PK-tree: A Dynamic Spatial Index Structure for Large Data Sets†

Wei Wang, Jiong Yang, and Richard Muntz

Department of Computer Science

University of California, Los Angeles

{weiwang,jyang,muntz}@cs.ucla.edu

November 17, 1997

Abstract

Large image and spatial databases are becoming more important in applications such as image archives and spatial data mining. In similarity searches, e.g., in image and multimedia systems providing content based search, high dimensional spaces are common. Thus, spatial indexing methods which can effectively deal with large data sets of high dimensionality are of great interest. In this paper we present a new index structure which is based on regular spatial decomposition but which uses a unique set of constraints to eliminate useless nodes that can result from a skewed spatial distribution of objects. The index structure can be formally defined and is unique for a given data set. It is efficiently update-able and bounds on the number of nodes and the height of the tree can be proved. Also bounds on the expected height of the tree can be given if certain mild constraints on the spatial distribution of objects can be given. Empirical evidence shows that the PK-tree outperforms the existing spatial indexes based on the R-tree such as the SR-tree and X-tree.

1 Introduction

Large image and spatial data bases are becoming more important in applications such as medical image archives and spatial data mining. Particularly in similarity search applications, e.g., in content based access in image databases, high dimensional spaces are common. Therefore, a variety of dynamic spatial indexing structures have been proposed in the past few years [Sam90] [Ber96] [Hen89] [Kat97].

The nature of spatial indexing indicates that spatial decomposition is a natural method to generate the spatial indexes. Any existing spatial indexing structure for large amounts of data involves some kind of spatial decomposition which recursively divides the space into subregions and employs a tree-like structure

†A companion paper emphasizing the formal proofs of properties of the PK-tree which has been submitted to PODS'98. A postscript copy of this manuscript is available from http://dml.cs.ucla.edu/~weiwang/paper/TR-97040.ps for anyone interested in the formal proofs.

for search in the space of interest. Most of these indexes can be divided into two classes based on the method used to subdivide space: the quad-tree family and the R-tree family. The quad-tree employs disjoint spatial decomposition techniques while the R-tree family uses minimum bounding boxes or spheres which may overlap.

We begin by examining the tradeoffs between the two approaches in general. The major advantage of the quad-tree approach is that there are no overlapping siblings since the space is recursively divided into disjoint subregions. During a point query, it is only necessary to traverse a single path from the root to a leaf. There are two popular methods for the disjoint spatial decomposition. (1) with regular spatial decomposition, the space is divided according to some predefined rules independent of the data stored in the index. An example is the PR quad-tree [Sam90]. This method can result in many empty or near empty nodes particularly when the spatial data has a skewed spatial distribution in which case the height of the tree can be very large. (2) with irregular spatial decomposition, the space is divided according to the actual data distribution, e.g., subdividing a region such that each subregion contains at least certain percentage of points. Two problems can arise from this kind of spatial division. Since the spatial division is made based on the data distribution, then all data in a region have to be considered during a spatial division of that region. Thus the data insertion and deletion can be costly such as in the BSP-tree [Abr95]. In order to eliminate the high cost of data insertion and deletion, a B-tree-like structure could be employed. However, many nodes in the index structure could be empty or near empty, such as in the K-D-B-tree [Rob81].

The structures in the R-tree family employ minimum bounding boxes or minimum bounding spheres to recursively divide the space. They are B-tree like structure so that each node has a bounded number of children and the height of the tree structure is bounded to $O(\log N)$ where N is the cardinality of the data set and updates can be easily made with adjustments to the size of the minimum bounding boxes or minimum bounding spheres in a limited part of the tree. The main potential problem with this kind index is overlap among siblings. During a point search, if the point is inside the overlapping area of several siblings, then the subtree corresponding to all the overlapping siblings have to be searched. Many heuristics have been developed to eliminate or reduce the overlaps, such as in the X-tree [Ber96] and R+-tree [Sel87]. Unfortunately, some important properties of the B-tree have to be sacrificed in the X-tree or R+-tree, e.g., the bound on the number of children in a node is violated in the X-tree while there exists several paths from root to a particular minimum bounding box in the R+-tree.

All the existing dynamic spatial indexing structures achieve some degree of success. However, there is still room for improvement. Many applications can benefit from the simple and easy tree traversal property of regular spatial decomposition. Therefore, we introduce a new spatial indexing structure based on regular spatial decompositions, the PK-tree. A PK-tree achieves a better bound on the height of the tree for skewed data distributions than other trees based on regular decomposition because it only instantiates non-leaf cells with at least a certain number of nonempty children. This restriction largely eliminates the problem of the height of the tree growing large due to skew in the spatial distribution of points. Updates to the

PK-tree are inexpensive and the performance of the PK-tree is shown via experiments to exceed that of any existing variations of the R-tree for the most common types of queries. Moreover, a PK-tree is independent of the order of data insertions and deletions. Last but not least, the PK-tree is based on a solid theoretical foundation. Its uniqueness for any data set, a bound on the mean height, and bounds on the number of children in a node can all be proved formally [Yan98].

The remainder of the paper is organized as follows. Related work is introduced in Section 2. We formally define the PK-tree structure in Section 3. Then in Section 4, we introduce the properties of the PK-tree. The PK-tree generation, update, and query algorithms are introduced in Sections 5 and 6. In Section 7, we discuss methods for dealing with high dimensional spaces and objects. We summarize the results of empirical studies comparing PK-tree performance with other spatial index methods in Section 8. Finally, we summarize and discuss conclusions in Section 9.

2 Related Work

As mentioned previously, a variety of dynamic spatial indexing structures have been proposed. We present a few representatives in this section. The Pyramid Structures represent the regular disjoint spatial decomposition approach while the K-D-B-tree is representative of the irregular disjoint spatial decomposition. Variations of the R-tree has been widely studied in the past decade, we present two of the most recent variations of the R-tree, the SR-tree and X-tree, which are compared empirically with the PK-tree in Section 8.

2.1 Pyramid Structures

Pyramid structures are a widely used technique in image processing [Sam90a]. The root of the pyramid corresponds to the entire image. Then the space corresponding to the root is recursively divided into quadrants down to the pixel level. For a $Z \times Z$ image, where $Z = 2^d$, the height of the pyramid is d and the total number of nodes in the pyramid is $4 \times (Z \times Z - 1)/3$.

Many image processing applications benefit from this structure for the following reasons:

1. fast direct access. The physical location of a node can be calculated since we know exactly the number of nodes in a level. Thus, we can directly reach the area of interest.

2. summarized information. A node usually contains the summarized information of the area to which it corresponds. In general, it can speed up the search time (query time).

Since pyramid structures were originally developed for image processing applications, it is not surprising that it is not directly suitable for dynamic spatial indexing structures. In an image, each pixel needs to be stored. However, in a large spatial area, much of the area can be empty while the minimum distance between two objects can be very small. If a pyramid structure is used, then the number of nodes at the bottom level can be very large. In turn, the entire storage of the pyramid structure can be huge and most of the cells are empty or near empty nodes.

2.2 K-D-B-Tree

The K-D-B-Tree [Rob81] is a height balanced tree index structure for multidimensional data. It is similar to the B+-tree. Unlike the pyramid structures, there is no "bottom level" in the K-D-B-tree. The distance between any two objects in the K-D-B-tree can be arbitrarily small. The root of the K-D-B-tree represents the entire space of interest. Then the space is recursively subdivided into subregions. Each node corresponds to a region, and the subregion corresponding to a child node is entirely contained in the region corresponding to the parent node. The area of the regions corresponding to sibling nodes can be different because the split of the parent region is to create the subregions each containing a similar number of objects. Since the regions corresponding to sibling nodes are disjoint, the search path of a point query is a single path from the root to a leaf. As a result, the point query search time is $O(\log N)$ where N is the data set size.

The K-D-B-tree forces a split of an intermediate node to form its children subregions. The problem with this forced split is that it can cause empty or near-empty nodes. Therefore, it degrades the storage utilization which in turn, impacts the performance of range queries and K-Nearest-Neighbor queries.

2.3 SR-Tree

The SR-tree [Kat97] is one of the newest variations of the R-tree [Gut84]. As mentioned previously, a major drawback of the R-tree can be the overlap among sibling nodes. The SR-tree uses both a minimum bounding box and a minimum bounding sphere at each node. The region that corresponds to a node in an SR-tree is the intersection of the minimum bounding box and the minimum bounding sphere associated with that node. Thus, the overlapping area between two sibling nodes is reduced. Consequently, the search time improves significantly. However, the storage required for the SR-tree is quite large because it has to store both the minimum bound boxes and minimum bound spheres. For the same reason, the structure creation time and update time are impacted.

2.4 X-Tree

The X-tree [Ber96] is based on the R^*-tree [Bec90]. The major difference between these two index structures is that when a node needs to be split, the R*-tree always splits the node according to some heuristics. However, in the X-tree, if no good split can be found, (e.g., all splits result in relatively large overlap) then the node remains unsplit and it is called a supernode. Therefore, the overlapping area is reduced. In low dimensional space, an X-tree has similar performance as a R^*-tree. In high dimensional space, since the number of supernodes increases, the X-tree outperforms the R*-tree significantly. However, the number of children in a supernode is not bounded and the search time on a supernode is at least linear with the number of children it has. Therefore, the search time can be impacted. In Section 8, we present empirical data comparing the performance of the SR-tree and the X-tree with the PK-tree.

3 Structure of PK-tree

We assume a two dimensional space for now, but results can be generalized to higher dimensions without difficulty. We will discuss higher dimensional data briefly in a later section. For simplicity of discussion in this paper, we also assume a bounding box C_0 for the set of points to be stored is specified a priori. However, this also is a requirement that is not difficult to eliminate[1]. C_0 is recursively divided into rectangular cells at successive levels. Level 0 contains C_0 and, for $i \geq 0$, level $i+1$ contains cells which are a regular subdivision of the cells at level i. The lengths of the sides of C_0 (at level 0) along the x and y axis are denoted by L_x and L_y, respectively. At level $i+1$ ($i \geq 0$), the lengths of the sides of a cell along the x and y coordinates are $\frac{1}{r_x}$ and $\frac{1}{r_y}$ of those of level i, where r_x and r_y are two (small) constant integers. $R = (r_x, r_y)$ is called the dividing ratio tuple. Thus a cell at level i is always divided into $r_x \times r_y$ cells at level $i+1$. The cell size at level i is $\frac{L_x \times L_y}{(r_x \times r_y)^i}$. This is illustrated in Figure 1.

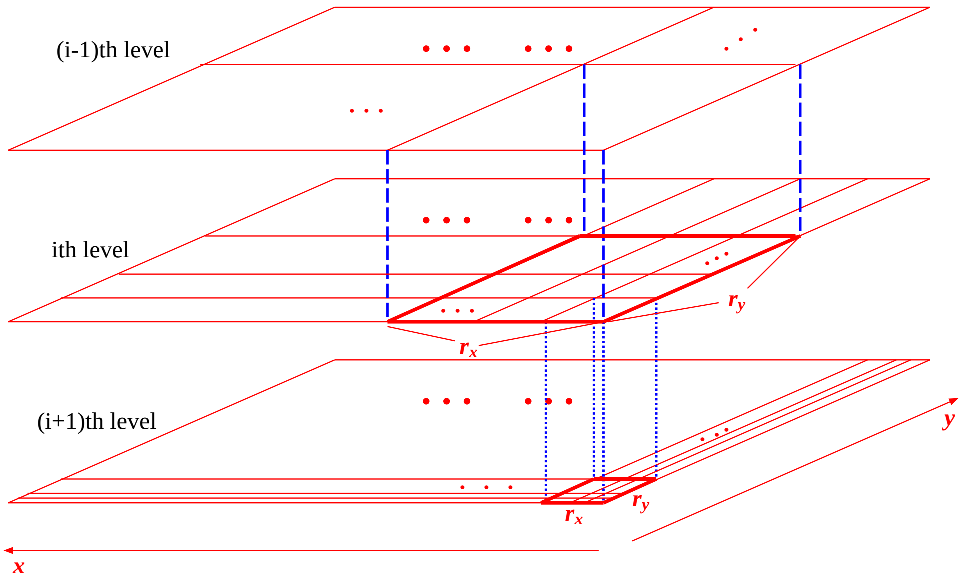


Figure 1: Space Dividing of part of PK-tree

Now, we introduce the following definitions.

For any data point d, we use (x_d, y_d) to denote its location. $C_0 = \{d | X_{init\_min} < x_d < X_{init\_max}, Y_{init\_min} < y_d < Y_{init\_max}\}$ where X_{init_min}, X_{init_max}, Y_{init_min}, and Y_{init_max} are the ranges of C_0. Let D be a finite

[1] One simply can predefine an expanding order of spatial area from the initial cell. Once a point is to be inserted which is located outside the current C_0, the current C_0 expands to the minimum area that contains the point in the predefined order. All properties presented in Section 4 still hold.

set of data points, $D \subset C_0$.

Definition 3.1 *A cell which contains only one point location is called a point cell.*

A point cell always has zero length sides and zero area. In the rest of this paper, a cell can be either a point cell or a cell resulting from the recursive spatial division of C_0.

Definition 3.2 *Let C be a cell and (x_{C1}, y_{C1}), (x_{C1}, y_{C2}), (x_{C2}, y_{C1}), and (x_{C2}, y_{C2}) be the location of C's vertices, where $x_{C1} \leq x_{C2}$ and $y_{C1} \leq y_{C2}$. For any data point $d \in D$, we say $d \in C$ iff both of the following conditions are satisfied:*

1. *$x_{C1} \leq x_d < x_{C2}$ for cells resulting from spatial division or $x_{C1} = x_d$ for point cells;*

2. *$y_{C1} \leq y_d < y_{C2}$ for cells resulting from spatial division or $y_{C1} = y_d$ for point cells.*

Otherwise, $d \notin C$.

We can view each cell as a set of points and use set notation to express the relationship among cells. For example, C_1 is a sub-cell of C_2 (or C_2 is a super-cell of C_1) if $C_1 \subseteq C_2$ and C_1 is a proper sub-cell of C_2 (or C_2 is a proper super-cell of C_1) if $C_1 \subset C_2$.

Definition 3.3 *Given a finite data set D contained in C_0 with a dividing ratio $R = (r_x, r_y)$, a cell C is K-instantiable $(K > 1)$ iff*

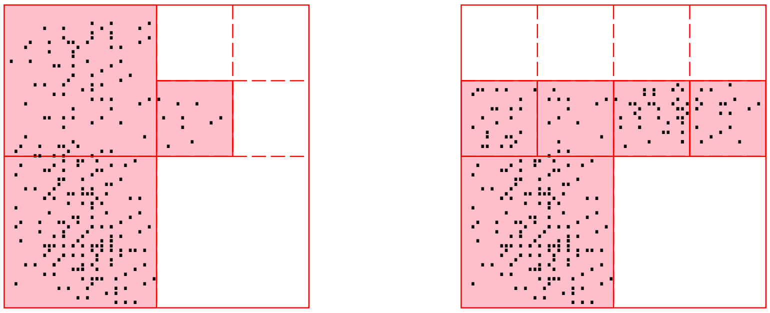
1. *C is a point cell containing $d \in D$, or*

2. *$\nexists\ k \leq K-1$ K-instantiable cells $C_1, \ldots, C_k \subset C$,*
   such that $C_i \subset C$, $1 \leq i \leq k$ and $\forall d \in D(d \in C \Longrightarrow d \in \{\bigcup_{i=1}^{k} C_i\})$.

The first condition states that point cells corresponding to the elements in D are K-instantiable. The second condition is a little bit more complicated. The idea is that for any cell, if all data points in it can be "covered" by less than K K-instantiable proper sub-cells, then this cell is not K-instantiable. For example, the cell in Figure 2(a) is not 4-instantiable since all data points can be covered by three 4-instantiable proper sub-cells, whereas the cell in Figure 2(b) is 4-instantiable. (The shaded area indicates 4-instantiable proper sub-cells.)

Definition 3.4 *Given a finite data set D contained in C_0 and a dividing ratio R, a PK-tree of rank K $(K > 1)$ is defined as follows.*

1. *Every K-instantiable cell is instantiated and becomes a node of the PK-tree; and all non-root nodes in the PK-tree are K-instantiable.*



(a) (b)

Figure 2: Examples of 4-instantiable Cell and not 4-instantiable Cell

2. *The cell at level* 0 *(corresponding to* C_0*) is always instantiated and serves as the root of the PK-tree.*

3. *For any two nodes* C_1 *and* C_2 *in a PK-tree,* C_1 *is a child of* C_2 *(or* C_2 *is the parent of* C_1*) iff*

   (a) C_1 *is a proper sub-cell of* C_2*, i.e.,* $C_1 \subset C_2$*, and*

   (b) $\nexists$ *node* C_3 *in PK-tree such that* $C_1 \subset C_3$ *and* $C_3 \subset C_2$*.*

The terms "nodes" and "cells" are used interchangeably in the remainder of this paper. In a PK-tree, a node C is called an *intermediate node* iff there exist two nodes C_1 and C_2 such that $C_1 \subset C$ and $C \subset C_2$. For each node C, the center location (x, y) and the level l are stored and are denoted as $C.x$, $C.y$, and $C.level$ respectively. These are used to define the spatial location of C. Moreover, the number of data points that are in C, denoted by $C.datanum$, will also be stored to facilitate query processing. All global parameters such as L_x, L_y, rank K, and the dividing ratio R are stored in the root node. We will use $Root_{PK}.L_x$, $Root_{PK}.L_y$, $Root_{PK}.L_y$, $Root_{PK}.K$, and $Root_{PK}.R$ respectively to denote these.

Note that sibling nodes in a PK-tree are not necessarily at the same level. For example, in a two dimensional space with dividing ratio $R = (2, 2)$, a node at level $i - 1$ in a PK-tree of rank 4 could have a child at level i and 4 children at level $i + 1$, as is illustrated in Figure 3. The shaded cells in the figure are 4-instantiated cells.

4 Properties

In this section, we will discuss several important properties of PK-trees. Following each property, we give a briefly intuitive explanation why it is true. The formal proofs of these properties are in [Yan98].

Property 4.1 *For any two cells* C_1 *and* C_2 $(C_1 \neq C_2)$*, their relationship must be one of the following.*

1. C_1 *and* C_2 *disjoint.*

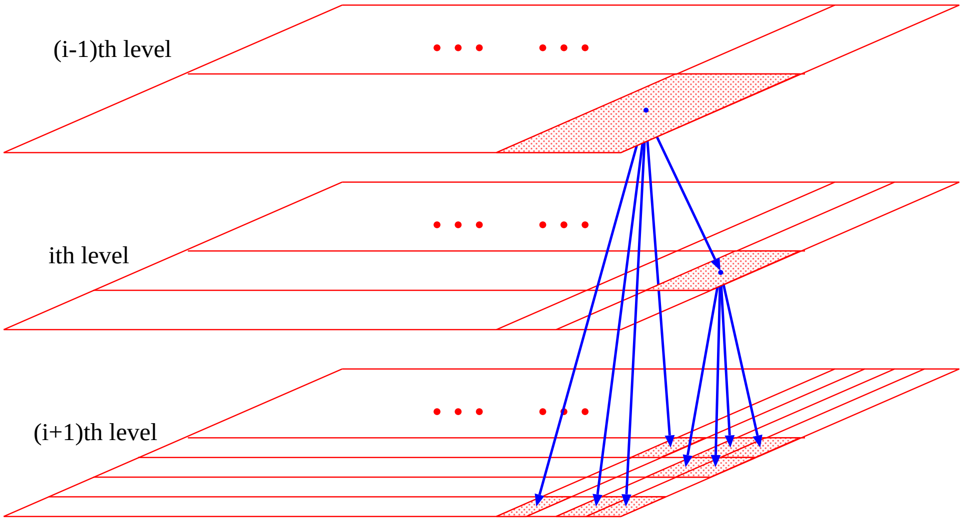


Figure 3: An Example of part of a PK-tree of rank 4

2. *Either* $C_1 \subset C_2$ *or* $C_2 \subset C_1$.

According to the recursive division of the space, cells at the same level are disjoint and cells at different levels are either disjoint or one is a proper subset of the other.

Property 4.2 *For any non-leaf node in a PK-tree, its children are mutually disjoint.*

By Definition 3.4, a cell and any of its subcells can not share a parent. Therefore, all children of a parent are mutually disjoint according to Property 4.1.

Property 4.3 *For a given two dimensional bounding box* C_0 *with dividing ratio* $R = (r_x, r_y)$, *each intermediate node in an PK-tree of rank* K $(K > 1)$ *has at least* K *and at most* $(K-1) \times r_x \times r_y$ *children.*

If a node C_T has more than $(K-1) \times r_x \times r_y$ children, then by the Pigeonhole Principle there must exist a proper subcell C, such that at least K children of C_T are the proper subcells of C. This violates Definition 3.4 since C is K-instantiable.

Property 4.3 gives us the range of the number of children of an intermediate node in a PK-tree. This range is one factor in choosing the dividing ratio R of the space and the rank K of the PK-tree to be built. For example, one reasonable goal might be to put all children of an intermediate node in one disk block to improve the retrieval time.

Property 4.4 *Any PK-tree of rank* K $(K > 1)$ *for a data set* D *of cardinality* N *in a two dimensional bounding box* C_0 *with dividing ratio* $R = (r_x, r_y)$ *has at least* $N + \frac{N-1}{(K-1) \times r_x \times r_y - 1}$ *and at most* $N + \frac{N+K-2}{K-1}$ *nodes.*

Denote by X the number of non-leaf nodes in a PK-tree. Any non-leaf node is a parent of some nodes and any non-root node is a child of some node. Moreover, according to Property 4.3, each parent node has at least K children and at most $(K-1) \times r_x \times r_y$ children. Therefore, we obtain the following inequality:

$$(X-1) \times K + 1 \leq X + N - 1 \leq X \times (K-1) \times r_x \times r_y$$

By solving this equation, we find the range of the total number of nodes $X + N$: $N + \frac{N-1}{(K-1)\times r_x \times r_y - 1} \leq X + N \leq N + \frac{N+K-2}{K-1}$. Thus, the property holds.

Property 4.5 *For a given data set D of cardinality N, where D is contained in the bounding box C_0 in a two dimensional space, there exists a unique PK-tree of rank K ($K > 1$) with dividing ratio $R = (r_x, r_y)$.*

Algorithm 5.1 in the next section demonstrates the existence of a PK-tree by construction. (Due to space limitations, we omit the formal proof of the correctness and termination of Algorithm 5.1.) A formal proof of the uniqueness of the PK-tree is presented in [Yan98]. Informally, the proof goes as follows. If for the same set of points there exist two different PK-trees for the same R, K, and spatial area, then there exists a node which appears in one tree, but not the other. However, by the definition of the PK-tree, this kind node can not exist since a node being K-instantiable is a function only of the set D and the pyramid of cells. Moreover, the parent/child relationship among all nodes are the same by definition, thus the uniqueness of PK-tree is guaranteed. A formal proof of the uniqueness of the PK-tree is presented in [Yan98]. In other words, this property states that a PK-tree is independent of the order of data insertions and deletions. Structures in the R-tree family do not have the uniqueness property. As a result, two entirely different R-trees can result for the same data set if the sequence of data insertions/deletions are ordered differently.

Statistical Properties:

Let $D = \{x_1, x_2, \ldots, x_N\}$ be a random set of N points in a space C_0. $P_D(x_1, x_2, \ldots, x_N)$ is the probability distribution for D. We construct the PK-tree T from D and denote the height of T by $h(T)$. For a given C_0 and R, let $\wp(C_0, K, R)$ be the set of cells in the corresponding pyramid structure. Let C_1, C_2, ..., C_i, ... be a *cell sequence* where $C_1 \supset C_2 \supset \ldots \supset C_i \supset \ldots$ in which C_i is a cell at level i of $\wp(C_0, K, R)$. Let $P_i = P(d \in C_i \mid d \in C_{i-1})$ be the probability that a random point $d \in D$ is in C_i given that d is in C_{i-1}.

Given a *cell sequence* $C_1, C_2, \ldots$, the corresponding *path sequence* ρ is $C_1, C_2, \ldots, C_{\pi-1}$ where π is the smallest integer such that $P_\pi = 0$ if there exists such a π. Otherwise, $\pi = \infty$ and the corresponding path sequence is the cell sequence. Given a path sequence $\rho = C_1, C_2, \ldots$, the corresponding *sub-path sequence* ρ' is a subsequence[2] of ρ such that each C_i is an element in ρ' iff $P_i < 1$. The motivation for defining the sub-path sequence is to eliminate from the path all cells that would not be K-instantiable; any cell for which $P_i = 1$ only contains points in one subcell at the next level and is not K-instantiable.

Let $f(N) = O(\log N)$. For a given sub-path sequence ρ', let ψ_f be the smallest real number such that $\mid \{P_j \mid P_j > \psi_f\} \mid \leq f(N)$, and let Ψ be the least upper bound of ψ_f over all sub-path sequences. Also, let

[2] Given a sequence, a subsequence contains a subset of elements from the original sequence in the same order, but not necessarily consecutive in the original sequence.

$\alpha_{\rho'} = \mid \{P_j \mid P_j > \Psi\} \mid$ for a given sub-path sequence ρ'. Define $\alpha = \max(\alpha_{\rho'})$ over all sub-path sequences. Clearly, $\alpha \leq f(N)$, which implies $\alpha = O(\log N)$ holds. For example, for a uniform spatial distribution of points, $\Psi = \frac{1}{r_x \times r_y}$.

Property 4.6 *Let D be a set of N points over a space C_0 where D is a random sample of such sets which has distribution $P_D(x_1, x_2, \ldots, x_N)$ with parameters Ψ and α for some $f(N) = O(\log N)$ and let T_D be the PK-tree for D with rank K. Let $h(T_D)$ be the height of T_D. Then $P(h > H) = O\left(\Psi^{H^2}\right)$ where $H = \Omega(\log N)$ and $H > c \times \alpha$ for some constant $c > 1$.*

According to this property, for any $g(N) = \omega(\log N)$, the probability that $h(T) \geq g(N)$ quickly goes to 0 as N gets large when $\Psi < 1$. The formal proof of this property can be found in [Yan98].

Property 4.7 *The expected height of the PK-tree for a set D of N data points with parameter Ψ is $O(\log N + N \times \Psi^{\log^2 N})$.*

Let h be the height of the PK-tree. By Property 4.6, $P(h > H) = O\left(\Psi^{H^2}\right)$ where $H = \Omega(\log N)$ and $H > c \times \alpha$. Since $\alpha = O(\log N) \leq c_0 \log N$ for some constant $c_0 > 1$, we choose $H = c_h \log N$, where $c_h > c \times c_0$ is a constant. Because the height of a PK-tree with N leaves is at most N, the expected height is

$$\begin{aligned}
E(h) &\leq (c_h \log N) \times (1 - P(h > c_h \log N)) + N \times P(h > c_h \log N) \\
&\leq c_h \log N \times \left(1 - O\left(\Psi^{c_h^2 \log^2 N}\right)\right) + N \times O\left(\Psi^{c_h^2 \log^2 N}\right) \\
&\leq c_h \log N + N \times O\left(\Psi^{c_h^2 \log^2 N}\right) \\
&= O(\log N + N \times \Psi^{\log^2 N}).
\end{aligned}$$

From this result a sufficient condition for the expected height of a PK-tree to be $O(\log N)$ is that Ψ be less than 1 and independent of N. When Ψ is independent of N, $N \times \Psi^{\log^2 N}$ approaches 0 for large N. If Ψ is dependent on N, then a sufficient condition for the expected height of a PK-tree to be $O(\log N)$ is $\Psi = O\left(\sqrt[\log^2 N]{\frac{\log N}{N}}\right)$.

Sufficient for $h(T) = O(\log N)$

The above results can be difficult to apply directly in a real situation. However, we are developing simpler, sufficient conditions to guarantee $\Psi < 1$ (and therefore, $E(h) = O(\log N)$) that should be more easily verified in practice. Several examples are described next.

(1) M-Level Clustering Spatial Distributions

We begin by describing 0-level clustering spatial distributions. A distribution of a set of points over C_0 is said to satisfy 0-level clustering if each point in D is an independent sample from the uniform distribution over C_0. As mentioned previously, $\Psi = \frac{1}{r_x \times r_y}$ in this case, thus the height of the PK-tree is $O(\log N)$ if the data set is from a 0-level clustering distribution.

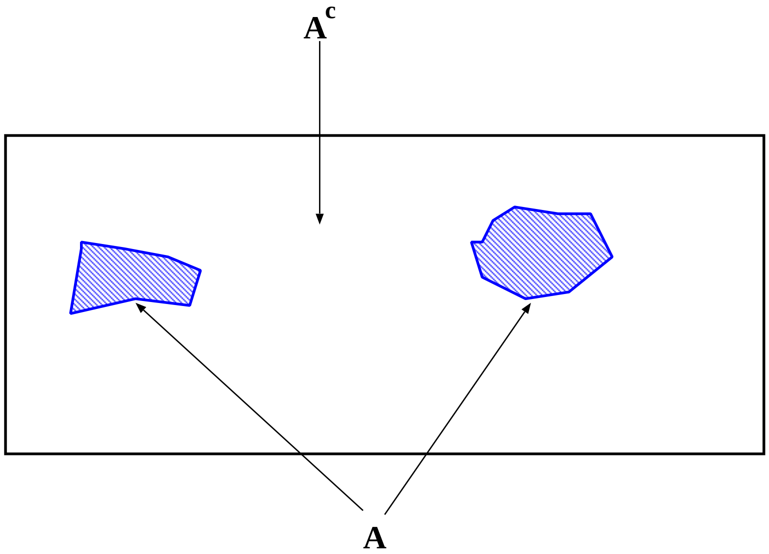


Figure 4: 1-Level Clustering

A distribution satisfying the 1-level clustering constraint is defined as follows. Let $A \subset C_0$ be some subset of C_0 and let $A^c = C_0 - A$. The distribution for points in A and A^c satisfies 0-level clustering constraint, i.e., the points in each region are uniformly and independently distributed over the region. Let D_A and D_{A^c} be the subsets of points in regions of A and A^c, respectively. The cardinality of these two subsets are denoted by N_A and N_{A^c}, respectively and $N = N_A + N_{A^c}$. Also denote the density in A and A^c by d_A and d_{A^c}, respectively. The 1-level clustering constraint requires that $\frac{Area(A)}{Area(A^c)}$ be independent of N. An 1-level clustering spatial distribution is illustrated in Figure 4. The conditional probability $P(d \in C_{i+1} \mid d \in C_i)$ for $C_{i+1} \subset C_i$ is equal to the expected number of points in C_{i+1} divided by the expected number of points in C_i. Without loss of generality, assume $d_{A^c} < d_A$ and $R = (r_x, r_y)$. So $\Psi \leq \frac{d_A}{d_A + (r_x \times r_y - 1) \times d_{A^c}}$ which is independent of N and less than 1 when $d_{A^c} > 0$. Therefore, for $d_{A^c} > 0$, the height of the PK-tree is $O(\log N)$. For $d_{A^c} = 0$, we do not have such an easy bound on Ψ. Instead we resort to the device of adding N points uniformly distributed over A^c to the original data set D. Let D' be the new data set. The corresponding $\Psi' \leq \frac{d_A}{d_A + (r_x \times r_y - 1) \times d'_{A^c}}$ where d'_{A^c} is the new density in A^c and $\frac{d'_{A^c}}{d_A} = \frac{Area(A)}{Area(A^c)}$ which is independent of N. Thus the expected height of the PK-tree for D' is again $O(\log N)$. Since the height of a PK-tree never decreases when we insert points, the height of the PK-tree for D is at most the height of the PK-tree for D'. Therefore, the expected height of the PK-tree for a data set that satisfies the 1-level clustering constraint is $O(\log N)$.

The definition of M-level clustering distribution is a direct extension of the definition of 1-level clustering. So the expected height of a PK-tree for a data set which has M-level clusters is $O(\log N)$ if M is bounded by a constant.

(2) Bounded Intensity Constraint

Intensity function, $\lambda(B)$, where B is any subset of C_0, is defined as the probability that a point is in B [Sto94]. Data set D consists of N independent samples and $\lambda(B)$ is independent of N for all B. Let I_{glb} and I_{lub} be the greatest lower bound and the least upper bound of the intensity for all B. The bounded intensity constraint requires that $I_{glb} > 0$. In this case, $\Psi \leq \frac{I_{glb}}{I_{lub}+(r_x \times r_y - 1) \times I_{glb}} < 1$ is independent of N. Therefore, the PK-tree for any data set that satisfies the bounded intensity constraint has an expected height equal to $O(\log N)$.

The following bound on the height of the tree applies in the case that points are never closer than some specified distance. The bound on the tree height is not based on the special properties of the PK-tree and apply to all pyramid structures. However, we state the result for completeness.

Minimum Distance Constraint

If the lengths of the side of the root spatial area are L_x and L_y respectively and the minimal distance between any two data points is $Dist$, then the height of the tree in the worst case is $O\left(\min(h_x, h_y)\right)$ where $h_x = \log_{r_x}\left(\frac{L_x}{dist}\right)$, $h_y = \log_{r_y}\left(\frac{L_y}{dist}\right)$, and (r_x, r_y) is the dividing ratio no matter how the locations of the data points are distributed.

For example, consider houses in California as the data set of interest. If we represent each house by its center point, then we might reasonably assume that the minimal distance between any two such points cannot be less than 1 meter. In turn, the height of the tree is less than 15 (no matter how the locations of these houses are distributed) if we use $R = (2, 2)$ as the dividing ratio.

5 Generation and Update

5.1 Generation

An easy way to generate a PK-tree is to begin from an empty PK-tree (a PK-tree that only has the root node) and insert the data points one by one. The algorithm is presented in Algorithm 5.1.

Algorithm 5.1 *A Simple Algorithm to Build PK-tree*

```
BUILDPKTREE(D, K)
{
   instantiate the root Root_PK
   for each data point d in D
      DATAINSERT(Root_PK, d)
   return Root_PK
}
```

The DATAINSERT procedure is described in Section 5.2. The complexity of this creation algorithm is N times the complexity of the DATAINSERT procedure. Later, we will show that the complexity of DATAINSERT is $O(\log N)$ in the average case. Therefore, the complexity of this PK-tree creation algorithm is $O(N \log N)$ on average. However, this is not the only way to generate PK tree. We present another more efficient algorithm in [Yan98]. The computational complexity of that algorithm is $O(N \log N)$ in the worst case. In addition, if the input data is ordered, the computational complexity is $O(N)$.

5.2 Update

In this section, we discuss two kinds of update: insertion and deletion. For a given PK-tree T of rank K, $Root_{PK}$ denotes the root of T. Let d denote the data point that needs to be inserted into T or deleted from T and let C_d be the point cell that will contain (or contains) d.

5.2.1 Insertion

The insertion of a data point d into the PK-tree T is achieved by inserting the corresponding point cell C_d into T. This can be done by first following the path from the root $Root_{PK}$ down to the leaf level to locate all (potential) ancestors of C_d and, second, following the path from the leaf level back to the root along the same path to make all necessary changes (instantiate or deinstantiate cells) to enforce the constraints required of a PK-tree. The algorithm is presented in Algorithm 5.2.

Algorithm 5.2 *Data Point Insertion*

DATAINSERT($Root_{PK}, d$)
/* *insert data point d in the PK-tree rooted at* $Root_{PK}$ */
{
    instantiate the corresponding point cell C_d ($d \in C_d$)
    INSERT($Root_{PK}, Root_{PK}, C_d$)
}
INSERT($Root_{PK}, C_T, C_d$)
/* *insert point cell* C_d *into the subtree rooted at* C_T */
{
    if $\exists$ *a child* C *of* C_T *such that* $C_d \subset C$
        then *INSERT*($Root_{PK}, C, C_d$)
        else
            make C_d *be a child of* C_T
            CHECKINSTANTIABLE($Root_{PK}, C_T$)
}

$CHECKINSTANTIABLE(Root_{PK}, C_T)$
/* *check instantiability of* C_T *and all its ancestors* */
{
  $ChildrenSet \leftarrow$ *set of all children of* C_T
  if $\mid ChildrenSet \mid < Root_{PK}.K$ *and* $C_T \neq Root_{PK}$
    then
      $P \leftarrow$ *parent of* C_T
      make all nodes in $ChildrenSet$ *be children of* P
      deinstantiate C_T
      $CHECKINSTANTIABLE(Root_{PK}, P)$
    else if $(\exists C' \subset C_T$ *and* $C'_1, \ldots, C'_{Root_{PK}.K} \in ChildrenSet$
        such that $C'_1 \subset C', \ldots, C'_{Root_{PK}.K} \subset C')$ *and*
        $(\nexists C'' \subset C'$ *and* $C''_1, \ldots, C''_{Root_{PK}.K} \in ChildrenSet$
        such that $C''_1 \subset C'', \ldots, C''_{Root_{PK}.K} \subset C'')$
      then
        instantiate C'
        make C' *child of* C_T
        for *each node* C_i *in* $ChildrenSet$
          if $C_i \subset C'$
            then *make* C_i *be a child of* C'
        $CHECKINSTANTIABLE(Root_{PK}, C_T)$
}

The computational complexity of data point insertion is linearly proportional to the height of the PK-tree. (Since the number of children of each node is bounded, the complexity of the algorithm at each level is constant.) Since the average height of the PK-tree is $O(\log N)$, the average computational complexity is $O(\log N)$ and the worst case complexity is $O(N)$ where N is the number of data points in the PK-tree.

5.3 Deletion

The deletion of a data point d from the PK-tree T is achieved by deleting the corresponding point cell C_d from T. Similar to insertion, this can also be done in two parts: the first part is to follow the path from the root, $Root_{PK}$, down to the leaf level to locate all ancestors of C_d and remove C_d while the second part works from the leaf level back to the root along the same path to make all necessary changes (instantiate or deinstantiate cells) to maintain the tree. The algorithm is presented in Algorithm 5.3.

Algorithm 5.3 *Data Point Deletion*

$DATADELETE(Root_{PK}, d)$

/* *delete data point* d *from the PK-tree rooted at* $Root_{PK}$ */

{

$DELETE(Root_{PK}, Root_{PK}, d)$

}

$DELETE(Root_{PK}, C_T, d)$

/* *delete data point* d *from the subtree rooted at* C_W */

{

if $\exists$ *a child* C *of* C_T *such that* $d \in C$

then if C *is a leaf cell*

then

deinstantiate C

$CHECKINSTANTIABLE(Root_{PK}, C_T)$

else

$DELETE(Root_{PK}, C, d)$

}

The computational complexity of data point deletion is the same as data point insertion, which is $O(\log N)$ on average and $O(N)$ in the worst case.

6 Query

In this section, we restrict our attention to point objects in a 2-dimensional space and study three common spatial queries. Queries concerning objects with higher dimensions are discussed in the following section.

- *Range queries.* Given a point, find all other points within a certain distance from the given point.
- *K nearest neighbor queries.* Given a point, find the K closest points.
- *Rectangle enclosure queries.* Given a rectangular box, find all points enclosed by it.

The query algorithms of the PK-tree are similar to those of other trees. In the interest of conserving space, here we only informally describe the algorithms for answering these three types of queries. The formal description of the algorithms are in the appendix.

6.1 Range Queries

Given a location (x, y) and $range \geq 0$, $RANGE(Root_{PK}, x, y, range)$ will return the set of data points within $range$ distance from (x, y). The idea is to begin from the root of the PK-tree and recursively traverse down all the nodes that have nonzero intersection with the circle centered at (x, y) with radius $range$ until we reach the leaf level. If an intermediate node C' is entirely enclosed by the circle, then we will call another function RETRIEVE-CELL(C', $Result$) to add every data point in C' to the result set without further checking.

When $range$ is small, the number of data points returned can be regarded as constant. Then, the computational complexity is $O(h)$ where h is the height of the PK-tree. However, if $range$ is large and the number of data points returned is linearly proportional to N, then the computational complexity becomes $O(N)$ where N is the cardinality of the data set upon which the PK-tree is built. Single point retrieval is a special case of range queries where $range = 0$.

6.2 K Nearest Neighbor Queries

Given a location (x, y) and a constant k, K-N-N($Root_{PK}, x, y, k$) returns the k nearest neighbors in the PK-tree. The variable *bound* is used to store the range within which there are at least k data points. We utilize two sets $Candidates$ and $Result$. $Candidates$ contains the set of (non-leaf) cells that need be examined further. They are actually those non-leaf cells that overlap with the circle centered at (x, y) with radius *bound*. The cells in *Result* are those cells enclosed by this circle. At the beginning, *bound* is set to ∞. All children of the root $Root_{PK}$ are added to *Result* and all nonleaf children are added to $Candidates$. After this, $UPDATE(k, Result, Candidates, bound)$ is invoked to set *bound*. Each time, a cell C in $Candidates$ is chosen to be examined, it is removed from $Candidates$ and *Result*. There are many ways to choose a cell from the $Candidates$ set. Here we always choose the closest cell to (x, y) according to Euclidean distance. The $Candidates$ set is maintained by using a heap to facilitate this process. All children of C that are contained within the circle centered at (x, y) with radius *bound* are added to *Result* and all non-leaf children overlapping with the circle are added to $Candidates$. Then $UPDATE(k, Result, Candidates, bound)$ shrinks *bound* to be the smallest distance such that there are at least k data points in the cells that are contained within the circle centered at (x, y) with radius *bound*, and removes all "unqualified" elements according to the new *bound* from $Candidates$ and *Result*. This procedure continues until $Candidates$ is empty. At this point, the elements in *Result* are the k nearest neighbors of (x, y) and hence are returned. The complete algorithm is presented in [?]. If we regard k as a constant, then the complexity is $O(h)$ on average where h is the height of the PK-tree. Due to limitations on space, we omit the formal proof of correctness.

6.3 Rectangle Enclosure Queries

The rectangle enclosure query is another common query type. For a given rectangular box $[l_x, u_x] \times [l_y, u_y]$, RECTANGLE-ENCLOSURE($Root_{PK}, l_x, u_x, l_y, u_y$) returns all points within this rectangle box.

7 High Dimensionality

In this section, we denote by d_o and d_s the number of dimensions of objects and space, respectively. Objects are represented by d_o dimensional intervals which are bounding boxes for irregular shaped objects. (Note that the bounding boxes are aligned with the axis of the coordinate system.) To store a set of d_o dimensional boxes, we use the *transformation technique* [See88][Hen89] which stores these boxes as $d_o + d_s$ dimensional points. We choose the simple corner representation which considers for each of the d_o dimensions the lower and upper bounds for the boxes to be distinct dimensions. The remaining dimensions in the space do not change. For example, consider three 1-dimensional objects in a 2-dimensional space. As shown in Figure 5 (a), the coordinates of the three line segment objects are a: (0.9, 0.2) to (1.1, 0.2), b: (1.5, 1) to (2.1, 1), and c: (2.2, 0.5) to (2.3, 0.5). Each object varies in the first dimension but not in the second. We map the 2 endpoints of the object in Dimension 1 to two distinct dimensions: $1'$ and $1''$. Since, the objects do not vary on the second dimension, it remains unmodified. The resulting 3-dimensional points are (0.9, 1,1, 0.2), (1.5, 2.1, 1), and (2.2, 2.3, 0.5) as shown in Figure 5 (b).

Storage and search for non-point objects The basic mechanisms of PK-tree creation and updating in high dimensions are similar to those in 2-dimensions. In this section, we discuss the techniques for *box-enclosure* and *box-intersection* queries.

- *box-enclosure* queries. Given a d_s dimensional box, find all objects which are enclosed by it. This kind query is similar to the *rectangle-enclosure* query in 2-dimensional space.
- *box-intersection* queries. Given a d_s dimensional box, find all objects which intersect with it.

Assume that dimensions $1, 2, 3, \ldots, d_s$ are the dimensions of the space in which objects are located, among which dimensions $1, 2, 3, \ldots, d_o$ are the dimensions of the objects where $d_s \geq d_o$. A box in a query is of the form $[l_1, u_1] \times [l_2, u_2] \times \ldots \times [l_{d_s}, u_{d_s}]$ where l_i and u_i are the lower and upper bounds of the box in dimension i, respectively. The mapping from the original d_s dimensional space to the new $d_s + d_o$ dimensions is done as follows: any point p in the original d_s dimensional space, $p = (p_1, p_2, p_3, \ldots, p_{d_o}, p_{d_o+1}, \ldots, p_{d_s})$ is mapped to a point p' in $d_s + d_o$ dimensional space where $p' = (p_1, p_1, p_2, p_2, p_3, p_3, \ldots, p_{d_o}, p_{d_o}, p_{d_o+1}, p_{d_o+2}, \ldots, p_{d_s})$.

We begin by explaining the box-enclosure problem. We first translate the d_s dimensional box to a $d_s + d_o$ dimensional box as $[l_1, u_1] \times [l_1, u_1] \times [l_2, u_2] \times [l_2, u_2] \times \ldots \times [l_{d_s}, u_{d_s}] \times [l_{d_s}, u_{d_s}] \times [l_{d_o+1}, u_{d_o+1}] \times [l_{d_o+2}, u_{d_o+2}] \times \ldots [l_{d_s}, u_{d_s}]$. All the points enclosed by this box in the $d_o + d_s$ dimensional space are the d_o dimensional objects enclosed by the original box in the d_s dimensional space.

As for the box-intersection queries, for each dimension $d \in \{1, 2, \ldots, d_o\}$, we define

$$\theta[l_d, u_d] = [l_d - E_d, u_d] \times [l_d, u_d + E_d]$$

Here E_d is the maximum extension of objects in dimension d. Then the original box is translated to $\theta[l_o, u_o] \times \theta[l_1, u_1] \times \ldots \times \theta[l_{d_o}, u_{d_o}] \times [l_{d_o+1}, u_{d_o+1}] \times \ldots \times [l_{d_s}, u_{d_s}]$. The points enclosed by this $d_o + d_s$ dimensional box are all objects that have nonzero intersection with the original box in d_s dimensional space. If E_d is very large for some objects, then we can divide the large objects into small objects as in the R+-tree [Sel87] so that the E_d will be smaller for the divided objects.

For example, assume the query for the previous example is to find all objects within or intersecting rectangle $[0.8, 0] \times [1.9, 2]$. The spatial relationship between the objects and the rectangle in the query is shown in Figure 5 (a). The transformed boxes are the dotted line cube and solid line cube in Figure 5 (b), respectively. The result is that object a is inside the dotted line cube, which indicates that it is enclosed by the query two dimension rectangle; object b is inside the solid line cube, but outside the dotted line cube, which indicates that b intersects the query rectangle, but not enclosed by it; object c is outside both cubes, which indicates c is outside the query rectangle.

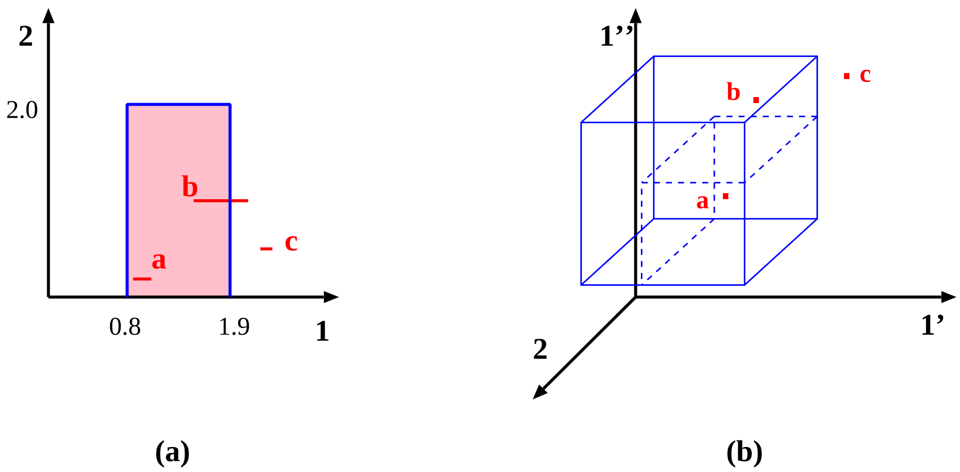


Figure 5: High Dimension Queries

Alternative subdivision of space

When we have a very large number of dimensions, there could be a problem with the dividing ratio $R = (r_1, r_2, \ldots, r_{d_o+d_s})$ because for a cell in level i, there are $r_1 \times r_2 \times \ldots \times r_{d_o+d_s}$ subcells in level $i+1$. If all $r_1, r_2, \ldots, r_{d_o}$ are integers larger than 1, then $r_1 \times r_2 \times \ldots \times r_{d_o+d_s}$ can be very large. In turn, a node in a PK-tree can be very large. To address this problem, we can vary the dividing ratio from level to level. For example, we only divide dimension 1 and 2 in odd levels, and dimension 3, 4, and 5 in even levels. As long as we keep the dividing ratio information in the PK-tree, query and update algorithms are simple to revise to accommodate this variation. A similar technique is also used in the k-d tree [Ben75].

8 Performance Comparison of PK-tree, SR-tree, and X-tree

All the performance tests in this section were done on a SPARC-10 workstation with 196 MB main memory and several GB of disk space. For the K-Nearest-Neighbor query and Range query, we choose to use no cache for both PK-tree and SR-tree. X-tree has two settings: with cache and without cache. The numbers in Figure A.2 and Figure 8 give the size of cache used for the X-tree with the cache option on. We use the version of X-tree source code given to us by the authors of [Ber96] recently, and the SR-tree source code from the authors of [Kat97].

We choose the rank of the PK-tree to be 4 for all tests in this section. We also choose the dividing ratio to be 2×2. For greater than 2 dimensions, we divide along two dimensions at each level and sequence in a round robin fashion through the dimensions. We found that each indexing structure tested in this section spends approximately the same percentage of time doing I/O for the same test data/query. Therefore, we only plot the total elapsed time in the figures.

8.1 Structure Generation

For high dimensions and a large number of data points, it is crucial that the data structure can be created efficiently. We examine the performance of the structure generation time of the SR-tree, X-tree, and PK-tree. In particular, we study the creation time as a function of the number of points and as a function of the number of dimensions.

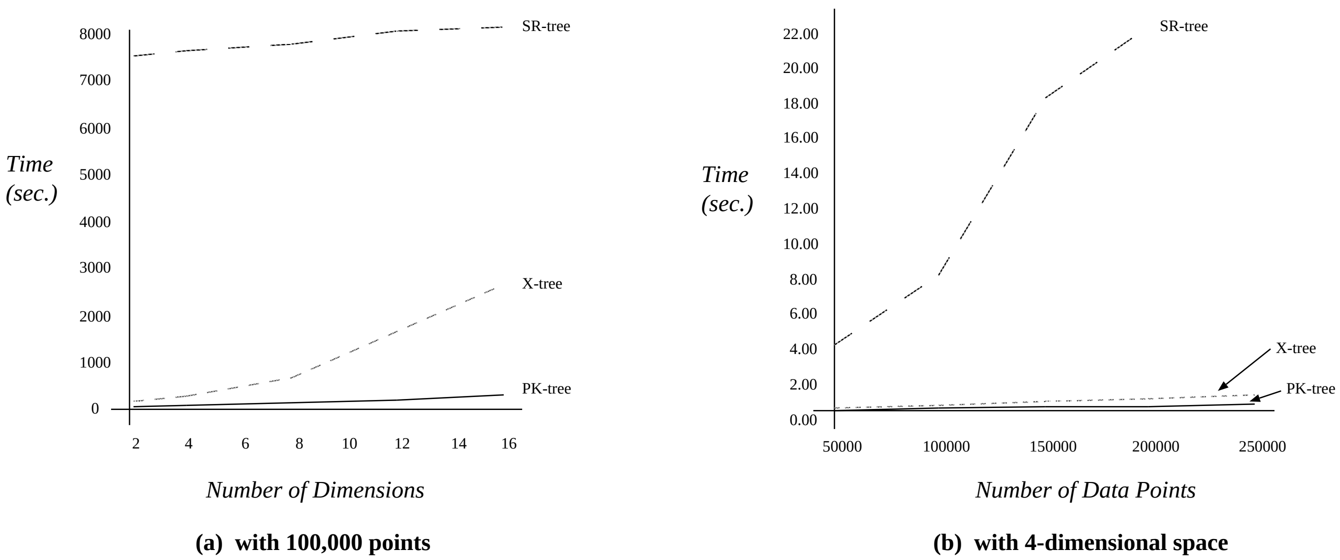


Figure 6: Structure Generation Time

Figure 6(a) shows the index creation time for one hundred thousand uniformly distributed points as a function of the number of dimensions. It is clear that the SR-tree has a much larger generation time. This is due to the fact that it requires computation of not only the minimum bounding boxes, but also the minimum bounding spheres as well. The storage requirement of the SR-tree is about 10 times larger than that of the X-tree or the PK-tree. This also contributes to the longer generation time.

The X-tree has a relatively short generation time in lower dimensions. As the number of dimensions increases, the generation time increases very fast. According to [Ber96], the X-tree does not have many supernodes in low dimensions so in this case it is very similar to a R^*-tree. However, the number of the supernodes becomes larger with an increase of the number of the dimensions. In this case, the time for generating supernodes is significant. As a result, the X-tree generation time increases sharply with the number of dimensions.

The PK-tree, on the other hand, has a comparatively short generation time and the time increases very slowly with the number of dimensions. Since the spatial division follows a simple rule, the computation cost of splitting a node is low. As a result, the generation time is small for all dimensions in Figure 6(a).

Figure 6(b) shows the index generation time for 4 dimensional uniformly distributed points as the number of points increases. In this figure, the PK-tree takes about one third the time of X-tree generation, and both the PK-tree and the X-tree have much lower generation time than SR-tree for the reasons explained previously.

In the current implementation, all three trees use point-by-point insertion. Therefore, based on the average performance of the tree generation, we can project that PK-tree takes much less time for insertions than the SR-tree and the X-tree.

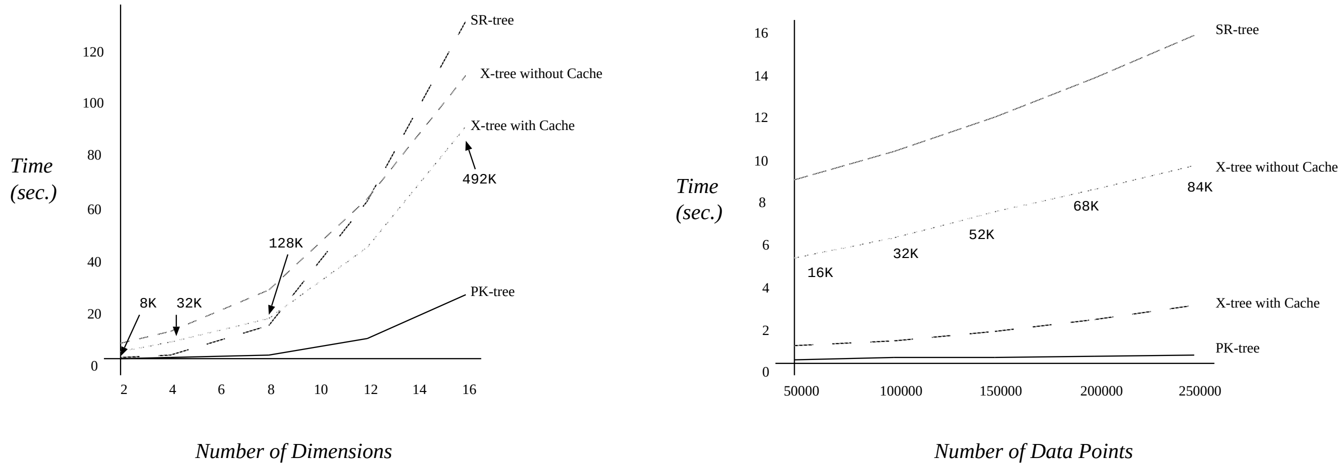
[Ber97] describes a batch loading algorithm that can be applied to most indexing structures to improve the structure generation time. However, at this time, we are not sure what the effect of the batch loading algorithm would be on the three indexing structures.

8.2 K-Nearest-Neighbor Query

K-Nearest-Neighbor (K-N-N) query is a very common query used in many applications. We examine the performance of this type of query in this subsection. We choose $K = 10$ for the K-N-N query in this section.

Figure A.2(a) shows the performance of K-N-N queries using each of the three trees with one hundred thousand uniformly distributed points. In low dimensional spaces, the SR-tree has lower response time than the X-tree because X-tree has performance similar to the R^* tree while SR-tree employs both the minimum bounding boxes and the minimum bounding spheres simultaneously. As a result, the overlapping between siblings is reduced and the response time of the K-N-N query improves in lower dimensionality. However, in a high dimensional space, the X-tree outperforms the SR-tree due to the effect of supernodes.

The PK-tree outperforms both the SR-tree and X-tree because the PK-tree uses regular spatial division. There are no overlapping siblings in the PK-tree. In addition, the average height of the PK-tree is $O(\log N)$. Therefore, the query time for the PK-tree is lower in both the low and high dimensional space.

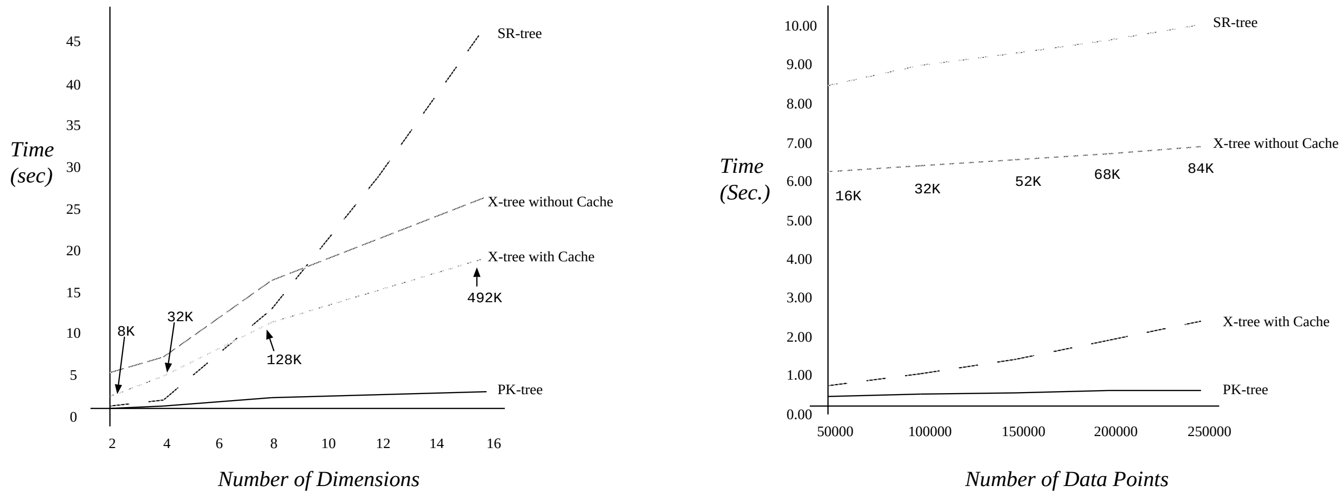


(a) with 100,000 points **(b) with 4 dimensional space**

Figure 7: K-Nearest-Neighbor Query Time

Figure A.2(b) shows the average performance of K-N-N query in a 4 dimensional space as a function of the data set size. It is clear that the PK-tree outperforms the other two tree structures by a large margin.

8.3 Range Query



(a) with 100,000 points **(b) with 4-dimensional space**

Figure 8: Range Query Time

Figure 8(a) shows the average performance of range queries with one hundred thousand uniformly distributed points. The size of the query region is chosen so that 6 to 10 points will be returned on average. The response time of the SR-tree increases fastest among the three indexing structures. The response time of the X-tree increases at a slower pace than the SR-tree while the PK-tree has the slowest rate of increase with the dimensionality. Once again, this is due to overlapping of minimum bounding rectangles or spheres occurring more frequently in the high dimensions for the SR-tree while the X-tree has more and more supernodes. Although supernodes reduce the number of overlapping minimum bounding rectangles, each supernode can be very large and thus can impact the performance of the range queries. However, there is no overlap in a PK-tree. And we also find that the height of a PK-tree does not increase with the dimensions of the space. Therefore, the only increase of the response time of a PK-tree is the computation associated with additional dimensions.

Figure 8(b) shows the average performance of range queries with 4-dimensional uniformly distributed points. It is easy to see that the response time of the PK-tree increases very slowly due to the fact that the average height of the tree grows only as a logarithm function of the size of the data set. With an increase in the number of points, (but no increase in the size of the root, i.e., density increases) the overlap among nodes in the SR-tree and the X-tree increases. And as a result, the response time of the SR-tree and the X-tree increases at a much faster pace than with the PK-tree. The response time of the X-tree increases at a slower pace than the SR-tree due to the effect of supernodes.

8.4 Skewed Data Distribution

In the previous performance comparisons, we used uniformly distributed data points. However, in many real cases, the data distribution is skewed. Spatial indexes based on regular spatial division (such as the PK-tree) are thought to not work well in the case of skewed data distribution. However, the PK-tree eliminates all non-K-instantiated nodes that eliminates the performance impact of most skewed spatial distributions. For example, if most data points are concentrated in one sub-region of the root space, then there are only a few nodes between the root and the node(s) representing the region of data point concentration. As mentioned before, the average height of the tree is $O(\log N)$ for all but the most perverse types of skewed data distribution. In most cases, the PK-tree performs as well for skewed data distributions as for the uniform data distribution.

We chose a data set from the SEQUOIA 2000 benchmark [Sto93] as our test case for a skewed data distribution. The data set consists of 62557 2-dimensional points giving the locations of landmarks in California. Table 1 shows the average performance of indexing structure creation, K-N-N query, and range query on the California landmark data set.

It is clear that the performance of the PK-tree does not vary much from that of the uniformly distributed data points in this case and that the PK-tree outperforms the X-tree and SR-tree by a large margin.

	Generation Time	K-N-N Query Time	Range Query Time
X-tree without cache	120 sec	1.9 sec	0.92 sec
X-tree with 4 KB cache	120 sec	1.6 sec	0.67 sec
SR-tree	3651 sec	0.12 sec	0.045 sec
PK-tree	62 sec	0.041 sec	0.017 sec

Table 1: Performance of SEQUOIA 2000 Benchmark

9 Conclusion

In this paper, we introduced a novel spatial indexing structure: PK-tree. It significantly improves the creation time (insertion time) and query time compared to existing spatial indexing structures. It employs regular spatial decomposition but eliminates the empty or near empty intermediate nodes. The PK-tree performs well for uniformly distributed data and for most skewed data distributions. In addition, we can formally analyze the performance of the PK-tree [Yan98]. Therefore, we believe that the PK-tree has significant advantages over existing spatial indexing structures.

References

[Abr95] M. Abrash. BSP trees. *Dr. Dobbs sourcebook*, 20(14), 49-53, May/June 1995.

[Bec90] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. The R*-tree: an efficient and robust access method for points and rectangles. *Proc. ACM SIGMOD Conf. on Management of Data*, 322-331, 1990.

[Ben75] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, vol. 18, No. 9, pp. 509-517, 1975.

[Ber96] Stefan Berchtold, Daniel A. Keim, and Hans-Peter Kriegel. The X-tree : an index structure for high-dimensional data. *Proc. 22nd Intl. Conf. on Very Large Data Bases (VLDB)*, 28-39, 1996.

[Ber97] Jochen Van den Bercken, Bernhard Seeger, Peter Widmayer. A Generic Approach to Bulk Loading Multidimensional Index Structures. *Proc. 23rd Intl. Conf. on Very Large Data Bases (VLDB)*, 406-415, 1997.

[Cia97] Paolo Ciaccia, Marco Patella, and Pavel Zezula. M-tree: an efficient access method for similarity search in metric spaces. *Proc. 23rd Intl. Conf. on Very Large Data Bases (VLDB)*, 426-435, 1997.

[Gut84] Antonin Guttman. R-trees: a dynamic index structure for spatial searching. *Proc. ACM SIGMOD Conf. on Management of Data*, 47-57, 1984.

[Hen89] Andreas Henrich, Hanas-Werner Six, and Peter Widmayer. The LSD tree: spatial access to multi-dimensional point and non-point objects. *Proc. 15th Intl. Conf. on Very Large Data Bases (VLDB)*, 45-54, 1989.

[Kat97] Norio Katayama and Shin'ichi Satoh. The SR-tree: an index structure for high-dimensional nearest neighbor queries. *Proc. ACM SIGMOD Conf. on Management of Data*, 369-380, 1997.

[Lin94] King-Ip Lin, H. V. Jagadish, and Christos Faloutsos. The TV-tree: an index structure for high-dimensional data. *VLDB Journal*, 3(4):517-542, 1994.

[Pag93] B. U. Pagel, H. W. Six, H. Toben, and P. Widmayer. Towards an analysis of range query performance in spatial data structures. *9th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, 214-221, 1993.

[Rob81] J. T. Robinson. The K-D-B-tree: a search structure for large multidimensional dynamic indexes. *Proc. ACM SIGMOD Conf. on Management of Data*, 10-18, 1981.

[Sam90] H. Samet. *The Design and Analysis of Spatial Data Structures.* Addison-Wesley, 1990.

[Sam90a] H. Samet. Efficient processing of window queries in the pyramid data structure. *9th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, 265-272, 1993.

[See88] Bernhard Seeger and Hans-Peter Kriegel. Techniques for design and implementation of efficient spatial access methods. *Proc. 14th Intl. Conf. on Very Large Data Bases (VLDB)*, 360-371, 1988.

[Sel87] Timos K. Sellis, Nick Roussopoulos, and Christos Faloutsos. The R+-tree: a dynamic index for multi-dimensional objects. *Proc. 13th Intl. Conf. on Very Large Data Bases (VLDB)*, 507-518, 1987.

[Sel97] Timos K. Sellis, Nick Roussopoulos, and Christos Faloutsos. Multidimensional access methods: trees have grown everywhere. *Proc. 23rd Intl. Conf. on Very Large Data Bases (VLDB)*, 13-14, 1997.

[Sto93] M. Stonebraker, J. Frew, K. Gardels, and J. Meredith. The SEQUOIA 2000 storage benchmark. *Proc. 1993 ACM SIGMOD Int. Conf. Management of Data*, pp. 2-11, 1993.

[Sto94] D. Stoyan and H. Stoyan. *Fractals, Random Shapes and Point Fields.* John Wiley & Sons, 1994.

[Wan97] Wei Wang, Jiong Yang, and Richard Muntz. STING: a statistical information grid approach to spatial data mining. *Proc. 23rd Intl. Conf. on Very Large Data Bases (VLDB)*, 186-195, 1997.

[Whi96] D. A. White and R. Jain. Similarity indexing with the SS-tree. *Proc. 12th intl. Conf. on Data Engineering (ICDE)*, 516-523, 1996.

[Yan98] Jiong Yang, Wei Wang, and Richard Muntz. Yet another spatial index structure. Submitted to *17th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, 1998. http://dml.cs.ucla.edu/~weiwang/paper/TR-97040.ps

A Query Algorithms

Algorithm A.1 *Range Queries*

$RANGE(Root_{PK}, x, y, range)$

{

  $Result \leftarrow \emptyset$

  $RANGE\text{-}SUBTREE(Root_{PK}, x, y, range, Result, Root_{PK}.r_x, Root_{PK}.r_y, Rot_{PK}.L_x, Root_{PK}.L_y)$

  return $Result$

}

$RANGE\text{-}SUBTREE(C, x, y, range, Result, r_x, r_y, L_x, L_y)$

{

  for *each child* C' *of* C

    if $ENCLOSE(C', x, y, range, r_x, r_y, L_x, L_y)$

    then $RETRIEVE\text{-}CELL(C', Result)$

    else if $OVERLAP(C', x, y, range, r_x, r_y, L_x, L_y)$

      then $RANGE\text{-}SUBTREE(C', x, y, range, Result, r_x, r_y, L_x, L_y)$

}

$RETRIEVE\text{-}CELL(C, Result)$

{

  if C *is a point cell*

  then $Result \leftarrow Result \cup \{C.data\}$

  else for *each child* C' *of* C

    $RETRIEVE\text{-}CELL(C', Result)$

}

Algorithm A.2 *K Nearest Neighbor Queries*

$K\text{-}N\text{-}N(Root_{PK}, x, y, k)$

{

  $Candidates \leftarrow \emptyset$

  $Result \leftarrow \emptyset$

  $bound \leftarrow \infty$

for *each child* C *of* $Root_{PK}$

  $GET\text{-}DISTANCE(C, d_{min}, d_{max}, x, y, Root_{PK}.r_x, Root_{PK}.r_y, Root_{PK}.L_x, Root_{PK}.L_y)$

  if C *is not a point cell*

  then

    $Candidates \leftarrow Candidates \cup \{(C, d_{min}, d_{max})\}$

  $Result \leftarrow Result \cup \{(C, d_{min}, d_{max})\}$

$UPDATE(k, Result, Candidates, bound)$

while $Candidates \neq \emptyset$

  $C \leftarrow GET\text{-}NEAREST(Candidates)$

  for *each child* C' *of* C

    $GET\text{-}DISTANCE(C', d'_{min}, d'_{max}, x, y, Root_{PK}.r_x, Root_{PK}.r_y, Root_{PK}.L_x, Root_{PK}.L_y)$

    if $d'_{min} \leq bound$ *and* C' *is not a point cell*

    then

      $Candidates \leftarrow Candidates \cup \{(C', d'_{min}, d'_{max})\}$

    if $d'_{max} \leq bound$

    then $Result \leftarrow Result \cup \{(C', d'_{min}, d'_{max})\}$

  $UPDATE(k, Result, Candidates, bound)$

return $Result$

}

$GET\text{-}DISTANCE(C, d_{min}, d_{max}, x, y, r_x, r_y, L_x, L_y)$

{

if C *is a point cell*

then

  $d_{min} \leftarrow \sqrt{(C.x - x)^2 + (C.y - y)^2}$

  $d_{max} \leftarrow d_{min}$

else

  $sidelength_x \leftarrow L_x / r_x^{C.level}$

  $sidelength_y \leftarrow L_y / r_y^{C.level}$

  $dis_{x_{min}} \leftarrow \min\{\mid C.x - x \mid - sidelength_x/2, 0\}$

  $dis_{y_{min}} \leftarrow \min\{\mid C.y - y \mid - sidelength_y/2, 0\}$

  $dis_{x_{max}} \leftarrow \mid C.x - x \mid + sidelength_x/2$

  $dis_{y_{max}} \leftarrow \mid C.y - y \mid + sidelength_y/2$

  $d_{min} \leftarrow \sqrt{dis^2_{x_{min}} + dis^2_{y_{min}}}$

  $d_{max} \leftarrow \sqrt{dis^2_{x_{max}} + dis^2_{y_{max}}}$

}

UPDATE$(k, Result, Candidates, bound)$

{

  $datanum \leftarrow 0$

  sort elements (C', d'_{min}, d'_{max}) *of Result in ascending order of* d'_{max} *into a list*

    $(C^1, d^1_{min}, d^1_{max}), \ldots, (C^j, d^j_{min}, d^j_{max})$

  $i \leftarrow 0$

  while $datanum < k$ *and* $i \leq j$

    $i \leftarrow i + 1$

    $datanum \leftarrow datanum + C^i.datanum$

  $bound \leftarrow d^i_{max}$

  while $i < j$

    $i \leftarrow i + 1$

    if $d^i_{max} > bound$

    then $Result \leftarrow Result - \{(C^i, d^i_{min}, d^i_{max})\}$

    for *each element* (C, d_{min}, d_{max}) *in Candidates*

      if $d^i_{min} > bound$

      then $Candidates \leftarrow Candidates - \{(C^i, d^i_{min}, d^i_{max})\}$

}

Algorithm A.3 *Rectangle Enclosure Queries*

RECTANGLE-ENCLOSURE$(Root_{PK}, l_x, u_x, l_y, u_y)$

{

  $Result \leftarrow \emptyset$

  RECTANGLE-SUBTREE$(Root_{PK}, l_x, u_x, l_y, u_y, Result, Root_{PK}.r_x, Root_{PK}.r_y, Root_{PK}.L_x, Root_{PK}.L_y)$

  return *Result*

}

RECTANGLE-SUBTREE$(C, l_x, u_x, l_y, u_y, Result, r_x, r_y, L_x, L_y)$

{

  for *each child* C' *of* C

    if *INSIDE*$(C', l_x, u_x, l_y, u_y, r_x, r_y, L_x, L_y)$

    then *RETRIEVE-CELL*$(C', Result)$

    else if *INTERSECT*$(C', l_x, u_x, l_y, u_y, r_x, r_y, L_x, L_y)$

then $RANGE\text{-}SUBTREE(C', l_x, u_x, l_y, u_y, Result, r_x, r_y, L_x, L_y)$

}

$INSIDE(C, l_x, u_x, l_y, u_y, r_x, r_y, L_x, L_y)$

{

if C *is a point cell*

then if $C.x \geq u_x$ *and* $C.x \leq l_x$ *and*

$C.y \geq u_y$ *and* $C.y \leq l_y$

then return *true*

else return *false*

else

$sidelength_x \leftarrow L_x / r_x^{C.level}$

$sidelength_y \leftarrow L_y / r_y^{C.level}$

if $C.x + sidelength_x/2 \leq u_x$ *and* $C.x - sidelength_x/2 \geq l_x$ *and*

$C.y + sidelength_y/2 \leq u_y$ *and* $C.y - sidelength_y/2 \geq l_y$

then return *true*

else return *false*

}

$INTERSECT(C, l_x, u_x, l_y, u_y, r_x, r_y, L_x, L_y)$

{

if C *is a point cell*

then if $C.x \geq u_x$ *and* $C.x \leq l_x$ *and*

$C.y \geq u_y$ *and* $C.y \leq l_y$

then return *true*

else return *false*

else

$sidelength_x \leftarrow L_x / r_x^{C.level}$

$sidelength_y \leftarrow L_y / r_y^{C.level}$

if $\mid C.x - \frac{l_x+u_x}{2} \mid \leq sidelength_x + \frac{u_x-l_x}{2}$ *and*

$\mid C.y - \frac{l_y+u_y}{2} \mid \leq sidelength_y + \frac{u_y-l_y}{2}$

then return *true*

else return *false*

}