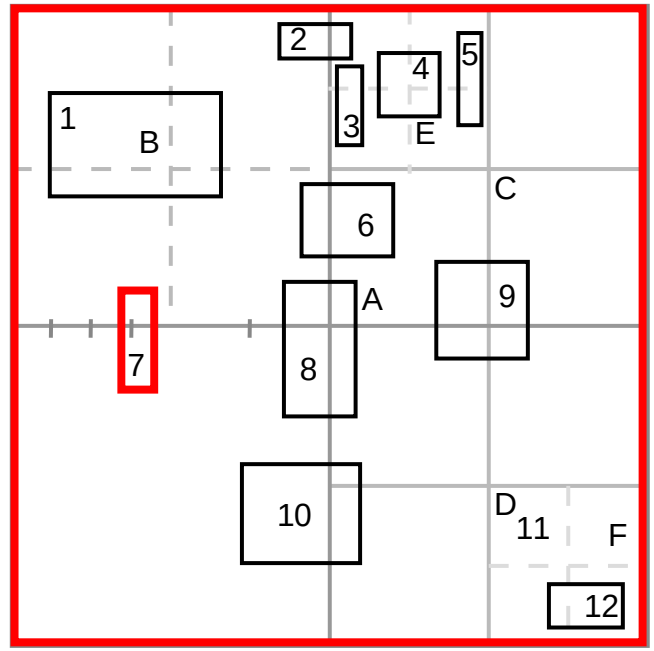
- Overcomes drawback of MX-CIF quadtree that only the minimum width w of the minimum enclosing quadtree cell of object o is a function of o's size

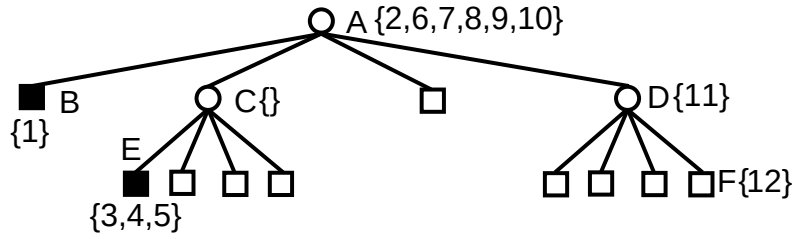




Loose Quadtree (Octree)/Cover Fieldtree

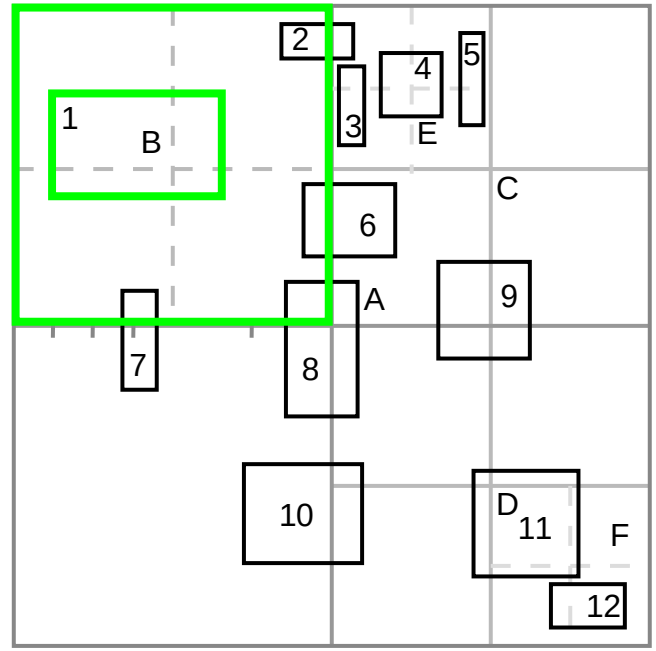
- Overcomes drawback of MX-CIF quadtree that only the minimum width w of the minimum enclosing quadtree cell of object o is a function of o's size
- Instead, the maximum width w depends on both the size and position of the centroid of o and is unbounded

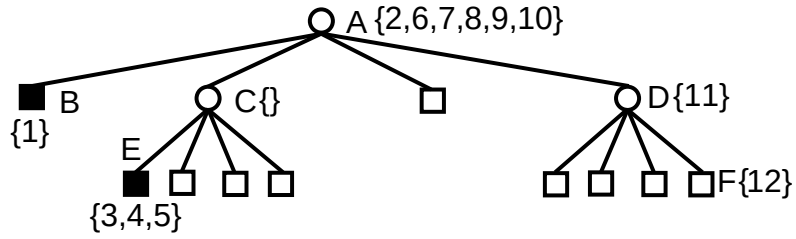




Loose Quadtree (Octree)/Cover Fieldtree

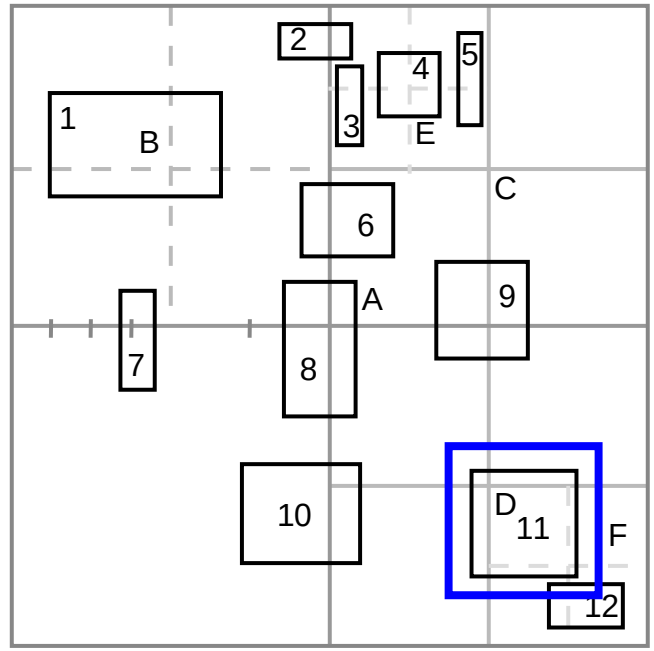
- Overcomes drawback of MX-CIF quadtree that only the minimum width w of the minimum enclosing quadtree cell of object o is a function of o's size
- Instead, the maximum width w depends on both the size and position of the centroid of o and is unbounded

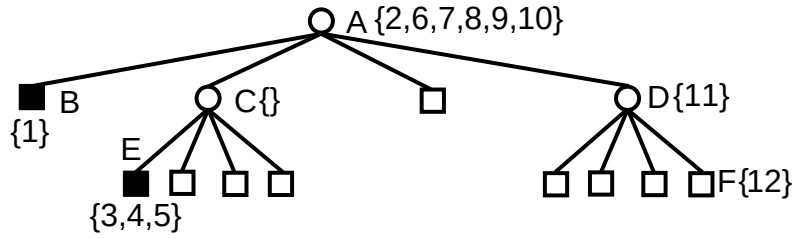




Loose Quadtree (Octree)/Cover Fieldtree

- Overcomes drawback of MX-CIF quadtree that only the minimum width w of the minimum enclosing quadtree cell of object o is a function of o's size
- Instead, the maximum width w depends on both the size and position of the centroid of o and is unbounded
- Solution: expand size of space spanned by each quadtree cell of width w by expansion factor p ($p > 0$) so expanded cell is of width $(1 + p)w$

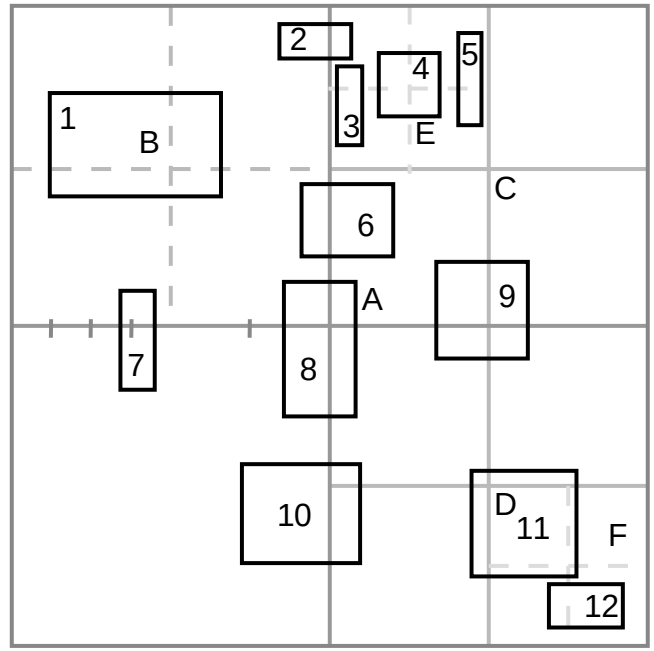


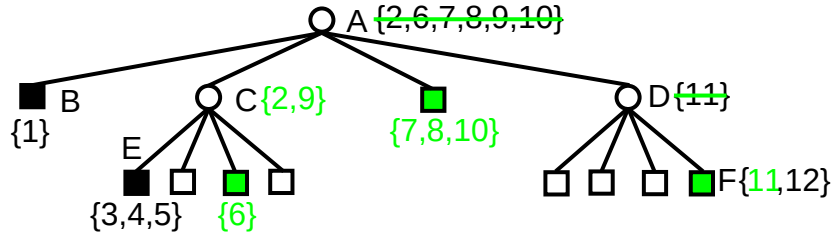


Loose Quadtree (Octree)/Cover Fieldtree

- Overcomes drawback of MX-CIF quadtree that only the minimum width w of the minimum enclosing quadtree cell of object o is a function of o's size
- Instead, the maximum width w depends on both the size and position of the centroid of o and is unbounded
- Solution: expand size of space spanned by each quadtree cell of width w by expansion factor p ($p > 0$) so expanded cell is of width $(1 + p)w$

  1. $p = 0.3$

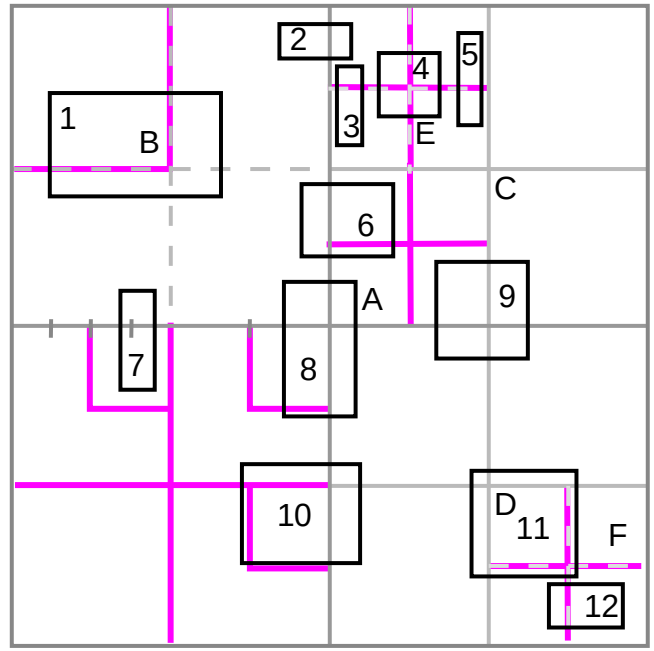


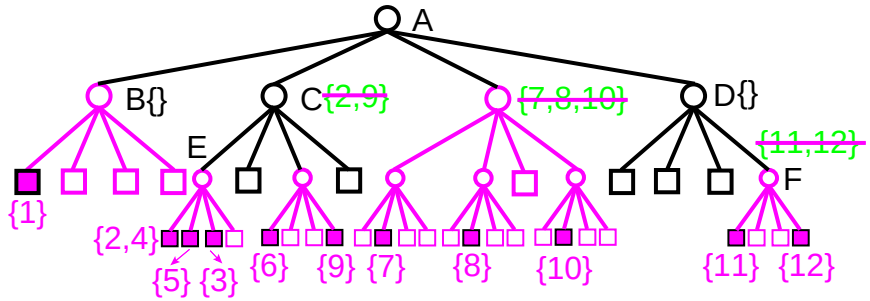


Loose Quadtree (Octree)/Cover Fieldtree

- Overcomes drawback of MX-CIF quadtree that only the minimum width w of the minimum enclosing quadtree cell of object o is a function of o's size
- Instead, the maximum width w depends on both the size and position of the centroid of o and is unbounded
- Solution: expand size of space spanned by each quadtree cell of width w by expansion factor p ($p > 0$) so expanded cell is of width $(1 + p)w$

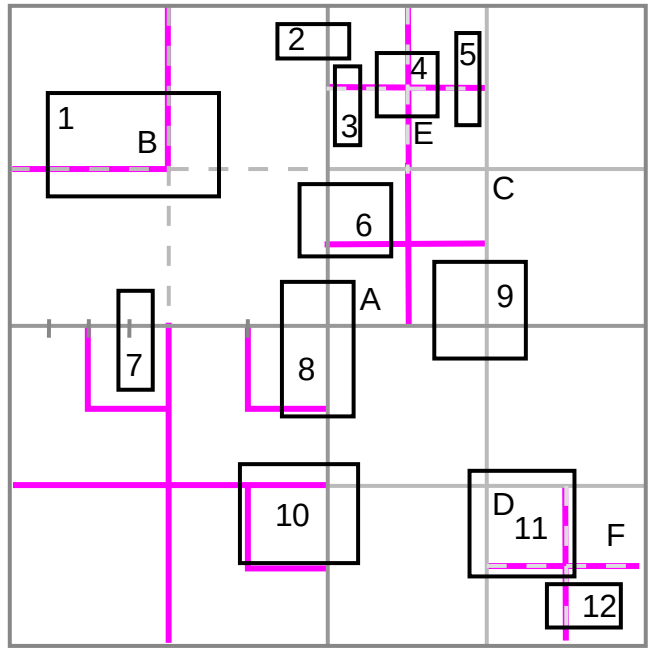
  1. $p = 0.3$
  2. $p = 1.0$

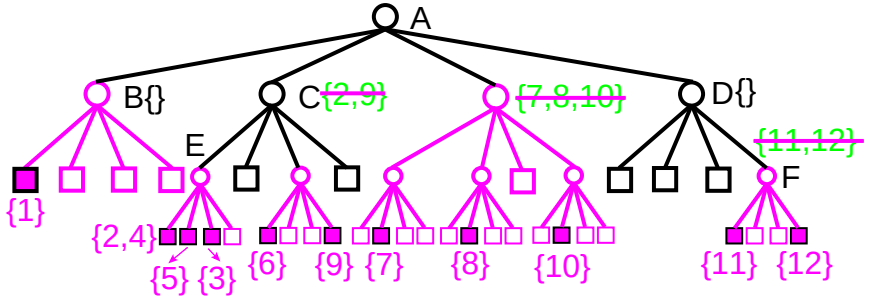




Loose Quadtree (Octree)/Cover Fieldtree

- Overcomes drawback of MX-CIF quadtree that only the minimum width w of the minimum enclosing quadtree cell of object o is a function of o's size
- Instead, the maximum width w depends on both the size and position of the centroid of o and is unbounded
- Solution: expand size of space spanned by each quadtree cell of width w by expansion factor p ($p > 0$) so expanded cell is of width $(1 + p)w$
  1. $p = 0.3$
  2. $p = 1.0$
- First formulated by Frank for spatial applications terming it *cover fieldtree*
- Ulrich devised it for game applications and called it a *loose quadtree*
- Ulrich sought the range of object sizes that can be associated with a cell
  - not very useful
- Want inverse: which range of cell sizes can be associated with an object?





Partition Fieldtree

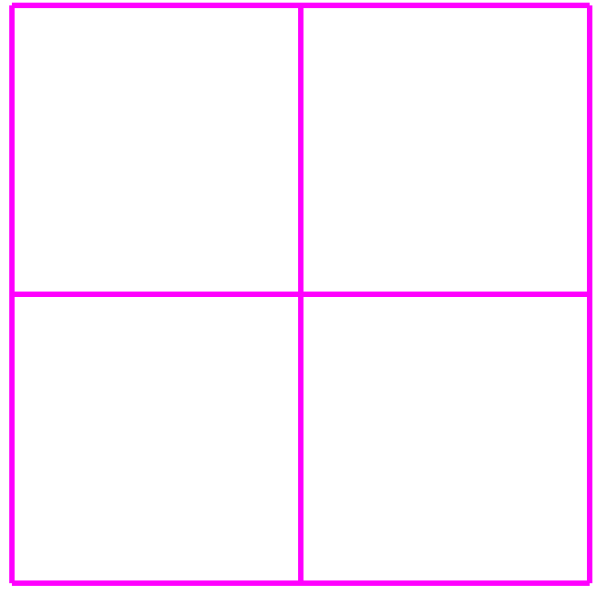
- Alternative to loose quadtree (octree)/cover fieldtree at overcoming drawback of MX-CIF quadtree that the width w of the minimum enclosing quadtree cell of a rectangle o is not a function of the size of o

Partition Fieldtree

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

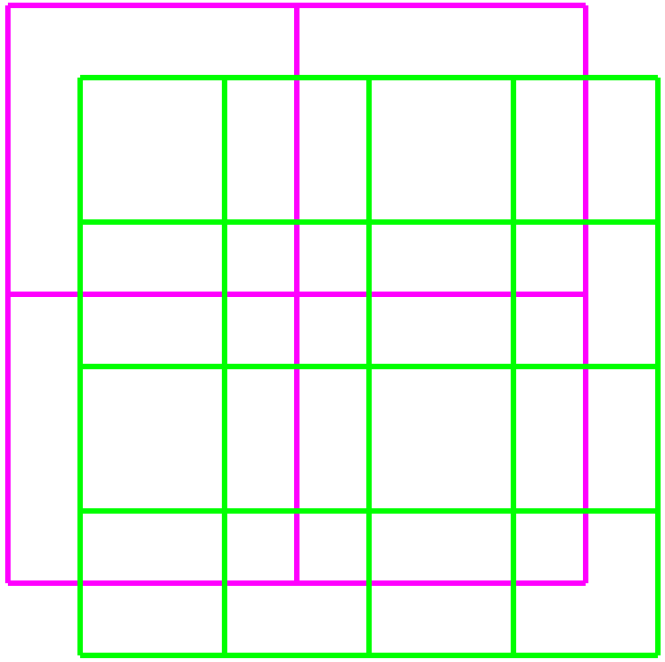
Partition Fieldtree

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



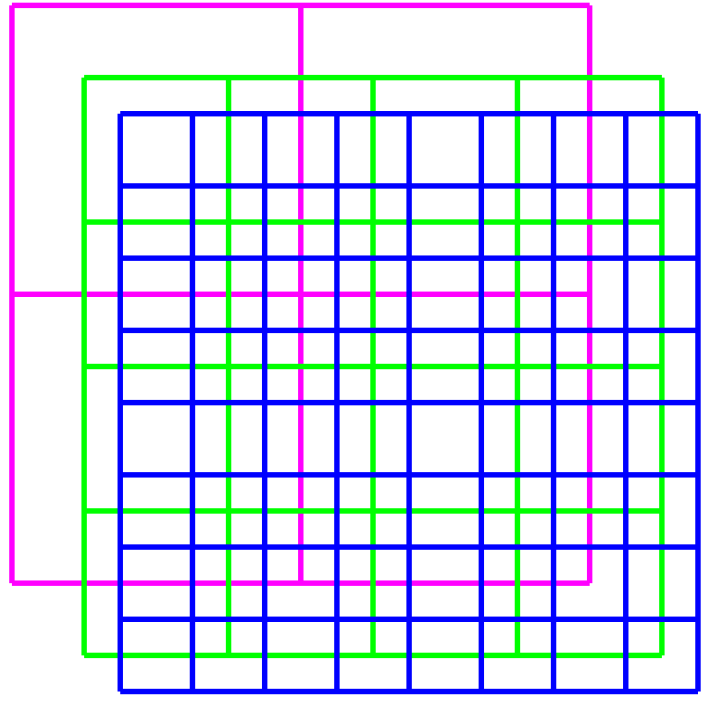
Partition Fieldtree

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



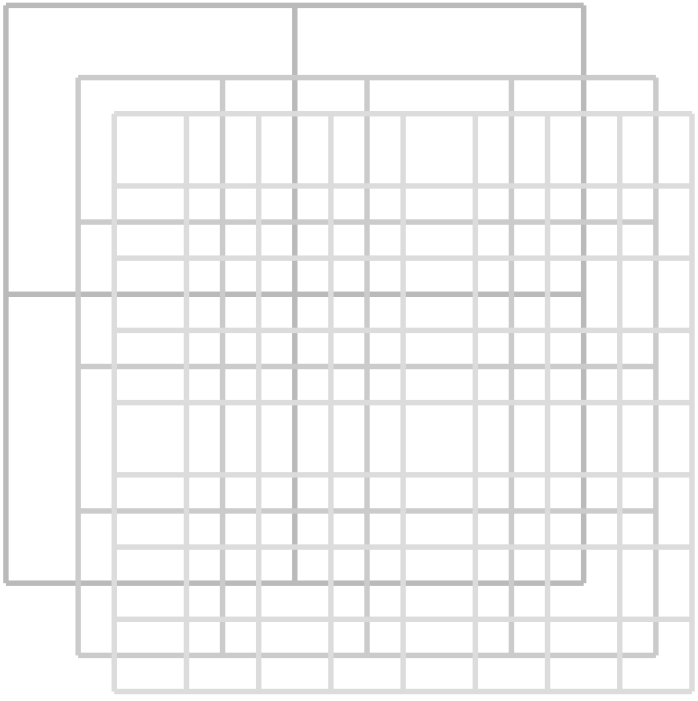
Partition Fieldtree

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



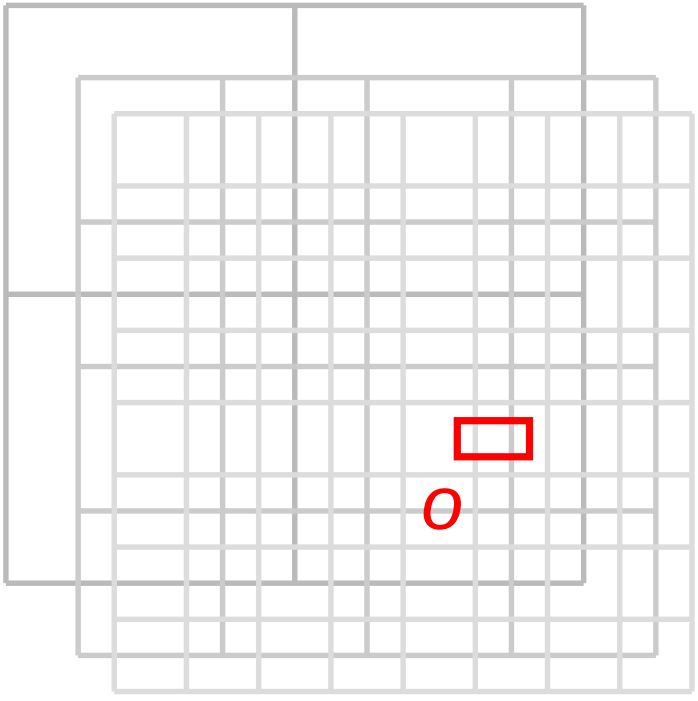
Partition Fieldtree

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



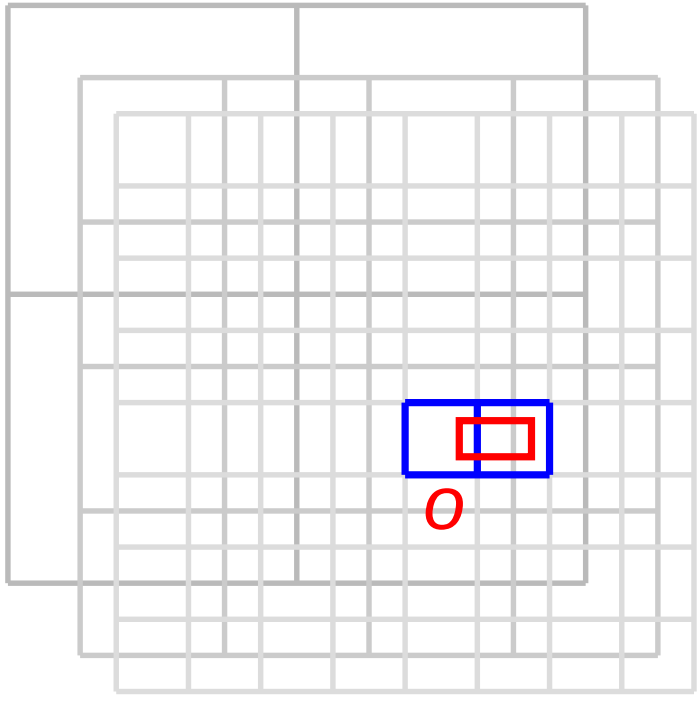
Partition Fieldtree

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



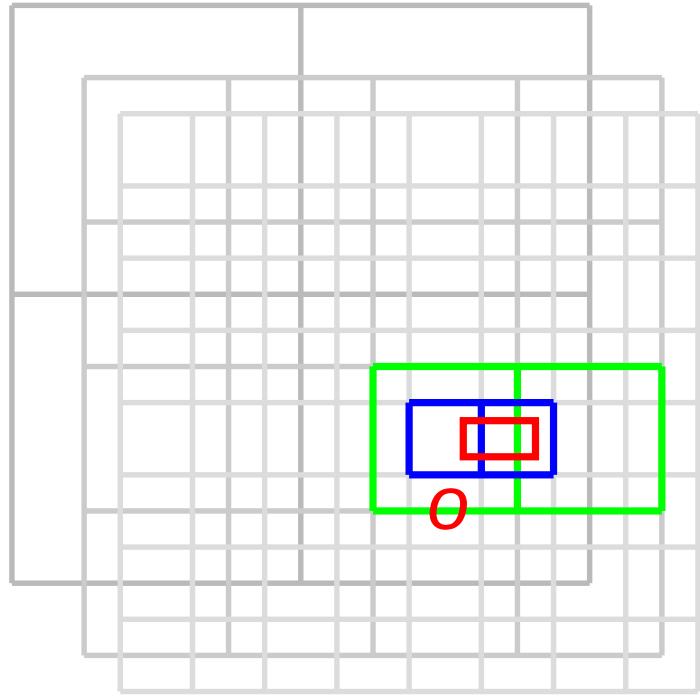
Partition Fieldtree

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



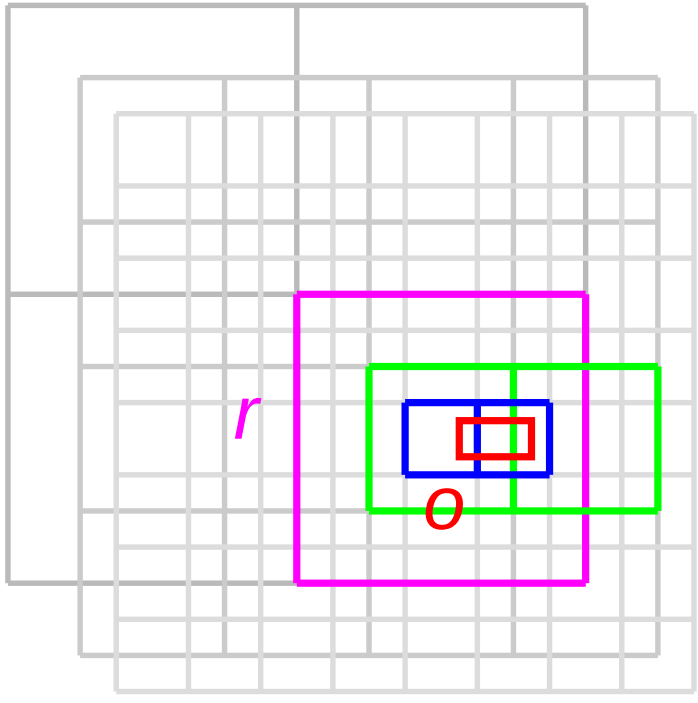
Partition Fieldtree

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



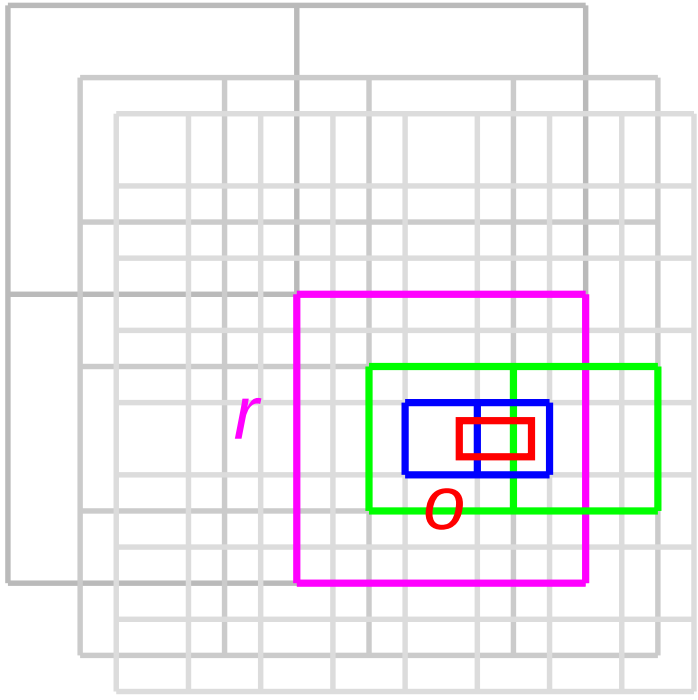
Partition Fieldtree

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



Partition Fieldtree

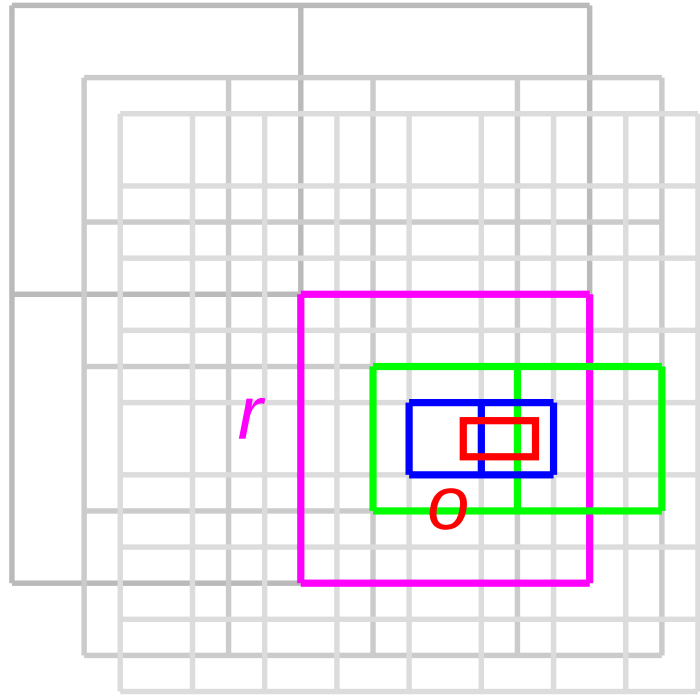
- Alternative to loose quadtree (octree)/cover fieldtree at overcoming drawback of MX-CIF quadtree that the width w of the minimum enclosing quadtree cell of a rectangle o is not a function of the size of o
- Achieves similar result by shifting positions of the centroid of quadtree cells at successive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



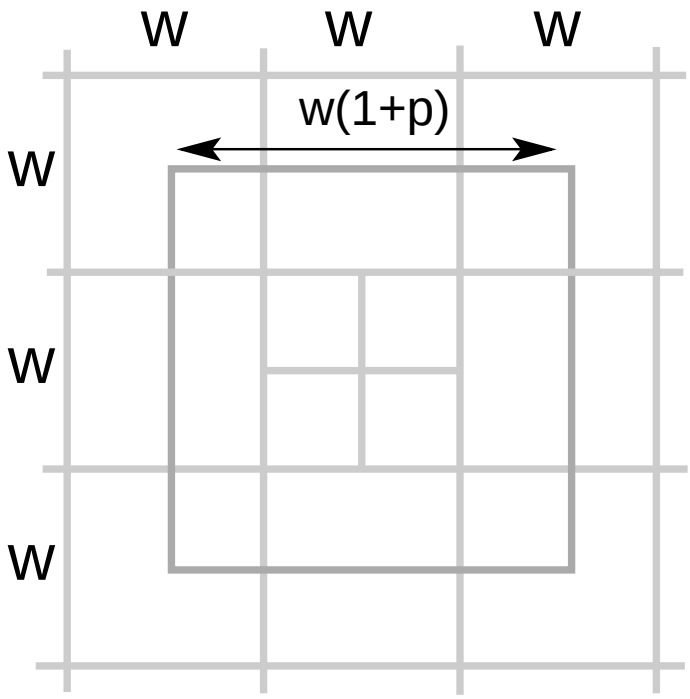
Partition Fieldtree

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

- Same ratio is obtained for the loose quadtree (octree)/cover fieldtree when p =1/4, and thus partition fieldtree is superior to the cover fieldtree when p <1/4
- Summary: cover fieldtree expands the width of the quadtree cells while the partition fieldtree shifts the positions of their centroids

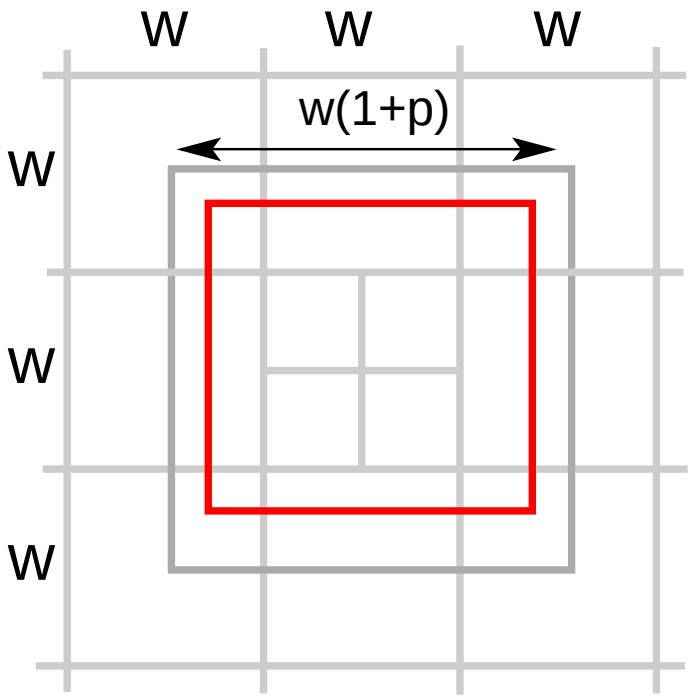


Range of Object Sizes Associated with a Quadtree Cell



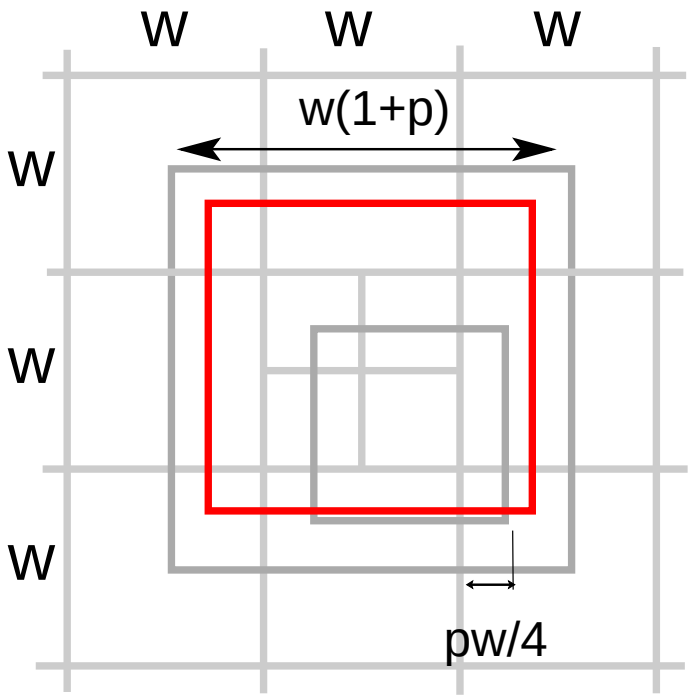
- Key is to vary the position of the centroid of the object vis-a-vis the expanded cell

Range of Object Sizes Associated with a Quadtree Cell



- Key is to vary the position of the centroid of the object vis-a-vis the expanded cell
- The width of the object is maximized when its centroid is coincident with the centroid of the cell

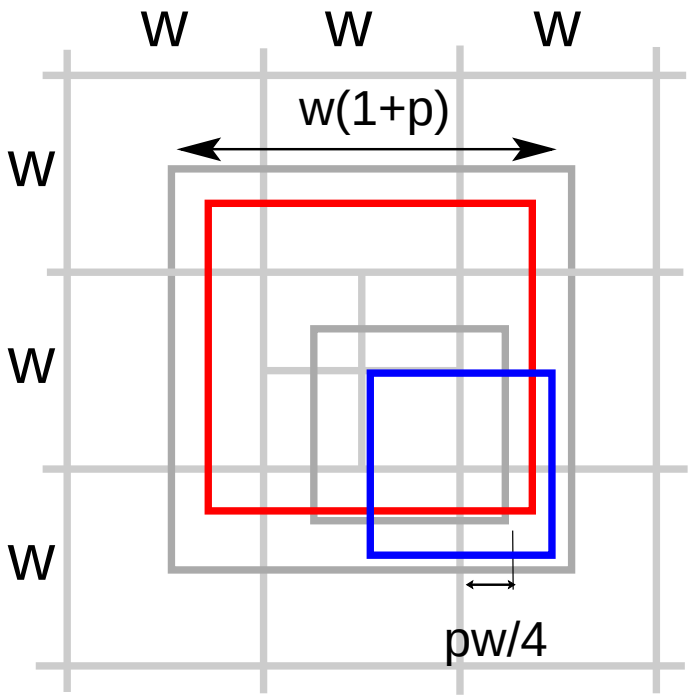
Range of Object Sizes Associated with a Quadtree Cell



- Key is to vary the position of the centroid of the object vis-a-vis the expanded cell
- The width of the object is maximized when its centroid is coincident with the centroid of the cell

- The width of the object is minimized when as much of the object as possible lies in the expanded area of the cell
  - Occurs when the centroid of object is coincident with a corner of cell

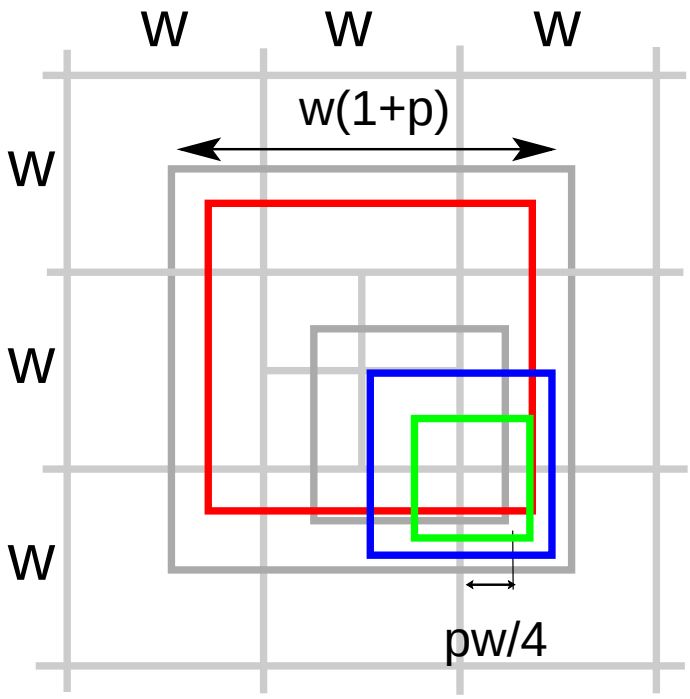
Range of Object Sizes Associated with a Quadtree Cell



- Key is to vary the position of the centroid of the object vis-a-vis the expanded cell
- The width of the object is maximized when its centroid is coincident with the centroid of the cell

- The width of the object is minimized when as much of the object as possible lies in the expanded area of the cell
  - Occurs when the centroid of object is coincident with a corner of cell
  - Object cannot be too big so that it would belong to the parent of the expanded cell

Range of Object Sizes Associated with a Quadtree Cell

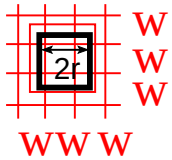


- Key is to vary the position of the centroid of the object vis-a-vis the expanded cell
- The width of the object is maximized when its centroid is coincident with the centroid of the cell

- The width of the object is minimized when as much of the object as possible lies in the expanded area of the cell
  - Occurs when the centroid of object is coincident with a corner of cell
  - Object cannot be too big so that it would belong to the parent of the expanded cell
  - Object cannot be too small so that it would belong to a child of expanded cell

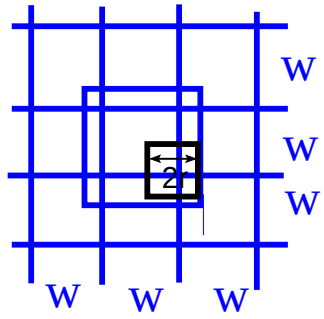
Quadtree Cell Sizes Associated with an Object

Quadtree Cell Sizes Associated with an Object



- Width w of cell c is minimized when centroid of object with radius r is coincident with centroid of c and object fits in the expanded cell of c
  - $2r \leq (1+p)w$ or $w \geq 2r/(1+p)$ (attainable) or $1/(1+p) \leq w/2r$

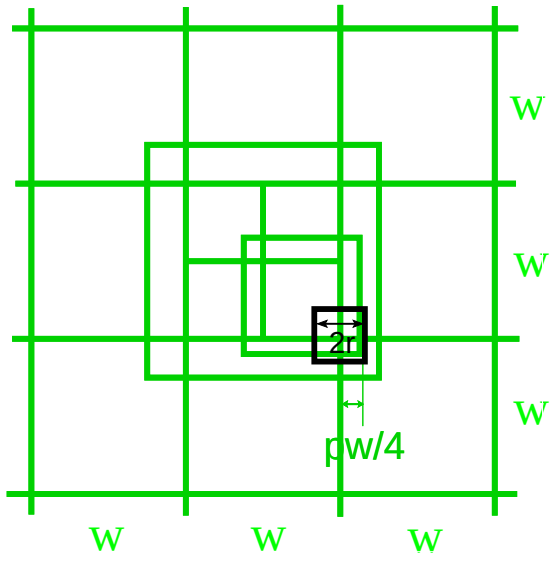
Quadtree Cell Sizes Associated with an Object



- Width w of cell c is minimized when centroid of object with radius r is coincident with centroid of c and object fits in the expanded cell of c
  - $2r \leq (1+p)w$ or $w \geq 2r/(1+p)$ (attainable) or $1/(1+p) \leq w/2r$
- Width w of cell c is maximized when centroid of object with radius r is coincident with a corner of c AND

1. The cell is just big enough so that the object does not overlap the expanded part of c so that it would require a cell of width $2w$, while
   - $r \leq pw/2$ or $w \geq 2r/p$ (attainable) or $w/2r \geq 1/p$

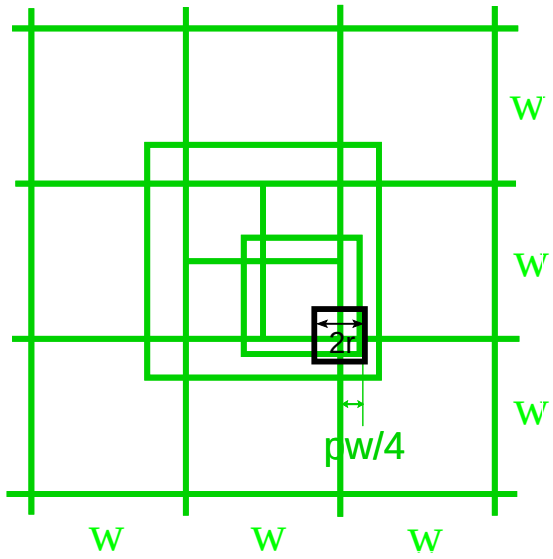
Quadtree Cell Sizes Associated with an Object



- Width w of cell c is minimized when centroid of object with radius r is coincident with centroid of c and object fits in the expanded cell of c
  - $2r \leq (1+p)w$ or $w \geq 2r/(1+p)$ (attainable) or $1/(1+p) \leq w/2r$
- Width w of cell c is maximized when centroid of object with radius r is coincident with a corner of c AND

1. The cell is just big enough so that the object does not overlap the expanded part of c so that it would require a cell of width $2w$, while
   - $r \leq pw/2$ or $w \geq 2r/p$ (attainable) or $w/2r \geq 1/p$
2. The cell is just large enough so that object is too large to fit in the expanded child cell of c
   - $r > pw/4$ or $w < 4r/p$ (attainable) or $w/2r < 2/p$

Quadtree Cell Sizes Associated with an Object



- Width w of cell c is minimized when centroid of object with radius r is coincident with centroid of c and object fits in the expanded cell of c
  - $2r \leq (1+p)w$ or $w \geq 2r/(1+p)$ (attainable) or $1/(1+p) \leq w/2r$
- Width w of cell c is maximized when centroid of object with radius r is coincident with a corner of c AND

1. The cell is just big enough so that the object does not overlap the expanded part of c so that it would require a cell of width $2w$, while
   - $r \leq pw/2$ or $w \geq 2r/p$ (attainable) or $w/2r \geq 1/p$
2. The cell is just large enough so that object is too large to fit in the expanded child cell of c
   - $r > pw/4$ or $w < 4r/p$ (attainable) or $w/2r < 2/p$

- We have proved, Theorem 3.1: $2r/(1+p) \leq w < 4r/p$ or $1/(1+p) \leq w/2r < 2/p$
  - Retain $2r/(1+p)$ instead of $2r/p$ as a lower bound on w as both are attainable and $2r/(1+p) < 2r/p$

Loose Quadtree Insertion

- Definition: Loose quadtree associates an object o with the smallest expanded quadtree cell c containing all of o's minimum bounding hypercube b in its entirety
- No need to start search at root
  - Instead search at largest possible quadtree cell that can contain o
- c always contains the centroid of b (and without loss of generality o) as otherwise a smaller expanded quadtree cell contains b
- Insertion algorithm need only search for smallest quadtree cell that contains the centroid of b and whose expanded cell also contains b

Mechanics of Loose Quadtree Insertion

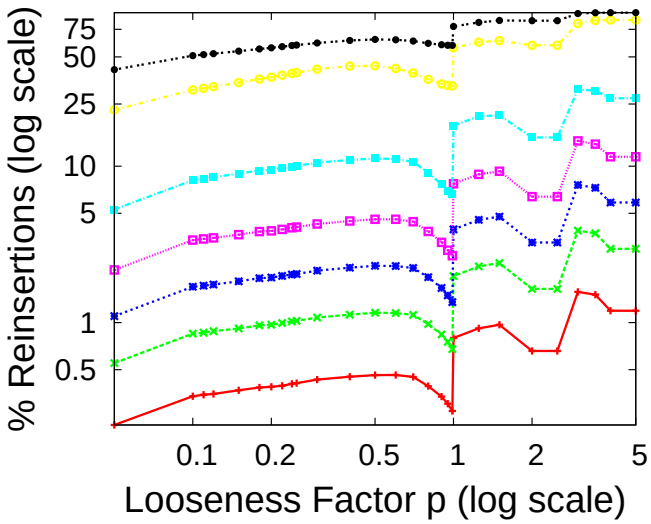
- Theorem 3.1: $1/(1+p) < w/2r \leq 2/p$ shows given a minimum bounding hypercube with radius r, width w of associated loose quadtree cell c lies within a small range of values thereby limiting number of possible cells needing testing for the inclusion of b and no need to start the search at the root of the loose quadtree as is the case for the MX-CIF quadtree
- Let $M(x) = 2^k,$ s.t. $2^{k-1} < x \leq 2^k$
- w is always a power of 2 and use the ceiling power of 2 for $w/2r$ so that Theorem 3.1 can be expressed in terms of levels
  - Let $w = 2r$: means minimum bounding box b is always a quadtree cell
  - Not critical but ensures objects can always make small motions in the expanded quadtree cell containing them without needing reinsertion
  - Explanation for optimal behavior at p=0.999 vis-a-vis $p = 1$
- $V = \log_2(M(2/p)) - \log_2(M(1/(p+1)))$ is an upper bound on the number of levels which might contain the expanded minimum containing quadtree cell of the object and these are the levels that must be searched
  - Example: $V = 3$ for $p = 1/4$, $V = 2$ for $p = 1/2$, $V = 2$ for $p = 1$, $V = 1$ for $p = 2$, $V = 2$ for $p = 3$, etc.

Experimental Evaluation

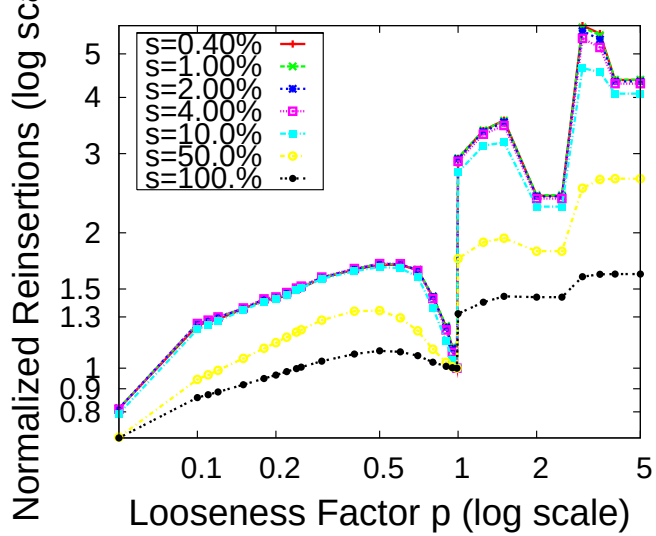
- Linux (2.6.18) quad 1.86 GHz Xeon server with four gigabyte of RAM
- Algorithms implemented using GNU C++
- Many of the experiments used a collection of random rectangles obtained by generating their centroid and extents are random
  - Same method used by Ulrich in his tests
  - Objects stored using a Morton representation indexed by a B-tree
  - Updates are efficient as only need to update the index if the object's associated expanded quadtree cell changes
  - Compare with other indexes such as the R-tree which usually require a complete rebuild when the positions of the objects changes

Reinsertion Rates - Random Data

- Vary p, maximum translation s, and fixed relative object sizes δ
- Uniformly varying translation (Same results for fixed)



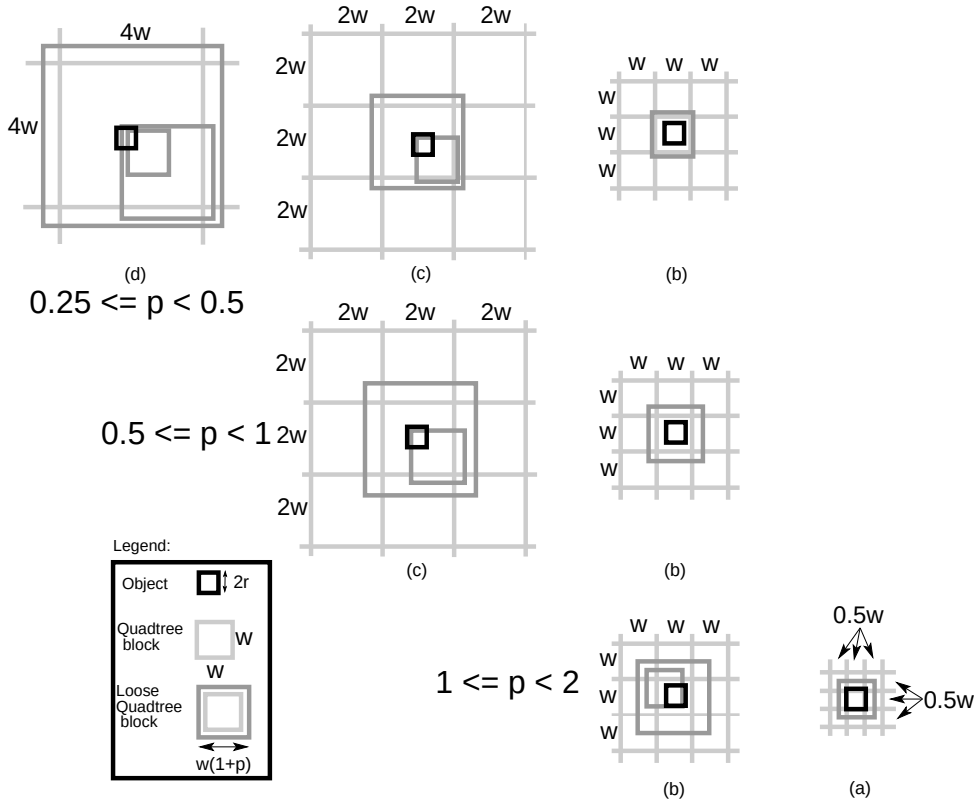
Regular



Normalized wrt p=0.999

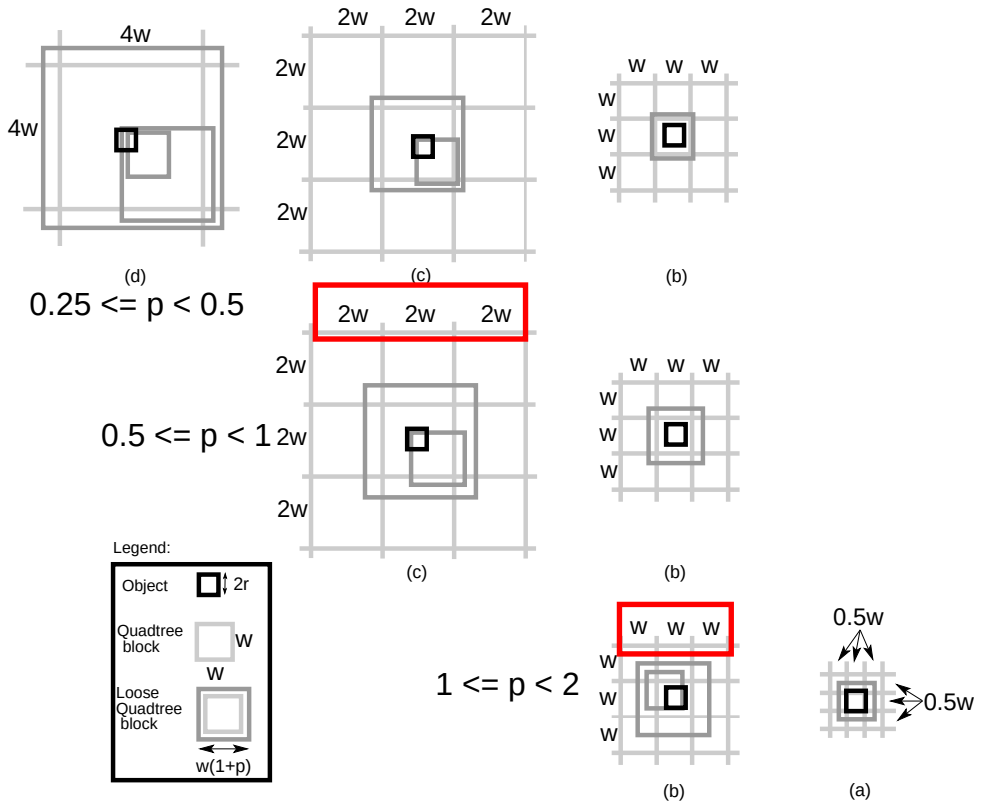
- Number requiring reinsertion increases with s (as expected) and p
- Increasing p enables objects to be associated with expanded quadtree cells with smaller widths thereby reducing the area in which the object can move without requiring reinsertion
- Increase up to $p = 0.5$ and then a precipitous drop to a minimum at p=0.999 where result is comparable to $p = 0$ and followed by a significant gain at $p = 1$ and rest recapture any earlier drops

Rationale for Sweet Spot at p=0.999



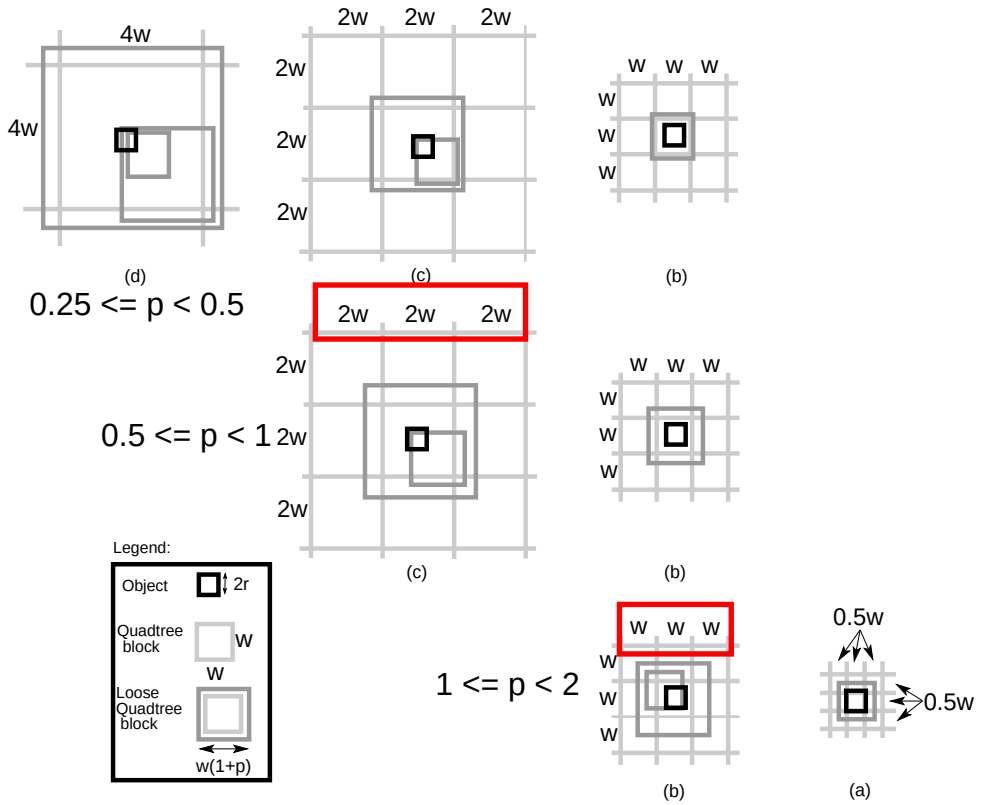
- Given an object o of a particular size, the range of sizes of the blocks that contain it is 2 for both p=0.999 and $p = 1$ but the area spanned by the largest expanded quadtree cell that contains o for p=0.999 is

Rationale for Sweet Spot at p=0.999



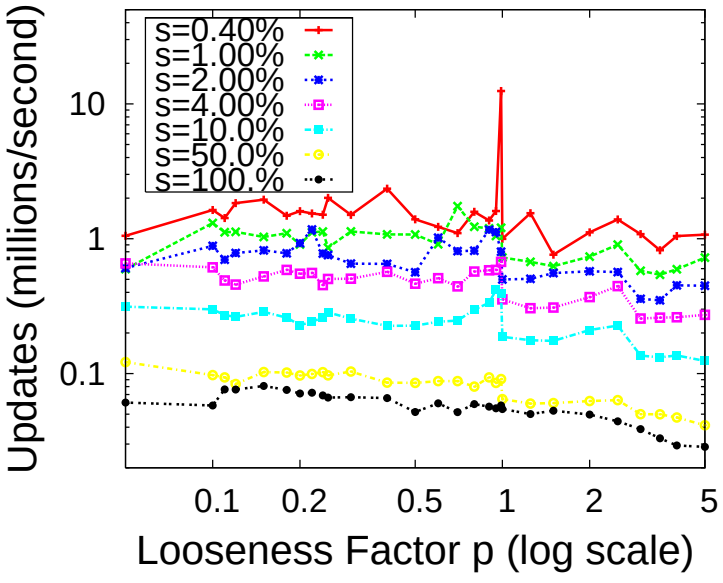
- Given an object o of a particular size, the range of sizes of the blocks that contain it is 2 for both p=0.999 and $p = 1$ but the area spanned by the largest expanded quadtree cell that contains o for p=0.999 is four times the corresponding expanded quadtree cell for $p = 1$

Rationale for Sweet Spot at p=0.999

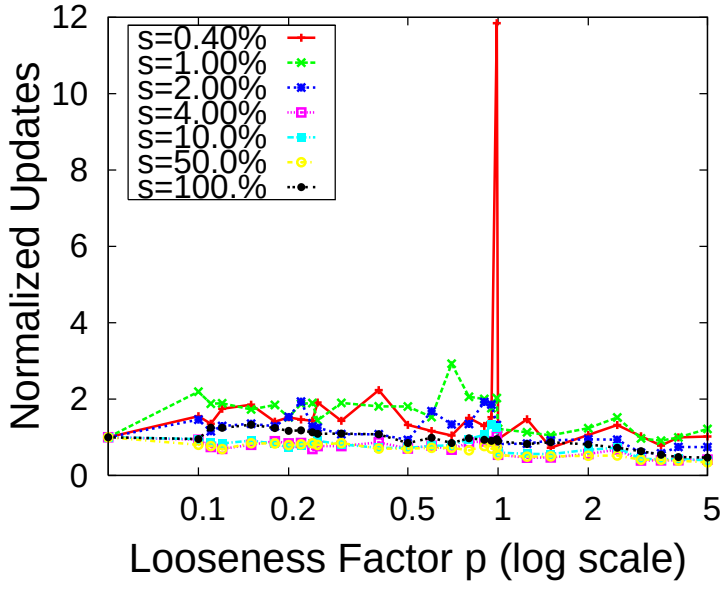


- Given an object o of a particular size, the range of sizes of the blocks that contain it is 2 for both p=0.999 and $p = 1$ but the area spanned by the largest expanded quadtree cell that contains o for p=0.999 is four times the corresponding expanded quadtree cell for $p = 1$
- Implies much larger area for p=0.999 in which object can move without needing reinsertion and also less work in updating so greater update rate
- Reinsertion rate decreases from $p \geq 0.5$ as width of expanded quadtree cells is increasing

Update Rate Performance - Random Data



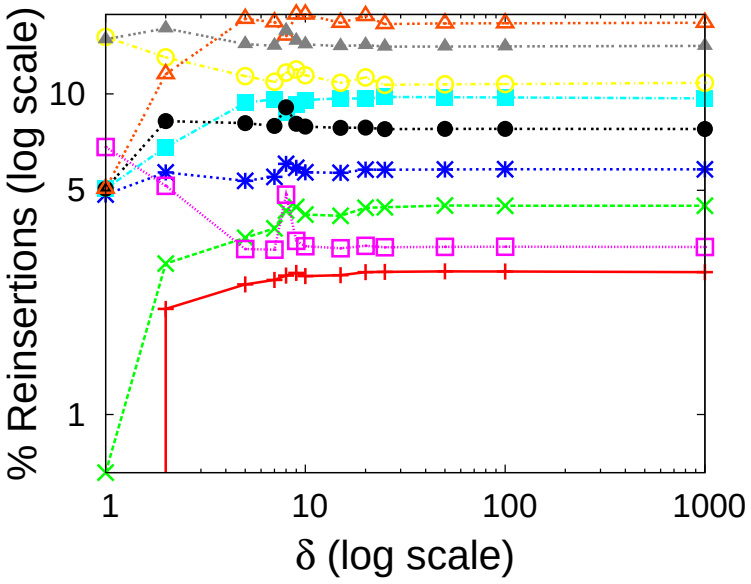
Regular



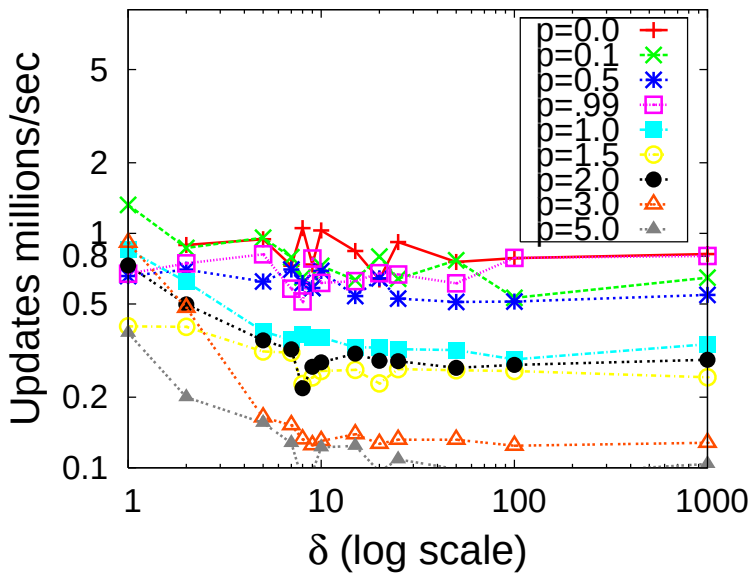
Normalized wrt $p = 0$

- Update rate decreases as s increases while little variation with p
- Normalize with result for $p = 0$ to show the effect of reduced cost of insertion as new cell can be determined in far fewer lookups (2 for $p = 1$) resulting in more updates for all but large s
- Assume fixed translation with similar result for uniform translations

Effect of Object Size - Random Data



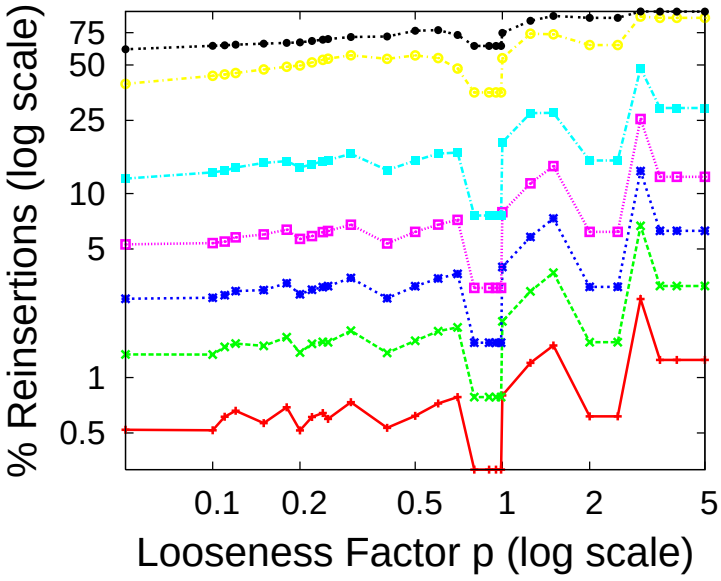
% Reinsertion



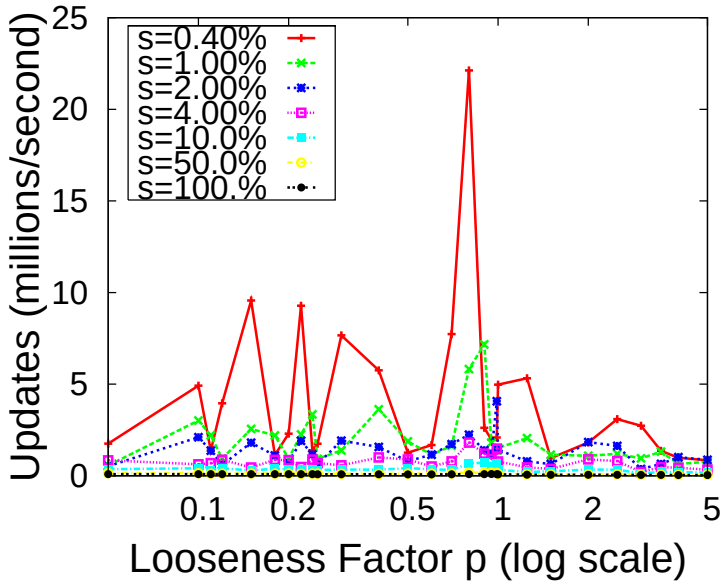
Update Rate

- Percent needing reinsertion and update rate are independent of relative object size
- Percent needing reinsertion decreased with increasing p
- Update rate decreases with increasing p and the percent needing reinsertion increases with increasing p

Reinsertion and Updates - Real Data



% Reinsertion



Update Rate

- Uniform translation - similar for fixed translation
- Update rate soars at p=0.999 due to drop in reinsertions
- Lowest variation in translation s has best performance in reinsertion (low) and update (high)

N-Body Simulation Experiments

- Designed to replicate common conditions for object dimension, object placement, and object movement based on modern video games using an N-body simulation
- Consists of insertion, deletion, update, range, and collision detection queries on the data structure that stores them
- Record time needed to render a fixed number of frames, as well as operation statistics for different numbers of objects and values of p
- During each frame, objects are allowed to move, receive force from external sources and collide with each other
- Every object is processed in physics engine and objects are stored using a loose quadtree with values of expansion factor p ranging from 0 to 1.0
- Physics engine: processes every object in the simulation in every frame
- If an object is moving, then engine performs a range query on the area occupied by object's bounding hypercube box to return a list of hit objects
- Each hit object is then entered into a physics equation and the object's velocities and positions (i.e., trajectories) are updated
- At any given time, about 50% of the objects are moving over an entire 1 kilometer squared game universe

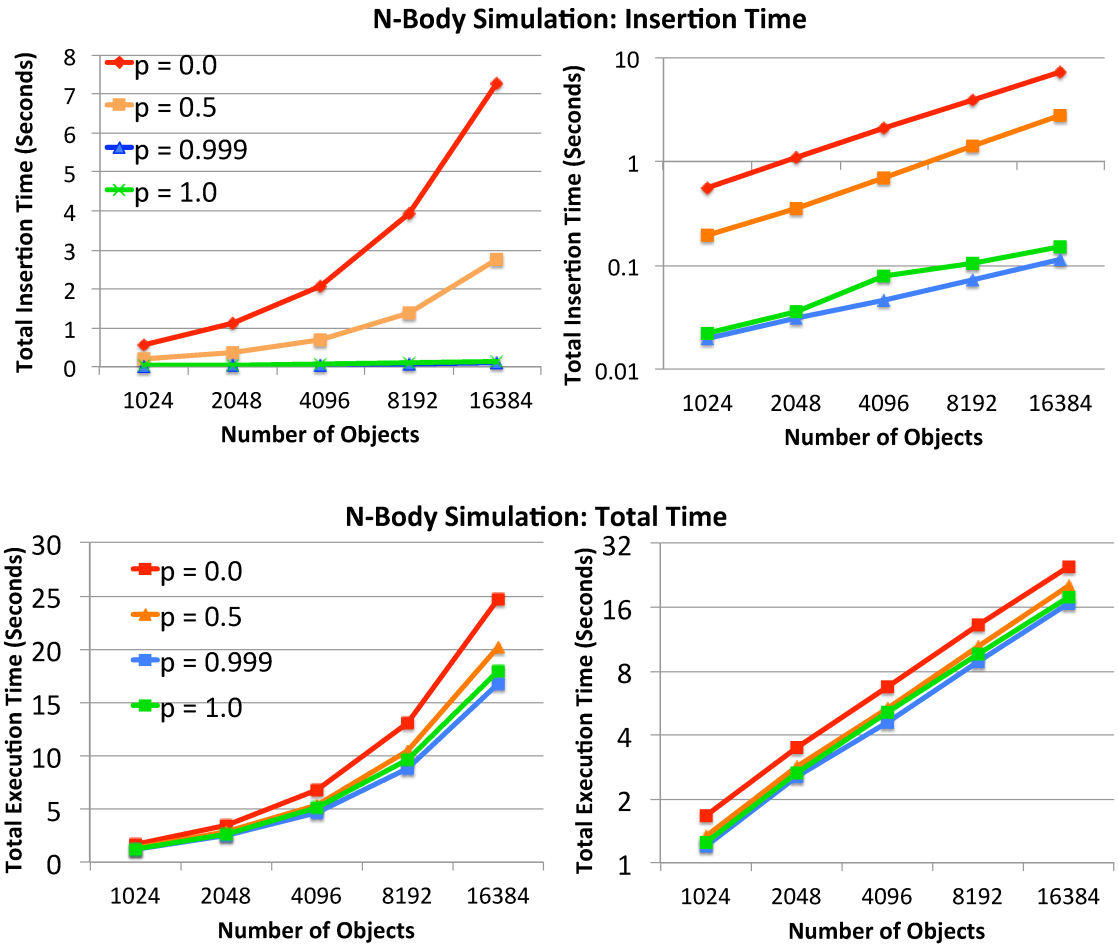
Computing Environment

- Windows 7 enterprise quad core 2.6GHz I7 workstation with eight gigabytes of 1600MHz DDR3 RAM and an Nvidia GT650m discrete GPU
- Compiled simulation and related code using Microsoft Visual Studio 2010's integrated 32 bit compiler
- Compared the MX-CIF quadtree ($p = 0$) with variants of the loose quadtree for a few values of p ($p = 0.5$, p=0.999 and $p = 1$).
- Used a pointer-based quadtree implementation
  - Simplifies algorithms as main focus was evaluating the efficacy of loose quadtree for moving object applications such as, but not limited to, video games
  - Pointer-based environment enables all execution to occur in main memory which is the environment used in video games

N-Body Simulation Details

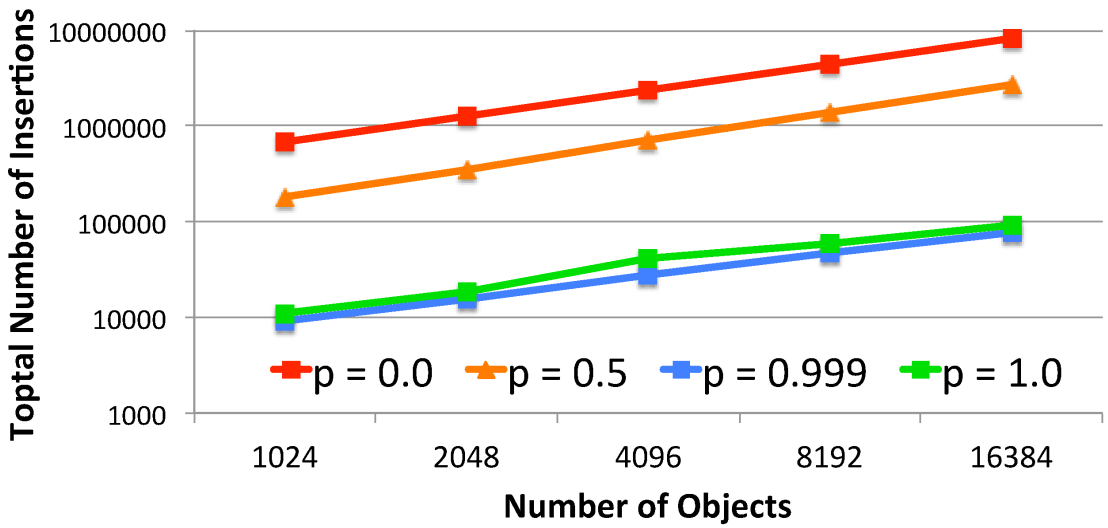
- Two-dimensional objects that interact with one another
- Each object exerts a force on all the other objects in scene and consequently each object moves due to the resulting force that is applied to it
- Computing force interaction between objects is single most time-consuming operation
- Periodically, a large radial force is exerted on system
- Resulting explosions are designed to simulate common game play cases where movement of many objects occurs in only a subset of the game universe
- However, since all objects move by varying amounts, updating the spatial data structure that indexes objects can be a significant bottleneck

Insertion and N-Body Simulation Costs



- One half order of magnitude difference in total N-body simulation execution times between $p = 0$ and p=0.999.
- Savings in insertion costs (≈ 7 seconds) account for difference between total time for N-body simulation with p=0.999 vs: $p = 0$

Number of Reinsertions vs: Number of Objects

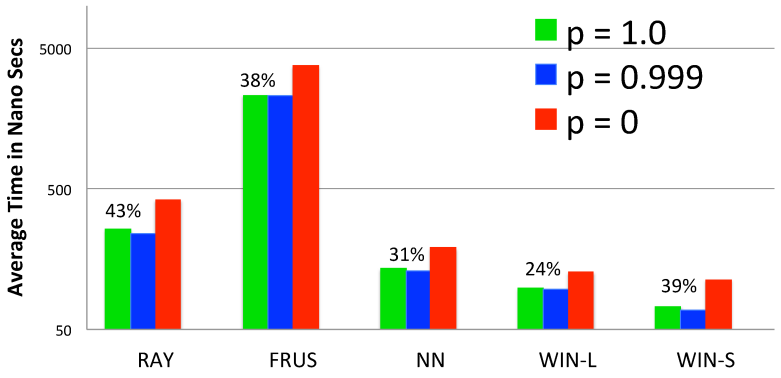


- Relatively large reduction in number of objects requiring reinsertion for p=0.999 vs: $p = 0$ (2 orders of magnitude difference)
- Motion of objects is very small and sensitivity of p=0 and p=0.5 to position result in large number of reinsertions vis-a-vis p=0.999 and p=1
- Log plots all show a linear relationship which means a power law of the form $y = ax^b$ where a and b constants with $1.60 < b < 1.75$
- 30% difference in insertions and time spent doing it, and 10% difference in execution time of simulation for p=0.999 vs: $p = 1.0$ which is less as simulation is much more complex than insertion

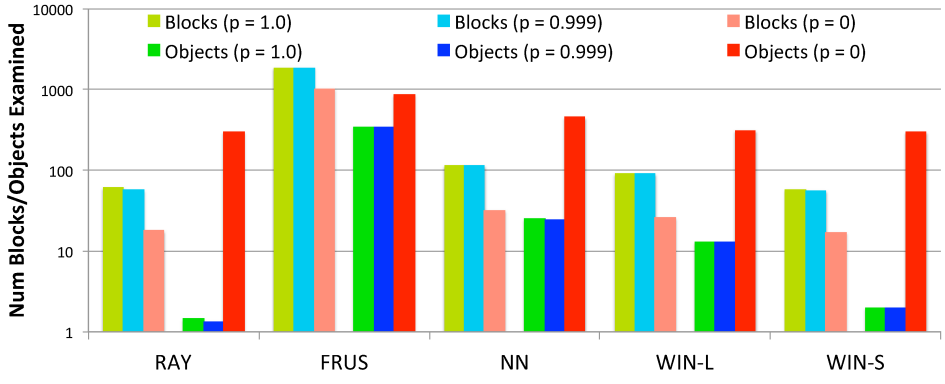
Query Performance and Scalability

- Motivation: showing that the good update performance (i.e., minimization of reinsertions upon object motion) does not come at expense of query performance, and that it scales
  1. Measure time to perform different queries
  2. Record number of blocks and objects visited by each query
  3. Examine performance on simulations with varying sizes of objects
- Show loose quadtree can scale to really large simulations way beyond limitations of MX-CIF quadtree (i.e., a loose quadtree with $p = 0$)
- Recreate a game environment to simulate a complex 3D scene
- Objects do not exert force on one another but move due to gravity
- Bottleneck is not computing inter-object forces but rather capturing the interactions between objects as they collide
  - Need to know which objects are in close proximity
- All objects are updated dynamically using a physics simulation solver
- Must render scene which requires applying geometrical operations such as ray tracing (RAY), view frustum culling (FRUS), nearest neighbor search (NN), and window queries for large (WIN-L) and small (WIN-S) windows on the spatial data structures.

Example



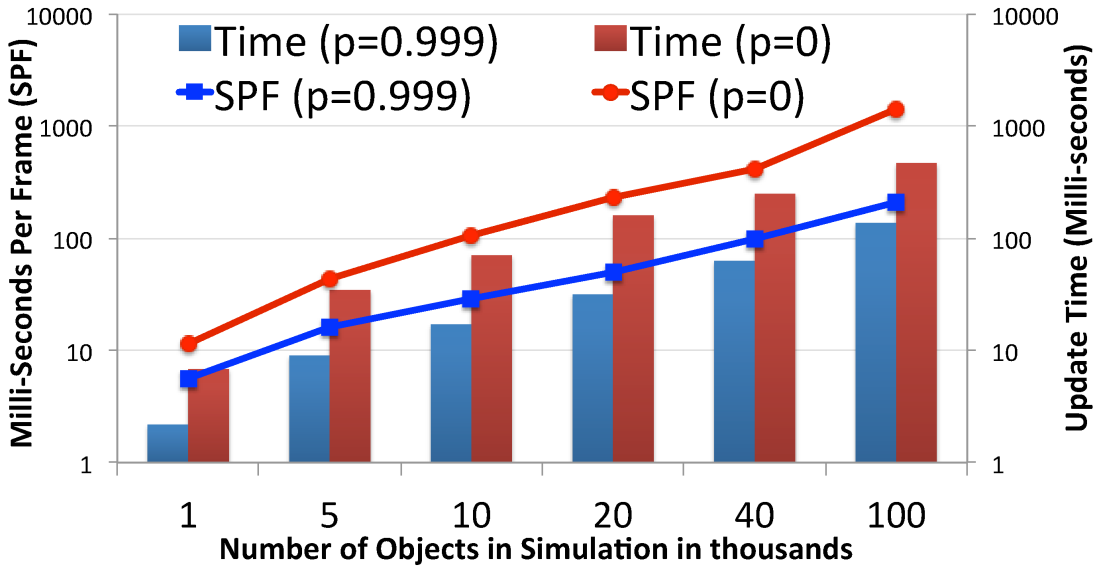
Query Times



Number of Blocks and Objects Visited

- 14,000 objects
- Loose quadtree always better than MX-CIF quadtree and by as much as 43%
- p=0.999 always better than $p = 0$ and also better than $p = 1$ (not shown)
- Both p=0.999 and 1 visit more blocks than $p = 0$ and less objects as sensitivity to position means larger blocks (but fewer) for $p = 0$ while visiting many more objects
  - Factors tend to negate each other but loose quadtree still better

Example: Rendering and Data Structure Update



- Loose quadtree renders frames 2–5 times faster than the MX-CIF quadtree
  - For 100,000 moving objects: loose quadtree can render 5 fps (210 ms/frame) in contrast to MX-CIF quadtree at 0.7 fps (1.4 sec/frame)
  - Current commercially available games typically do not involve more than 15,000 moving objects
- Time to update MX-CIF quadtree is at least 3 times more expensive when compared to updating loose quadtree with $p \approx 1$
- Clearly loose quadtree scales significantly vis-a-vis state of the art

R-tree

- Used Hilbert R-tree as easy to build
- Batched updates as otherwise reinserting one object at a time was several orders of magnitude slower than one at a time
  - For 14,000 objects, 107 ms for updating R-tree vs: 22.11 ms for p=0.999 (five times faster) and 82.26 ms for $p = 0$ (25% slower)
- Window Queries

Structure	WIN-L	WIN-S
R-tree	167 ns	142 ns
MX-CIF ($p = 0$)	128 ns	113 ns
Loose Quadtree p=0.999	98 ns	73 ns

Concluding Remarks

- Showed a variant of a data structure that associates an object with its minimum enclosing expanded quadtree cell such that the size of the cell is independent of the position of the object and only dependent on the object's size and the expansion factor
- Implies no search for the cell
- Enables representing a database of moving objects so that motion of the object usually does not require the index to be updated as the associated expanded cell is often the same
- Shown optimal behavior when expansion factor is 0.999

Postmortem

- Moral: If at first, don’t succeed try again! How many times?

Postmortem

- Moral: If at first, don’t succeed try again! How many times?
  1. TODS 2006: Proof of position independence - not enough! Need experiments

Postmortem

- Moral: If at first, don’t succeed try again! How many times?
  1. TODS 2006: Proof of position independence - not enough! Need experiments
  2. TODS 2007: Added applets showing functioning - not enough! Want experiments

Postmortem

- Moral: If at first, don’t succeed try again! How many times?
  1. TODS 2006: Proof of position independence - not enough! Need experiments
  2. TODS 2007: Added applets showing functioning - not enough! Want experiments
  3. VLDB 2008: Trivial - proof just used high school math

Postmortem

- Moral: If at first, don’t succeed try again! How many times?
  1. TODS 2006: Proof of position independence - not enough! Need experiments
  2. TODS 2007: Added applets showing functioning - not enough! Want experiments
  3. VLDB 2008: Trivial - proof just used high school math
  4. SIGMOD 2009: Ulrich did it!

Postmortem

- Moral: If at first, don't succeed try again! How many times?
  1. TODS 2006: Proof of position independence - not enough! Need experiments
  2. TODS 2007: Added applets showing functioning - not enough! Want experiments
  3. VLDB 2008: Trivial - proof just used high school math
  4. SIGMOD 2009: Ulrich did it!
  5. SIGMOD 2010: Already in Samet's book - can't counter due to double blind reviewing

Postmortem

- Moral: If at first, don't succeed try again! How many times?
  1. TODS 2006: Proof of position independence - not enough! Need experiments
  2. TODS 2007: Added applets showing functioning - not enough! Want experiments
  3. VLDB 2008: Trivial - proof just used high school math
  4. SIGMOD 2009: Ulrich did it!
  5. SIGMOD 2010: Already in Samet's book - can't counter due to double blind reviewing
  6. SIGMOD 2011: Detailed experiments and p=0.999 - wanted R-tree comparison

Postmortem

- Moral: If at first, don't succeed try again! How many times?
  1. TODS 2006: Proof of position independence - not enough! Need experiments
  2. TODS 2007: Added applets showing functioning - not enough! Want experiments
  3. VLDB 2008: Trivial - proof just used high school math
  4. SIGMOD 2009: Ulrich did it!
  5. SIGMOD 2010: Already in Samet's book - can't counter due to double blind reviewing
  6. SIGMOD 2011: Detailed experiments and p=0.999 - wanted R-tree comparison
  7. ICDE 2011: ...

Postmortem

- Moral: If at first, don't succeed try again! How many times?
  1. TODS 2006: Proof of position independence - not enough! Need experiments
  2. TODS 2007: Added applets showing functioning - not enough! Want experiments
  3. VLDB 2008: Trivial - proof just used high school math
  4. SIGMOD 2009: Ulrich did it!
  5. SIGMOD 2010: Already in Samet's book - can't counter due to double blind reviewing
  6. SIGMOD 2011: Detailed experiments and p=0.999 - wanted R-tree comparison
  7. ICDE 2011: ...
  8. VLDB 2012: Claim that good update at expense of query performance

Postmortem

- Moral: If at first, don't succeed try again! How many times?
  1. TODS 2006: Proof of position independence - not enough! Need experiments
  2. TODS 2007: Added applets showing functioning - not enough! Want experiments
  3. VLDB 2008: Trivial - proof just used high school math
  4. SIGMOD 2009: Ulrich did it!
  5. SIGMOD 2010: Already in Samet's book - can't counter due to double blind reviewing
  6. SIGMOD 2011: Detailed experiments and p=0.999 - wanted R-tree comparison
  7. ICDE 2011: ...
  8. VLDB 2012: Claim that good update at expense of query performance
  9. SIGMOD 2013: N-body game simulation example - BINGO!