Indexing Methods for Game Databases Tuned for Reducing Motion Update Times

ABSTRACT

Game applications require keeping track of objects that move and thus the database of objects must be constantly updated. The cover fieldtree (more commonly known as the loose quadtree and the loose octree, depending on the dimension of the underlying space) is designed to overcome the drawback of spatial data structures that associate objects with their minimum enclosing quadtree (octree) cells which is that the size of these cells depends more on the position of the objects and less on their size. In fact, the size of these cells may be as large as the entire space from which the objects are drawn. The loose quadtree (octree) achieves this by expanding the size of the space that is spanned by each quadtree (octree) cell c of width w by a cell expansion factor \( p (p > 0) \) so that the expanded cell is of width \((1 + p) \cdot w \) and an object is associated with its minimum enclosing expanded quadtree (octree) cell. The novelty our work lies in our demonstration that the loose quadtree, the maximum possible width \( w \) of \( c \) given an object \( o \) with minimum bounding hypercube box \( b \) of radius \( r \) (i.e., half the length of a side of the hypercube) is determined as just a function of \( r \) and \( p \), and is independent of the position of \( o \). More importantly, we explore the range of possible ratios of the width of the cell and the width of the minimum bounding hypercube box \( w/2r \) as a function of \( p \), and prove that for \( p \geq 0.5 \), \( w/2r \) takes on at most two values, and usually just one value for \( p \geq 1 \). This makes updating very simple and fast as for \( p \geq 0.5 \), there are at most two possible new cells associated with the moved object. In particular, since such quadtrees are usually implemented with a pointerless representation with the aid of a B-tree, motion updates are reduced to B-tree update operations in contrast to more complicated, and significantly slower, update operations as is the case with representations such as an R-tree. Experiments show that the update time to support motion in such an environment is minimized when \( p \) is infinitesimally less than \( 1 \) with as much as a one order of magnitude increase in the number of updates that can be handled vis-a-vis the \( p = 0 \) case in a given unit of time.

Keywords

game databases, moving objects, spatial data structures, cover fieldtree, loose quadtree, loose octree, spatial indexing, spatial databases, game programming

1. INTRODUCTION

Game applications (e.g., 3, 18), require keeping track of objects that move and thus the database of objects must be constantly updated. An attractive method of representing spatial objects to support the tracking process uses an object hierarchy where minimum bounding boxes (e.g., an R-tree 17, 27) are used to speed up the process of detecting if objects are present or overlap other objects. One of the drawbacks of such a representation is that the hierarchies of different sets of objects are not in registration thereby making set operations between the two sets such as unions and intersections more complex.

A solution is to use a hierarchy of congruent cells while still not decomposing the objects. In this case, the hierarchy is based on a regular decomposition of the underlying space such as a region quadtree (e.g., 24) and then associating each object with its minimum enclosing quadtree cell. Methods that employ this technique include the MX-CIF quadtree 4, 21, multilayer grid file 32, R-file 20, filter tree 29 (used for spatial join algorithms), and SQ-histogram 4 (used for selectivity estimation in processing spatial queries) where the primary difference lies in the nature of the access structure that is used. For example, Figure 4 is an MX-CIF quadtree for a collection of rectangle objects, where we see that more than one object is associated with some of the nodes in the tree which means that the objects have the same minimum enclosing quadtree cell (e.g., the root and its NE child, where the children are referred to as NW, NE, SW, and SE denoting the Northwest, Northeast, Southwest, and Southeast quadrants, respectively, of corresponding cells).

The drawback of these methods is that the size of these minimum enclosing quadtree cells depends on the position of the centroids of the objects and is independent of the size of the objects, subject to a minimum which is the size of the object. In fact, it may be as large as the entire space from which the objects are drawn. This has bad ramifications for applications where the objects move including games, traffic monitoring, and streaming. In particular, if the objects are
moved even slightly, then they usually need to be reinserted in the structure.

The cover fieldtree [13] and the more commonly known loose quadtree (octree) [34] are designed to overcome this need for reinsertion drawback by expanding the size of the space that is spanned by each quadtree cell $c$ of width $w$ by a cell expansion factor $p$ ($p > 0$) so that the expanded cell is of width $(1 + p) \cdot w$ and an object is associated with its minimum enclosing expanded quadtree (octree) cell. To better understand the loose quadtree and its behavior, see the publicly available web site at http://donar.umiacs.umd.edu/quadtree/rectangles/loosequad.html.

The novelty of the work reported in this paper is based on the following contributions.

1. We demonstrate that for the loose quadtree, the maximum possible width $w$ of $c$ given an object $o$ with minimum bounding hypercube box $b$ of radius $r$ (i.e., half the length of a side of the hypercube) is determined as just a function of $r$ and $p$, and is independent of the position of $o$.

2. Our work differs from that of Ulrich [34] whose goal was to avoid associating a small object with a large minimum enclosing quadtree cell, which speeds up culling-based operations (e.g., viewing, collision detection). In particular, Ulrich focused on determining the largest object that could be associated with a given expanded cell, while we focus on the inverse problem—that is, given an object size, we seek the largest expanded cell with which it can be associated (really the ratio of the cell size to the object size). Our work is the first to focus on minimizing the ratios, which is a shortcoming of prior work [13] [20] [24] [22].

3. Most importantly, we explore the range of possible ratios of width $w/2r$ as a function of $p$, and prove that for $p \geq 0.5$, $w/2r$ takes on at most two values, and usually just one value for $p \geq 1$. This makes updating the index very simple when objects are moving as there are at most two possible new cells associated with a moved object, instead of $\log_2$ of the width of the space in which the objects are embedded (which can be as large as 16 assuming a $2^{16} \times 2^{16}$ embedding space as used by us).

4. We show an attainable lower bound of $1/(1 + p)$ on the ratios, while the construction in [34] yields a lower bound of $1/p$, which is quite different for relatively small values of $p$ as is the case here.

5. Since such quadtrees are usually implemented with a pointerless representation with the aid of a B-tree, motion updates are reduced to B-tree update operations in contrast to more complicated update operations as is the case with representations such as an R-tree, which requires expensive rebuilding algorithms. The experiments that we report show that the cost in execution time of motion updates in our environment is minimized when $p$ is infinitesimally less than 1, while the behavior for $p \geq 1$ is considerably worse. These results are very different from Ulrich who claims, without proof, that $p = 1$ is the right choice.

6. For $p = 0.99$, we observed as much as a one order of magnitude increase in the number of updates that can be handled vis-a-vis the $p = 0$ case in a given unit of time for relatively small motions, and lesser, but still noticeable improvements for all other values of $p$ (both $0 \leq p < 1$ and $p \geq 1$).

7. Ulrich [34] contains no discussion of the cost of lookup other than a mention of it being $O(1)$. He only addresses the performance of culling operations on random data for $p = 1$ vis-a-vis $p = 0$, and his experiments led him to conclude that $p = 1$ is better in this case. However, there is no discussion of insertion or the effect of varying $p$ for which we find that $p = 0.99$ is much better than $p = 1$.

The rest of this paper is organized as follows. Section 2 expands on the motivation for our work and discusses related work. Section 3 shows how to achieve position independence for the width of the minimum enclosing quadtree cell $c$ by examining the range of the relative widths of $c$ and the minimum bounding hypercube box $b$ of object $o$ and also how to take into account the constraints imposed by the fact that the range of values of the width of $c$ is limited to powers of 2. Section 4 discusses the ramifications of these results and also contains a cell insertion algorithm for the loose quadtree. Section 5 contains an experimental evaluation of the loose quadtree with respect to the extent that it needs to be updated on account of object motion for different values of $p$, object size distribution, and dimension of the space from which the objects are drawn. Concluding remarks are drawn in Section 6.

2. MOTIVATION AND RELATED WORK

As we pointed out in Section 1, one of the main drawbacks of methods such as the MX-CIF quadtree is that the maximum width $w$ of the cell $c$ corresponding to the minimum enclosing quadtree cell of object $o$'s minimum enclosing bounding box $b$ is not a function of the size of $b$ or $o$, subject to a minimum which is the size of $o$. Instead, it is dependent on the position of $o$. There are several ways of overcoming this drawback. One easy way is to introduce redundancy (i.e., representing the object several times

Figure 1: (a) Cell decomposition induced by the MX-CIF quadtree for a collection of rectangle objects and (b) its tree representation (from [27]).
thereby replicating the number of references to it) by decomposing the quadtree cell into smaller quadtree cells, each of which minimally encloses some portion of $o$ (or, alternatively, some portion of $o$'s minimum enclosing bounding box $b$) and contains a reference to $o$. The expanded MX-CIF quadtree is a simple example of such an approach where $d$ is decomposed once into four subcells $c_i$, which are then decomposed further until obtaining the minimum enclosing quadtree cell $s_i$ for the portion of $o$, if any, that is covered by $c_i$. A more general approach, used in spatial join algorithms, sets a bound on the number of replications, (termed a size bound) and used in the CESS method or on the size of the covering quadtree cells resulting from the decomposition of $c$ that contain the replicated references (termed an error bound).

Replicating the number of references to the objects is reminiscent of the manner in which the non-disjointness of the decomposition of the underlying space resulting from the use of an object hierarchy is overcome by use of an R+-tree instead of an R-tree, and thus has the same shortcoming of possibly requiring the application of a duplicate object removal step prior to reporting the answer to some queries (e.g., [14, 18]). The multiple shifted quadtree methods [7, 23] and the partition fieldtree [13, 11] and the cover fieldtree (also the equivalent loose quadtree and octree [33]) adopt different approaches at overcoming the independence of the sizes of $c$ and $b$ drawback. In particular, they do not replicate the objects.

The multiple shifted quadtree methods make use of the observation that the number of cells required for the quadtree for a rectangular object $o$ depends on the positioning of $o$ relative to the origin of the underlying space from which $o$ is drawn. This number can be reduced by applying shift operations to $o$, (e.g., [12, 30]), and, of course, is minimized when the width of $o$ is a power of 2 along all of the coordinate axes. An alternative, and equivalent, method of achieving the same result shifts the origin of the underlying space from which $o$ is drawn. For an arbitrary binary image, finding the optimal positioning of the image, in terms of minimizing the number of needed quadtree cells, has been addressed by the development of an algorithm that makes use of dynamic programming to attempt different shifts of the image in the directions of the coordinate axes.

Assuming data in $d$ dimensions, Chan [17], Liao et al. [23], and Lieberman et al. [24] make use of the above observation to achieve a bound on the ratio $H$ of the size of $c$ and the width of $b$, object $o$'s minimum bounding hypercube box, by using a set of $d + 1$ shifted quadtrees, instead of one, that guarantees that the ratio $H$ is bounded by $O(d)$ in at least one of the quadtrees and the quadtree for which this ratio is a minimum is the one in which $o$ is stored. Assuming, without loss of generality, that the underlying space $S$ from which the objects are drawn is the $d$-dimensional unit hypercube, each object $o$ in $S$ is inserted into each of the $d + 1$ quadtrees, where prior to insertion into the $i$th, $0 \leq i \leq d$ quadtree, the coordinate values of $o$ are shifted by the fraction $2^{-i}$. For example, for $d = 2$, we have three quadtrees, which are constructed as follows. The first quadtree is constructed on object set $S$, while the second quadtree is constructed after shifting all the objects in $S$ by $1/3$ across both the $x$ and $y$ coordinate axes, and the third and final quadtree is constructed after shifting all the objects in $S$ by $2/3$ across both the $x$ and $y$ coordinate axes. Notice that as the same shift vector is applied to all of the objects in a particular quadtree, but is varied across the different quadtrees, in all cases the relative positions of all of the objects in $S$ are preserved.

The rationale behind using multiple shifted quadtrees is that there may be an object and shift vector configuration that may lead to the case where the ratio $H$ is small. This may happen, for example, if the minimum bounding hypercube box $b$ of object $o$ does not intersect any of the $(d - 1)$-dimensional hyperplanes that correspond to boundaries of quadtree cells that are much larger than $b$. Note, however, that there is no easy way of identifying which of the $d + 1$ quadtrees provides the minimum value of $H$ for a given object without checking through all of shifted quadtrees. Moreover, implementations of operations on the multiple shifted quadtrees must be replicated across all of the $d + 1$ quadtrees, which may not be so attractive.

In contrast to the multiple shifted quadtrees method, the partition fieldtree and cover fieldtree (and equivalent loose quadtree and octree) methods restrict the ratio $H$ of the size of $c$ and the width of object $o$'s minimum bounding hypercube box $b$ to a small constant that is independent of $d$. This obviates the need for multiple shifted copies of the dataset, which can be cumbersome to build, maintain, and perform operations on. In particular, the partition fieldtree overcomes the independence of the sizes of $c$ and $b$ drawback by shifting the positions of the centroids of quadtree cells at successive levels of subdivision by one-half the width of the cell that is being subdivided. Figure 2 shows an example of such a subdivision. For two-dimensional data, it can be shown that this subdivision rule guarantees that the width $w$ of the minimum enclosing quadtree cell $c$ for the minimum bounding hypercube box $b$ for object $o$ is bounded by eight times the maximum width of $b$. The drawback of the partition fieldtree is that searching is more complex as given a location, in order to determine the objects associated with it, we potentially need to examine three covering cells at successive levels of decomposition.

![Figure 2: Example of the subdivision induced by a partition fieldtree (from [27]).](image)

The cover fieldtree, and the equivalent loose quadtree (loose octree in three dimensions) [33], as used in the rest of this paper and motivated by game programming applications overcomes this independence of the sizes of $c$ and $b$ drawback by expanding the size of the space that is spanned by each quadtree cell $c$ of width $w$ by a cell expansion factor $p$ ($p > 0$) so that the expanded cell is of width $(1 + p) \cdot w$. 
In this case, an object is associated with its minimum enclosing expanded quadtree cell. For example, letting \( p = 1 \), Figure 3 is the loose quadtree corresponding to the collection of objects in Figure 4(a) and its MX-CIF quadtree in Figure 4(b). In this example, there are only two differences between the loose and MX-CIF quadtrees:

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

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

![Figure 3](image)

**Figure 3:** (a) Cell decomposition induced by the loose quadtree for a collection of rectangle objects identical to those in Figure 1 and (b) its tree representation (from [22]).

Ulrich [34] has shown that given a quadtree cell \( c \) of width \( w \) and cell expansion factor \( p \), the radius \( r \) of the minimum bounding hypercube box \( b \) of the smallest object \( o \) that could possibly be associated with \( c \) must be greater than \( pw/4 \). In particular, the utility of the loose quadtree is best evaluated in terms of the inverse of this relation as we are interested in the maximum possible width \( w \) of \( c \) given an object \( o \) with minimum bounding box \( b \) of radius \( r \). This is because reducing \( w \) is the real motivation for the development of the loose quadtree as an alternative to the MX-CIF quadtree for which \( w \) can be as large as the width of the underlying space. We achieve our result in Section 3 by examining the range of the relative widths of \( c \) and \( b \) as this provides a way of taking into account the constraints imposed by the fact that the range of values of \( w \) is limited to powers of 2. Section 4 discusses the ramifications of these results and also contains a cell insertion algorithm for the loose quadtree.

### 3. Calculation of the Maximum Loose Quadtree Cell Width

A key principle to observe is that in the loose quadtree, the smallest expanded quadtree cell \( c \) of width \( w \) that contains the object \( o \) has the property that the centroid of \( o \) (actually of \( o \)'s minimum bounding hypercube box \( b \) of radius \( r \)) is contained in the non-expanded portion of \( c \). Thus insertion proceeds by finding the smallest quadtree cell \( c \) that contains the centroid of \( b \), and whose expanded cell also contains \( o \). The traditional way of finding \( c \) is to recursively search the quadtree starting at the root and descend to the appropriate child based on the value of the centroid. In fact, it turns out that there is even an easier way of determining \( c \), which involves little search (i.e., few descents in the quadtree). In particular, we show below that the width \( w \) of \( c \) must lie within a relatively small range of values, thereby greatly restricting the number of possible cells that must be tested for the inclusion of \( o \).

Recall that one of the key drawbacks of data structures such as the MX-CIF quadtree that associate an object \( o \) with the minimum sized quadtree cell \( c \) of width \( w \) that encloses the minimum bounding hypercube box \( b \) of radius \( r \) of \( o \) is that \( w \) is a function of the position of \( o \), and to a lesser extent, a function of \( r \) in the sense that only its minimum is a function of \( r \). In contrast, in the loose quadtree, as we show in the rest of this section, the dependence of \( w \) on the position of \( o \) is reduced significantly. In particular, we demonstrate that \( w \) lies within a range of values that only depend on the radius \( r \) of \( o \)'s minimum bounding hypercube box \( b \) and the value of the cell expansion factor \( p \). In fact, Theorem 3.1 shows that the ratio of the range of the widths of \( c \) and \( b \) (i.e., \( w/2r \)) is only dependent on \( p \).

**Theorem 3.1.** The ratio \( w/2r \) of the widths of the expanded minimum enclosing quadtree cell \( c \) and the minimum bounding box \( b \) of the object \( o \) obeys

\[
\frac{1}{1 + p} \leq \frac{w}{2r} < \frac{2}{p}.
\]

**Proof.** We first derive a lower bound on the range of the ratios. From the definition of the cell expansion factor \( p \), we know that given an object \( o \) with minimum bounding hypercube box \( b \) of radius \( r \), the smallest quadtree cell \( c \) of width \( w \) with which \( o \) can be associated so that \( o \)'s centroid lies in the non-expanded portion of \( c \) arises when the centroids of \( b \) and \( c \) coincide, and moreover the cell \( c' \) resulting from the expansion of \( c \) (i.e., having width \( (1 + p)w \)) is just large enough to contain \( b \) of width \( 2r \) (see Figure 1(a)). This leads to the inequality which is given below as:

\[
(1 + p)w \geq 2r \tag{1}
\]

and can be rewritten as

\[
\frac{w}{2r} \geq \frac{1}{1 + p}. \tag{2}
\]

We can use similar reasoning to obtain an upper bound on the range of the ratios, and in the process use a similar construction to that of Ulrich [34] except that for a given cell expansion factor \( p \), Ulrich assumed the existence of a quadtree cell \( c \) of width \( w \) and was seeking the radius \( r \) of the minimum bounding hypercube box \( b \) of the smallest object \( o \) that could possibly be associated with the expanded cell \( c \), while we are assuming that for a given cell expansion factor \( p \), we are given an object \( o \) with minimum bounding hypercube box \( b \) of radius \( r \) and are seeking the width \( w \) of the largest cell \( c \) with whose expanded cell \( b \) would be associated. We make use of our observation that the centroid of the object \( o \) with minimum bounding hypercube box \( b \) of radius \( r \) is always required to be contained in the non-expanded portion of the associated quadtree cell.

Given this observation, we note that the largest quadtree cell \( c \) of width \( w \) that can satisfy this requirement on the placement of the centroid has the property that one of \( c \)'s corners is coincident with the centroid of \( o \), and that the
radius \( r \) of \( b \) is not too large so that \( b \) is too large for the expanded region of \( c \) (i.e., an attainable upper bound on \( r \) of \( pw/2 \) as shown in Figure 4(b)), and just large enough so that \( b \) does not fit in the expanded region of one of the subcells of \( c \) of width \( w/2 \) (i.e., an unattainable lower lower bound on \( r \) of \( pw/4 \) as shown in Figure 4(c)). Equivalently, for this particular configuration, we say that \( pw/4 = 2^{k-1} = r - \delta' < r \leq 2^k = pw/2 \) for some value of \( k \) and \( \delta' > 0 \). Simplifying the notation by letting \( \delta' = \delta w/4 \), we have \( pw/4 = 2^{k-1} = r - \delta w/4 < r \leq 2^k = pw/2 \) for some \( \delta > 0 \). Since the width \( w \) of \( c \) is the same for all values of \( r \) in this range, we point out that \( c \)'s width relative to that of \( b \) is maximized when \( r \) takes on the value:

\[
r = pw/4 + \delta w/4, \delta > 0. \tag{3}
\]

which can be rewritten as:

\[
w/2r = \frac{w}{\frac{r}{2} + \frac{\delta w}{4}}, \delta > 0, \tag{4}
\]

\[
w/2r = \frac{2}{p + \delta} < \frac{2}{p}, \tag{5}
\]

\[
w/2r < \frac{2}{p}. \tag{6}
\]

Combining relations 4 and 5 yields the range:

\[
\frac{1}{1 + p} \leq \frac{w}{2r} < \frac{2}{p}. \tag{7}
\]

Without loss of generality, let us assume that the quadtree cell corresponding to the root of the loose quadtree has length \( 2^g \), where \( g \) is an integer. This enables us to avoid dealing with negative values of \( k \), which is somewhat counterintuitive, as would be the case were we to continue with the unit hypercube assumption. In this case, all cells \( c \) in the loose quadtree have width \( w = 2^k \), such that \( k \leq g \) is an integer. Now, for any given value \( x \), let us define a function \( M(x) \) which determines a \( k \) such that \( 2^{k-1} < x \leq 2^k \), and returns the value \( 2^k \). In other words,

\[
M(x) = 2^k, 2^{k-1} < x \leq 2^k. \tag{8}
\]

Moreover, we also have that

\[
1 \leq \frac{M(x)}{x} < 2. \tag{9}
\]

The rationale behind the function \( M(x) \) is that it quantizes \( x \) to the next higher power of 2 unless it is already a power of 2. To explain the utility of \( M(x) \) from a geometric point of view, consider an input object \( R \) with a minimum bounding hypercube box of radius \( r \). We have that \( M(r) \) is the radius of the smallest quadtree cell (i.e., half the width) that can potentially contain \( R \). We now derive the minimum and maximum possible ratios of \( w/2r \) in terms of \( M(\cdot) \). Our motivation is to be able to identify a set of quadtree cells (typically a few) in the loose quadtree that can potentially contain \( R \). From relation 5, we are given that \( w/2r \) is greater than or equal to \( 1/(p + 1) \), but is less than \( 2/p \). Consider an input object \( R \) with a minimum bounding hypercube box of radius \( r \). How many levels of the loose quadtree does the range \([1/(p + 1), 2/p]\) span? This is upper-bounded by the number of integers of the form \( 2^k \), where \( k \) is an integer, that is contained in the range \([1/(p + 1), 2/p]\). That is, we have just shown that the number of levels spanned by the range in relation 7 cannot exceed \( V \), which is given by \( \text{Lemma 3.1.} \)

**Lemma 3.1.** The number of levels in the loose quadtree at which the expanded minimum quadtree cell of the object could possibly lie is upper bounded by \( V \), where

\[
V = \log_2(M(2/p)) - \log_2(M(1/(p + 1))). \tag{10}
\]
4. DISCUSSION

Now, let us make some observations on the possible ranges of relative cell widths on the basis of relations 4 and 10. First, for the degenerate case of the MX-CIF quadtree, in which case no expansion takes place (i.e., \( p = 0 \)), we have an unbounded upper bound on the range of values and a lower bound of 1. As \( p \) increases towards 1, the range of values decreases. For example, for \( p = 1/4 \), we have a range of relative cell widths \([4/5, 8] \). This means that the relative cell widths of the set of possible quadtree cells containing a given input rectangle \( R \) with a minimum bounding hypercube box of radius \( r \) lie between \( [M(4/5) = 1, M(8) = 8] = [1, 2, 4] \). In other words, the quadtree cells containing \( R \) in the loose quadtree can be of radius \( M(r) \), \( 2M(r) \), and \( 4M(r) \) (i.e., half the width). In fact, these radii hold for all values of \( p \) such that \( 1/4 \leq p < 1/2 \).

For \( p = 1/2 \), there are just two possible relative cell widths corresponding to \([M(2/3) = 1, M(4) = 4] = [1, 2] \). In other words, the associated quadtree cells of \( R \) can be either the quadtree cell of radius \( M(r) \) or of radius \( 2M(r) \). These radii hold for all values of \( p \) such that \( 1/2 \leq p < 1 \). For \( p = 1 \), there are also just two possible relative cell widths corresponding to \([M(1/2) = 1/2, M(2) = 2] = [1/2, 1] \). In other words, the associated quadtree cells of \( R \) can be either the quadtree cell of radius \( M(r) \), or can be of radius half of \( M(r) \). These radii hold for all values of \( p \) such that \( 1 \leq p < 2 \). As \( p \) increases beyond 1, the number of possible ratios of relative cell widths oscillates between one and two. In particular, for \( 2^k - 1 \leq p < 2^k \), where \( k \geq 1 \) is an integer, the ratio \( w/2r \) takes on two values \([M(1/2^k) = 2^{-k}, M(2/(2^k - 1)) = 2^{2-k}] \), while for all other values of \( p \) (i.e., \( 2^k \leq p < 2^{k+1} - 1 \), where \( k \geq 1 \) is an integer), \( w/2r \) takes on just one value \( M(1/2^k) = 2^{-k} \).

We now briefly describe a simple \( O(1) \) time object insertion procedure for the loose quadtree using the example of \( p = 1/4 \). From relation 4, we have that the quadtree cells containing a given input rectangle \( R \) with a minimum bounding hypercube box of radius \( r \) can be associated with any of three possible cells of radius \( M(r) \), \( 2M(r) \), and \( 4M(r) \). The insertion algorithm proceeds as follows. We first find a cell \( b \) of radius \( M(r) \), such that it contains the centroid of \( R \). This can be done in \( O(1) \) time by noting that \( M(r) = 2^{\lceil \log_2 r \rceil} \). At this point, we have that either \( b \), the parent of \( b \) (say \( b' \)) of radius \( 2M(r) \), or the parent of \( b' \) (say \( b'' \)) of radius \( 4M(r) \) contains \( R \) and we insert \( R \) in the smallest one whose expanded region contains \( R \).

The actual insertion algorithm is given by procedure \textsc{LooseQuadtreeInsert} below. It does not assume that the loose quadtree is represented as a tree structure with out degree 4 (8 for a loose octree in three dimensions). Instead, it assumes the use of a pointerless quadtree representation (e.g., [9, 27, 51]) that just keeps track of the leaf nodes (i.e., cells) of the loose quadtree which are represented using, for example, a number, termed a locational code, that uniquely identifies each leaf node. This number can be formed by concatenating the size of the cell, say \( i \) for a cell of width \( 2^j \), with a number \( j \) resulting from interleaving the binary representations of the coordinate values of a predefined corner such as the lower-left corner assuming that the origin of the underlying space is at the lower-left corner (e.g., \((a, b)\) in two dimensions) so that \( i \) is at the right of \( j \). The collection of these numbers can be represented using any access structure including binary search trees, balanced binary search trees, B-trees, etc. Thus the role of \textsc{LooseQuadtreeInsert} is simply to create records for the loose quadtree which consist of the locational code and a pointer to the object so that we can differentiate between objects that are associated with the same leaf node (i.e., cell) of the loose quadtree. In this case, the cell is replicated in the access structure.

1 procedure \textsc{LooseQuadtreeInsert}(p, o)
2 /* Given a loose quadtree with expansion factor \( p \), create and return a loose quadtree record for object \( o \) which contains the object and its locational code. Object \( o \) is represented by a record of type \textit{object} having the fields \textit{XCent}, \textit{YCent}, and \textit{MbbRadius} corresponding to the \( x \) and \( y \) coordinate values of \( o \)'s centroid, and the radius of \( o \)'s minimum bounding hypercube box. The function \( M(r) \) returns the integer \( 2^k \) such that \( 2^{k-1} \leq r < 2^k \). The locational code is obtained by applying bit interleaving to the binary representations of \( x_{\text{low}} \) and \( y_{\text{low}} \), the \( x \) and \( y \) coordinate values of the lower-left corner of the loose quadtree cell \( b \) of width \( w \) which contains \( o \) and concatenating it to the depth of \( b \) (i.e., \( \log_2(w) \)) and its value is a pointer to object \( o \). If several objects are associated with the same cell of the loose quadtree, then the cell is replicated. These replicated loose quadtree cells are differentiated by virtue of the objects that are associated with them. The actual loose quadtree record for the cell including its locational code is constructed by procedure \textsc{FormBlock} (not given here). Note the use of “\( \lceil \cdot \rceil \)” to denote integer division, “\( \div \)” to denote real division, and “\( \uparrow \)” to denote exponentiation. */
3 value real \( p \)
4 value object \( o \)
5 real \( r \)
6 integer \( i, w, x_{\text{low}}, y_{\text{low}} \)
7 \( r \leftarrow \textit{MbbRadius}(o) \)
8 for \( i \leftarrow \log_2(M(1/(p + 1))) \) step 1
9 \( \text{until } \log_2(M(2/p)) = 1 \)
10 /* Calculate width of smallest possible cell \( b \) containing \( o \) */
11 \( w \leftarrow (2 \uparrow (i + 1)) \div M(r) \)
12 /* Determine position \((x_{\text{low}}, y_{\text{low}})\) of \( b \)'s lower-left corner */
13 \( x_{\text{low}} \leftarrow (\textit{XCent}(o) \div w) \times w \)
14 \( y_{\text{low}} \leftarrow (\textit{YCent}(o) \div w) \times w \)
15 /* Determine if \( b \)'s expanded region contains \( o \) */
16 if \( x_{\text{low}} - p \times w \leq \textit{XCent}(o) - r \) and
17 \( \textit{XCent}(o) + r \leq x_{\text{low}} + (1 + p/2) \times w \) and
18 \( y_{\text{low}} - p \times w \leq \textit{YCent}(o) - r \) and
19 \( \textit{YCent}(o) + r \leq y_{\text{low}} + (1 + p/2) \times w \)
20 then exit_for_loop
21 endif
22 enddo
23 return(\textsc{FormBlock}(x_{\text{low}}, y_{\text{low}}, w, o))

The calculations for \( p = 1/4, p = 1/2, \) and \( p = 1 \) lead to the observation that as \( p \) takes larger values (even for \( p \) as small as \( 1/4 \)), the loose quadtree treats the input objects as if they are points and it is their centroid that determines their associated quadtree cell, while their size and the value
of the cell expansion factor determine the size of their associated quadtree cell. Actually, the above statement must be tempered a bit. In particular, although it implies that the position of object o is not a factor in the determination of the width w of the expanded quadtree cell c with which o’s minimum bounding hypercube box b is associated, this is not quite true as the existence of a range of values for the ratio \( w/2r \) of the widths of c and b is a direct result of the variation in the position of o along with that of the value of p. However, as we showed above, for values of \( p \geq 1/2 \), the values of the ratio of the widths of c and b take on at most two values which differ by one where, in the case of \( p \geq 1 \), the only reason for the two possible ratio values is the fact that at times \( p \) takes on a value which is one less than a power of 2.

At this point, it is appropriate to ask what value \( p \) should one use. The answer must bear in mind that as \( p \) gets large, the radii (i.e., half the width) of the associated expanded quadtree cells get larger and thus they overlap adjacent quadtree cells of half the radius for \( p = 1 \) and of equal radius for \( p = 2 \), and even greater radii as \( p \) increases further. On the other hand, as \( p \) approaches 0, the radii of the quadtree cells associated with object o are increasingly dependent on the position of the centroid of object o and can get disproportionately large independent of the radius of o’s minimum bounding hypercube box. The cardinality of the set of possible values of these radii is minimized at 1 when \( p \geq 2 \) with the exception of \( p = 2^k - 1 \) for integer values of \( k \) in which case the cardinality of the set is 2 corresponding to radius values \( 2^i \) and \( 2^{i+1} \) for some integer \( i \). Clearly, there is no point in letting \( p \) get larger than 2 in which case the radius of the associated quadtree cell is pre-determined and depends solely on the value of the radius of o’s minimum bounding box.

Thus we remain with the range \( 1/2 \leq p < 2 \) for which the cardinality of the set of possible values of the radii of the quadtree cells is 2 corresponding to radius values \( 2^i \) and \( 2^{i+1} \) for some integer \( i \). Our rationale for choosing \( p \) in this range is that the expanded quadtree cells are not so large as is the case for \( p = 2 \) and hence the extent of the overlap with adjacent quadtree cells is reduced, while the burden of having two possible radii for the quadtree cells is not great. Of course, procedure LOOSEQUADTREEINSERT is not as simple for \( p = 1 \) as it is for \( p = 2 \), in which case there is no need for the loop in lines \( 16 \)–\( 24 \). Nevertheless, for \( p = 1 \), the loop in lines \( 16 \)–\( 22 \) need only be executed twice, which is still quite simple. Ulrich \( 34 \) lets \( p = 1 \), while results of our experiments described in Section 5 make a case for choosing \( p \) to be infinitesimally smaller than 1. It is important to observe that all of the results that we have described hold for loose quadtrees of arbitrary dimension (e.g., three dimensions such as the loose octree) as they are all formulated in terms of the radii of the quadtree cells.

Algorithms that make use of the loose quadtree are simplified by our observation that the centroid of object o (actually of o’s minimum bounding hypercube box b of radius r) is always contained in the non-expanded portion of the quadtree cell c with which o is associated. However, there are scenarios where users may wish to violate this property. For example, for certain values of r and p, r may be sufficiently large so that both the centroid of o lies in the expanded portion of c and o still fits in the expanded cell c. This situation is desirable when users want to move o as much as possible without having to associate it with another quadtree cell just because o’s centroid is no longer in the non-expanded region of c. Interestingly, this modification does not change the ranges of relative cell widths as the example in Figure 4(c) still corresponds to the largest value of the ratio. The difference is that now the motion of the object so that the centroid of o is also in the expanded portion of c does not result in the association of o with another cell as long as o lies entirely in the expanded portion of c. Of course, this complicates subsequent searches (as well as delete operations), as now instead of just looking for a cell whose non-expanded portion contains the centroid of o, we must examine all possible cells whose expanded cells can contain o. Notice that in essence, we have transformed the search problem from one involving points (i.e., centroids of the objects) to one involving regions (i.e., the minimum bounding hypercube boxes of the objects).

Recall from Section 2 that for two-dimensional data, the partition fieldtree guarantees that the width w of the minimum enclosing quadtree cell c for the minimum bounding hypercube box b for object o is bounded by eight times the maximum width of \( b \). It is interesting to point out that the loose quadtree achieves the same guaranteed ratio when \( p = 1/4 \). This can be seen by observing that in this case the smallest possible minimum bounding hypercube box has a radius of \( pw/4 \) which has value \( w/16 \) for \( p = 1/4 \), and substituting into \( w/2r \) yields 8. Therefore, notwithstanding the arguments made in Section 2 about the complexity of using the partition fieldtree to perform operations such as point location thereby making it appear to be less practical than the loose quadtree, the partition fieldtree is superior to the cover fieldtree, in the sense of the variation in the sizes of the object’s associated quadtree cells, when \( p < 1/4 \).

5. EXPERIMENTAL EVALUATION

Experiments were run on a Linux (2.6.18) quad 1.86 GHz Xeon server with four gigabyte of RAM. The algorithms were implemented using GNU C++. These experiments studied the behavior of loose quadtrees in an environment where the objects are in motion. Our experimental setup consisted of a large collection of rectangle objects of arbitrary dimension \( d \) \((d < 16)\) stored in a loose quadtree spatial index. For most, but not all, of our experiments, we used random rectangle data obtained by generating their centroid and extents at random, which is equivalent to the method used by Ulrich \( 34 \). Each object (i.e., rectangle) in the index is associated with its minimum enclosing quadtree cell (actually minimum enclosing expanded quadtree cell), which is represented by its bit-interleaved Morton representation \( 25 \). The Morton representation is indexed using a B-tree index, which is referred to as a linear quadtree \( 15 \) \( 27 \). It is important to note that in our setup, we represent an object in the input by the quadtree cell with which it is associated (not necessarily containing it as the loose quadtree permits objects to be associated with smaller cells), which means that the index does not really store the exact geometry of the object. However, given that we know that the ratio of the size of the object’s minimum enclosing expanded quadtree cell and the object is bounded by a small value which is a function of \( p \), we are in some sense implicitly
recording the geometry of the object in the index. Moreover, the Morton representation that is stored in the B-tree contains a pointer to the actual object, which is stored in an array in order to facilitate quick updates, when necessary. In this respect, the loose quadtree is distinguished from all other spatial indices, such as an R-tree, that explicitly store the positions of objects (i.e., rectangles). This means that when the position of an object changes, in the case of an R-tree and related spatial indices, we would have to always update the indices as they depend on the minimum bounding boxes of the objects which have changed, while in the case of the loose quadtrees, we only need to update the index if if the object is associated with a different quadtree cell. This property of the loose quadtree makes it attractive for serving as a spatial index for moving object applications. In contrast, as we pointed out, updates in spatial indices such as the R-tree, as well as other related spatiotemporal indices, will often require a complete rebuild step which is quite complicated. For this reason, we do not compare the loose quadtree with these methods.

We ran a number of experiments to test the sensitivity of the loose quadtree to the motion of the objects that it stores. We used a collection of one million randomly generated rectangles in a d-dimensional space, which were stored in a B-tree based loose quadtree index, and varied d between 2 and 32. We let the expansion factor p vary between 0 and 5. Recall that for the case p = 0, the loose quadtree corresponds to an MX-CIF quadtree. We first built an index for all the objects in a loose quadtree for a given p. Next, the objects were translated in order to mimic a moving object application. If the translations resulted in an object being associated with a different quadtree cell, then we update the index, which involves deleting an entry from the B-tree index and adding a new entry corresponding to the minimum expanded quadtree cell containing the object after the translation. We tabulated the number of objects for which the index needed to be updated. We controlled the motion of the objects using a value s, denoting the maximum translation of the object across a single dimension. For example, suppose that s is 5%, then all of the rectangles are translated across each of the dimensions by a value that is at most 5% of its side length across any of the dimensions.

In order to provide a better understanding of the effect of motion on the loose quadtree index, we distinguish between two types of motion, namely uniform and fixed translations. In the case of a uniform translation, the motion is controlled by a random variable, which is bounded by s. In other words, all the objects are subjected to different translations, where the translation across any dimension is less than s. In the case of a fixed translation, all of the objects are translated by a fixed value (i.e., s) across each of the dimensions, which basically represents the worst case scenario (in terms of the maximum amount of motion) of any moving object application.

Finally, we also varied the sizes of the input rectangle objects by using the value δ, which denotes the ratio of the largest side length of a rectangle object in the dataset to the smallest side length of a rectangle object in the dataset. It should be clear that a large value of δ means a large range of rectangle object sizes, while a small value of δ means that the rectangle objects are more or less of the same size. All of the experiments whose results we present varied one or more of these variables to showcase the utility of loose quadtrees for moving object applications. Note that this is a far more extensive experimental evaluation than the one conducted by Ulrich [34] who only studied the behavior of the loose quadtree for culling operations and only compared the p = 0 and p = 1 cases.

Our first experiment considered the case of one million two-dimensional rectangle objects for values of p ranging between 0 and 5 and values of s ranging between 0.40% and 100%. The value of δ was kept constant at 10. Figure 5 shows the percentage of objects that required reinsertion as a function of p, while the different curves in the plot show the behavior of the loose quadtree index for different values of s. As expected, the percentage of objects that require reinsertion increases as s increases. From the figure, we observe that this percentage increases with p with a precipitous drop at p = 0.99 where the results are comparable to p = 0. Figure 6 provides a more vivid illustration of the comparability of the results for p = 0.99 with those for p = 0 by showing the result of normalizing the reinsertion rates for the different values of p and s vis-à-vis those for p = 0.99. Here we see that the reinsertion rate for p = 0.99 is superior to the other values of p for all reasonable values of s (i.e., less than s <50%). It is interesting to note that for all values of s, for small values of p, this percentage increases with p with a local maximum at around p = 0.5, at which time it has a precipitous drop at p = 0.99 where the results are comparable to p = 0, and then increases sharply for p = 1, and continues to increase, but at a lesser rate, as p continues to increase (i.e., p >1). This phenomenon is explained in greater detail below.

The percentage of objects requiring reinsertion is relatively low at p = 0 since the range of the values of the side lengths of the minimum enclosing quadtree cells is large as is also the value of the maximum side length. This means that objects often have a large area in which to move without requiring reinsertion. As p increases, we observe that the range of values of the side lengths of the minimum enclosing expanded quadtree cells become increasingly smaller, which means that the area in which the objects can move without requiring reinsertion gets smaller. Figure 7 illustrates this observation using an example object with minimum bounding hypercube box of radius r. In particular, we can see that for values of p in the range [0.25, 0.5], the side length of the minimum enclosing expanded quadtree cell is either 2, 4, or 8 times the radius of the minimum bounding hypercube box.
of the object (first row of Figure 7); while for values of $p$ in the ranges $[0,1)$, the side length of the minimum enclosing expanded quadtree cell is either 2 or 4 times the radius of the minimum bounding hypercube box of the object (second row of Figure 7); and for values of $p$ in the ranges $[1,2)$ the side length of the minimum enclosing expanded quadtree cell is either 1 or 2 times the radius of the minimum bounding hypercube box of the object (third row of Figure 7). Notice that the percentage of objects requiring reinsertion starts to decrease at $p = 0.5$ with a minimum at $p = 0.99$. This is because at $p = 0.5$ there are only two choices for the size of the minimum enclosing expanded quadtree cell, having eliminated the cell with side length 8 times the radius of the minimum bounding hypercube box of the object. Moreover, the number of objects associated with the eliminated cell are relatively small. On the other hand, at $p = 1$, there are again only two choices for the minimum enclosing expanded quadtree cell, but now a large such cell is replaced by one with a quarter of its area thereby greatly limiting the ability of the objects to move without requiring reinsertion. The pattern of increasing percentages requiring reinsertion continues unabated for $p > 1$ as we have increasingly smaller replacement cells with sawtooth like behavior in the neighborhood of $p = 2^k$. Note that had we used an R-tree, there would be no savings at all in the sense that each object would have to be reinserted in the R-tree thereby causing an entirely new R-tree to be built.

For some data structures the process of updates can be sped up by batching the updates using bulk loading methods (e.g., \[\text{[9]}\] \[\text{[10]}\] \[\text{[11]}\]). For example, Dittrich et al. \[\text{[9]}\] assume that updates can come at fast rates so it may not be good to handle one update at a time. Instead, they use “snapshots”, which are basically large static data structures and an update pool, which is a fast data structure stored in the main memory. In this case, updates are collected in the update pool and once enough of them have been accumulated they are applied en mass to the snapshot structure to create a new snapshot structure; hence a variant of bulk updates which are cheaper than single updates. In our method, all insertions and deletions are always performed in $O(1)$ time, and thus there is no need for pooling the updates.

The second set of experiments used the above environment and measured the number of updates in millions/second that can be supported by the loose quadtree data structure since this correlates with updating the spatial index with the new positions of the objects. Recall that an update in a loose quadtree involves deleting an entry from the B-tree index and inserting another entry. The key advantage of the loose quadtree structure is that small motions of objects, most likely, do not require any changes to the index, and if they do, then they are less complex as $p$ becomes increasingly larger than 0 as the range of possible minimum enclosing expanded quadtree cells is much smaller. Figure 8 shows the number of updates in millions/second that a loose quadtree index can support under both the uniform and fixed translation cases. We see that this number decreases as $s$ increases with performance generally peaking at $p = 0.99$, although this is most noticeable for $s \leq 50\%$. We also observe that the number of updates per second for a given value of $s$ does not vary greatly across different value of $p$ with the

---

**Figure 6:** Reinsertions for two-dimensional rectangle input for varying values of $p$ and $s$ with $\delta = 10$, normalized with the reinsertion rate for $p=0.99$ for a) uniform and b) fixed translations.

**Figure 7:** Illustration of the variation of the relative sizes of the minimum enclosing expanded quadtree cells and of the minimum bounding hypercube boxes of the objects with respect to different ranges of values of $p$: (first row) $0.25 \leq p < 0.5$, (second row) $0.5 \leq p < 1$, and (third row) $1 \leq p < 2$.

**Figure 8:** Update rates for two-dimensional rectangle objects for varying values of $p$ and $s$ with $\delta = 10$, for a) uniform and b) fixed translations.
exception of relatively small values of $s$ (0.40% and 1.0%), for which with respect to $p=0$. Figure 9 provides a more vivid illustration of this improvement by showing the result of normalizing the number of updates per second for the different values of $p$ and $s$ vis-a-vis those for $p=0$, where for a fixed translation $s=0.40\%$ we see a one order of magnitude improvement. Again, these results are primarily due to the dramatically reduced cost of insertion since the new minimum enclosing expanded quadtree cell can be determined in at most 2 lookup operations for $p=0.99$ versus a significantly higher number for $p=0$. This figure shows that the improved throughput is observed for most values of $p$ for $s \leq 50\%$.

![Figure 9: Update rates for two-dimensional rectangle objects for varying values of $p$ and $s$ with $\delta=10$, normalized with the reinsertion rate for $p=0$ for a) uniform and b) fixed translations.](image)

We now examine the effect of varying the dimensionality of the underlying data on the performance of the loose quadtree data structure. In particular, we repeated the experiments in Figure 4 for input datasets of varying dimensionalities 3, 4, 8, and 16 which are given in Figure 11. From this figure we see that the percentage of reinsertions for varying values of $p$ and $s$ still remain relatively low, although it does increase with dimensionality. Next, Figure 12 tabulates the update rates in millions/second on the loose quadtree data structure for 3 and 4 dimensions as these are the ones most likely to be used in gaming applications and spatio-temporal databases. Once again we can see that the peak performance of the loose quadtree data structure is at $p=0.99$. Note that the loose quadtree can handle 1–2 million updates per second. This result indicates that the loose quadtree is suitable for indexing moving objects in higher dimensions as well.

![Figure 12: Update rates in millions/second for a) 3, and b) 4 dimensional rectangle inputs for varying values of $p$ and $s$ with $\delta=10$, for uniform translations.](image)

The third set of experiments examined the effect of varying $\delta$ on the performance of the loose quadtree data structure. Recall that $\delta$ bounds the ratio of the length of the largest side of a rectangle object in the input dataset to the length of the smallest side of a rectangle object in the input dataset. In other words, if $\delta$ is small, then all the objects in the input are of the same size. If $\delta$ is large, then the objects are of different sizes. Figure 10 shows the update rates in millions/second for varying values of $\delta$ and $p$, keeping $s=5\%$ with uniform translations. We can see that both the reinsertion and update rates are relatively independent of $\delta$, which means that the loose quadtree data structure is suitable for handling datasets containing objects of varying sizes.

![Figure 10: a) Reinsertion and b) update rates in millions/second for two-dimensional rectangle input for varying values of $\delta$ and $p$ keeping $s=5\%$ with uniform translations.](image)

Finally, we tested the efficacy of loose quadtrees on a real world dataset. We used a 2D rectangle dataset generated by taking a TIGER road dataset of Los Angeles County, CA and then fitting a bounding box around the line segments forming the edges in the road network. The dataset contains 267k 2D rectangles with a good mix of rectangles of different sizes. We subjected the input rectangles to both uniform and fixed translations as in prior experiments described in Figures 9 and 11. As before, we report both the percentage of the objects that had to be reinserted due to the translation, and the time that it took to perform the reinsertion (i.e., to update the data structure) which we report in terms of the number of updates per second. Figure 13 shows the percentage of updates (i.e., reinsertions) needed for uniform and fixed translations, and once again it is easy to see that, for $p=0.99$, very few updates to the data structure are needed. Figure 13 and 13 shows the number of updates per second for uniform and fixed translations, and again it is not surprising that, for $p=0.99$, the number of updates that can be done per second sky rockets due to the savings in the reinsertion cost.

6. CONCLUDING REMARKS

We have shown how to determine for a loose quadtree the maximum possible width $w$ of the minimum enclosing expanded quadtree cell for an object $o$ with minimum bounding hypercube box $b$ of radius $r$ and cell expansion factor $p$. We have also shown that $w$ is independent of the position of $o$. This property enables determining the cell with which $o$
is associated and can be used, for example, in an algorithm to build a loose quadtree in an environment that deploys a pointerless quadtree. In particular, this independence means that the algorithm requires little or no search and could be used, for example, to populate a spatial database with the latest wave of multiprocessors such as those that make use of GPUs. (e.g., [16, 24, 33]). Our experiments have demonstrated that letting \( p \) take on a value infinitesimally smaller than 1 leads to the best results in minimizing the need to update the index when objects are moving thereby increasing the size of the collection (i.e., database) of objects that can be supported. This is in contrast to the conventional use of \( p = 1 \) [34].

It is important to note that like the competing solutions based on the multiple shifted quadtree and the partition fieldtree, the loose quadtree also has a disadvantage. In particular, in the case of the loose quadtree, the tighter bound and dimensionality independence come at the expense that the non-disjoint property of the loose quadtree means that an expanded cell in the loose quadtree intersects (i.e., overlaps) with several of its adjoining cells. On the other hand, the fact that the partition fieldtree shifts the different levels of a quadtree has the disadvantage that the property that if cells \( a \) and \( b \) in a quadtree are disjoint, then the descendants of \( a \) and the descendants of \( b \) are also disjoint no longer holds. Of course, the multiple shifted quadtree method has the drawback of the complexity of determining the object's associated quadtree and the fact that there are a multiple number of quadtrees for the set of objects.

7. REFERENCES


