Yet Another Spatial Indexing Structure*

Jiong Yang, Wei Wang, and Richard Muntz
Department of Computer Science
University of California, Los Angeles
{jyang,weiwang,muntz}@cs.ucla.edu

November 17, 1997

Abstract

Large data sets of high dimensional objects have become important in applications such as image databases. We present a new hierarchical spatial indexing structure in this paper called the PK-tree, which provides efficient queries, low space overhead and efficient updates. There is a more declarative nature to the definition of the PK-tree which results in a structure that is unique for a given set of objects to be stored. This in turn enables analysis that is simpler than if the structure were dependent on the order in which points were inserted, deleted, etc. Results for this structure include a bound on the number of children of each node in a PK-tree, a generation algorithm with a worst case complexity of $O(N \log N)$ where N is the number of points, and general sufficient conditions for the average height of the PK-tree to be $O(\log N)$.

1 Introduction

Today, many applications such as image data bases and data mining involve not only large collections of objects but also objects with large dimensionality. In addition, in many applications objects can be dynamically updated, inserted, and deleted. How to organize/index such data sets becomes a challenging problem. By the nature of data intensive applications, a spatial indexing structure is needed to achieve a degree of success in each of the following:

1. Efficient storage. The overhead of the structure should not be too large compared to the data itself.
2. Efficient query (or object retrieval).
3. Efficient update. The structure has to allow object changes/updates, e.g., deletions and insertions. This also implies that the object update cost should not be too large or require frequent/expensive reorganization.

The relative frequencies of the operations are application dependent. Therefore, we study the performance of each operation separately so that for each application, the effect on overall performance of a particular structure can be estimated.

A spatial indexing structure that can achieve optimal success for all three metrics is difficult to find (if not impossible). For example, BSP trees [Abr95] are generated in a manner that exploits the distribution of object locations. It is very efficient in storage and retrieval. However, it can not support efficient object updates. Research on spatial indexing has been intensive in the past two or three decades and during this period, many spatial data indexing structures have been proposed, including [Kat97] [Bec90] [Ber96] [Hen89] [Sam90]. All these indexing structures achieve some degree of success in some areas, but more improvements are possible and desirable. This related work is discussed in more detail in the following section.

We introduce a new indexing structure, the PK-tree, in this paper. PK-trees employ uniform spatial subdivision. Unlike other regular decomposition approaches that perform poorly with non-uniform distribution of objects over the index space, PK-trees use a novel combination of regular spatial decomposition combined with rules for eliminating redundant tree nodes to achieve excellent performance relative to the best previous index schemes. PK-trees have the following attractive properties: uniqueness of the structure for a given set of objects, a bound on the maximum

*A companion paper emphasizes the applications of the PK-tree and empirical comparisons of the performance of the PK-tree and several of the leading contenders that have been published in the past several years. This paper has been submitted to SIGMOD'98. A postscript copy of this manuscript is available from http://dml.cs.ucla.edu/~weiwang/paper/TR-97039.ps for anyone interested in practical applications of the PK-tree.

and minimum number of children a node can have, no overlap among sibling nodes, inexpensive updates, efficient storage, and a bound on the average height of the tree of $O(\log N)$ where N is the size of the data set. In this paper, we only discuss point objects in a 2-dimensional space. However, PK-tree concepts can be easily extended to multi-dimensional objects in a multi-dimensional space. Due to space limitations, we will not address higher dimensional spaces here, but these are studied in [Wan97].

The remainder of the paper is organized as follows. Related work is introduced in Section 2 and the PK-tree structure is formally defined in Section 3. Then we present formal proofs of some important properties of PK-trees in Section 4. In section 5, we outline an efficient algorithm for structure generation. Insertion, deletion, and point search algorithms are presented in Section 6. Finally, we offer conclusions in Section 7.

2 Related Work

Most spatial index structures can be classified as being from the Quad-tree family or the R-tree family. In this section, we discuss the advantages and disadvantages of these two index structure families.

2.1 Quad-tree

The PR quad-tree (Point-Region Quad-tree) [Sam90] employs recursive regular spatial decomposition. Each non-leaf node in a PR quad-tree has four children, corresponding to the four quadrants. Each leaf node contains one data point (or object). During an insertion, if the inserted point is located very close to an existing point, then the leaf node which contains the existing point will be recursively decomposed until the existing point and the inserted point are contained in different leaf nodes. One of the major drawbacks of the PR quad-tree is that the height of the tree is dependent on the minimum Euclidean distance separating two data points rather than the total number of data points. This can lead to inefficient storage and search performance.

There are other relevant variations of the quad-tree concept, such as the k-d tree, the adaptive k-d tree, the MX quad-tree, etc. [Sam90]. The average tree height of some of these variations is $O(\log N)$, e.g., the adaptive k-d tree and the b-d tree [Ohs83]. The adaptive k-d tree and the b-d tree employ a data dependent top-down spatial division approach. During data insertions, if the number of data points in a node exceeds a threshold, the node will be split according to some rules, e.g., each resulting node contains at most two thirds of the total points in the original node [Ohs83]. This kind restriction is expensive to maintain. If points inserted after a split causes an unbalance, then a reorganization is required. This can result in an expensive reconstruction of an entire subtree. Therefore, the update operation can be expensive in the k-d tree and the b-d tree.

2.2 R-Tree Family

Since the 1980s, many researchers have been interested in the R-tree [Gut84]. Many variations such as R^*-tree [Bec90], X-tree [Ber96], and SR-tree [Kat97] have been proposed. The fundamental idea across this family of indexing structures is the same: a minimum bounding box (MBB) is associated with each non-leaf node which bounds the region in which all objects lie which are stored in (or indexed by) the sub-tree rooted in the node. The size of the minimum bounding box depends on the distribution and extent of objects. The average height in the family of R-trees is $O(\log N)$ where N is the number of data points because the R-tree uses a B+-tree like structure.

The main problem posed by R-trees is overlapping of nodes. If the area of interest in a search includes the overlap among several sibling nodes, then the subtrees rooted in those nodes have to be searched. This can be a source of less than optimal performance and leads to difficulty in predicting performance. Several heuristic schemes have been proposed to address this problem. R^*-trees try to minimize the overlap area among nodes and also employ a technique to periodically delete a percentage of objects and more optimally reinsert them. This is an expensive operation if it is done often. X-trees try to reduce the node overlap problem by not splitting the node if all possible splits will result in large overlap. This introduces another problem: there is no bound on how many children a node can have. [Ber96] acknowledges the problem that the supernodes in an X-tree can be quite large; especially in high dimensional spaces. Thus a node can grow very large and a search of this node will take time proportional to the number of children which can be expensive. The SR-tree tries to use both a minimum bounding box and a minimum bounding sphere at the same time. The authors claim this can reduce the overlap between siblings. However, the insertion and deletion operations can be more expensive and the storage requirement is larger. Moreover, all these schemes are entirely heuristic. There is no meaningful formal bound on the amount of overlap between sibling nodes (or the number of children of an X-tree node).

3 Definition of PK-tree

3.1 Space dividing

We deal explicitly with two dimensional space in this paper and assume that the space of interest is a predefined rectangular area. The development can be easily extended to the case that the spatial area of interest is dynamically defined as data points are added, but due to space limitations, we will not address this issue in this paper. In order to describe a PK-tree, consider that the spatial area is recursively divided into successively smaller rectangular cells. One cell that represents the entire area of interest is at level 0 and is denoted by C_0. (If the spatial area of interest is not a rectangular shape, then C_0 is the minimum bounding rectangle of the area of interest.) The following spatial dividing technique is similar to that used in constructing the pyramid structure for a region C_0. We divide the cell at level 0 into several disjoint subcells at level 1, these cells at level 1 are each divided into disjoint subcells at level 2, and so on. For the spatial area C_0, we thus have different sets of cells at different levels with different cell sizes. The higher the level is, the smaller the cell size is. Cells at the same level are of the same size. The lengths of the sides of a cell at level i along the two dimensions, say X and Y, are denoted by L_x^i and L_y^i, respectively, and $\forall i > 0$, the lengths of the sides of a cell along the x and y axis are $\frac{1}{r_x}$ and $\frac{1}{r_y}$ respectively, of those of level $i-1$, where r_x and r_y are two constant integers. We call $R = (r_x, r_y)$ the dividing ratio tuple. Then each cell at level i is always divided into $r_x \times r_y$ cells at level $i+1$. The cells at two successive levels are illustrated in Figure 1.

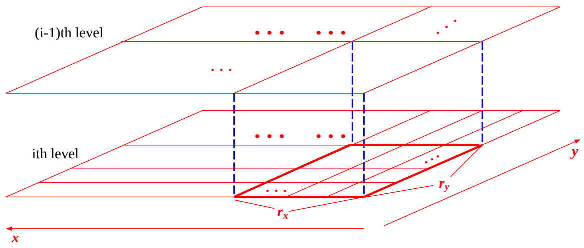


Figure 1: Multiple level division of the index space

Looking ahead, the PK-tree will have nodes, each of which corresponds to a cell at some level. The parent-child relationship corresponds naturally to the cell-subcell relationship. Let L_D be the minimum level such that all points in a set D are in separate cells. If all cells to level L_D are used in such a tree, the number of nodes is very large and the structure will tend to be inefficient. A PK-tree basically applies a simple rule to eliminate cells from the pyramid in a such way that the result is an efficient tree structure with a number of desirable properties, including uniqueness, bounded number of children, non-overlapping of sibling cells and a bound on the average height.

3.2 Definition of K-instantiable Cells

The concept of K-instantiable cells is fundamental to the definition of PK-trees and thus, we begin with the formal definition in this section. For any data point d, we use (x_d, y_d) to denote its location. Let C_0 be the rectangular region containing all possible data points, $C_0 = \{d | X_{min} \leq x_d < X_{max}, Y_{min} \leq y_d < Y_{max}\}$ where $X_{min}, X_{max}, Y_{min}$, and Y_{max} are the predefined minimum and maximum coordinates along the X and Y dimensions, respectively. We call C_0 the *index space*.

Definition 3.1 *Given an index space C_0, a finite set of points D and dividing ratio R, a cell which contains only one point (or several colocated points) and its parent cell contains more than one noncolocated points is called a point cell*[1]*.*

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 $x_{C1} < x_{C2}$ and $y_{C1} < y_{C2}$. For any data point $d \in C_0$, we say $d \in C$ iff both of the following conditions are satisfied:*

1. $x_{C1} \leq x_d < x_{C2}$;

[1] The definition of a point cell is slightly different from the definition in the paper submitted to SIGMOD'98. However, the resulting PK-tree structure remains the same.

2. $y_{C1} \le y_d < y_{C2}$.

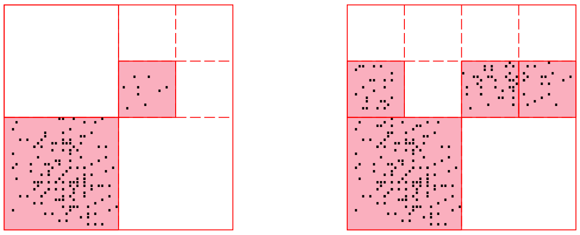
Otherwise, $d \notin C$.

We view each cell as the set of points contained in it and use set notation to express relationships among cells. For example, we say $C_1 \subseteq C_2$ iff $\forall d \in C_0 (d \in C_1 \Longrightarrow d \in C_2)$. We also say that 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 $C_1 \subset C_2$. Let D be the set of all data points in an application, $D \subset C_0$. The cardinality of D is finite and denoted by N.

Definition 3.3 *Given a finite set of points* $D \subset C_0$ *and a dividing ratio* $R = (r_x, r_y)$*, a cell* C *is* K*-instantiable* $(K > 1)$ *iff*

1. *C is a point cell, or*
2. $\nexists$ $K-1$ *K-instantiable cells* $C_1, \ldots, C_{K-1} \subset C$*,*
   such that $\forall d \in D (d \in C \Longrightarrow d \in \bigcup_{i=1}^{K-1} C_i)$.

This definition of K-instantiable cell can be viewed in a bottom-up recursive fashion. First, all point cells are K-instantiable. Then for any cell, if all points in it can not be covered by less than K K-instantiable subcells, then it is K-instantiable. For example, the cell in Figure 2(a) is not 3-instantiable since all data points can be covered by 2 3-instantiable proper sub-cells, whereas the cell in Figure 2(a) is 3-instantiable. (The shaded areas are 3-instantiable cells).



(a) (b)

Figure 2: Examples of non-3-instantiable and 3-instantiable Cells

Theorem 3.1 *For a given area* C_0*, data set* D*, and dividing ratio* R*, the set of* K*-instantiable cells is unique.*

Proof. Assume that there exist two distinct sets of K-instantiable cells, S_1 and S_2 ($S1 \neq S2$), for the same data set D. Let l be the highest (largest numbered) level at which the cells of S_1 do not match the cells of S_2. Without loss of generality, let $C_i \in S_1$, and $C_i \notin S_2$. If C_i is a point cell, then $C_i \in S_2$ and it contradicts our assumption. Therefore, C_i is a intermediate cell. Since C_i is K-instantiable ($C_i \in S_1$), $\nexists K-1$ or less K-instantiable cells at higher levels that cover all points in C_i. However, the same must be true for S_2 because S_2 have same K-instantiable cells at level higher than l. This contradicts with our assumption. Therefore, this theorem holds. □

3.3 Definition of PK-tree

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

1. *The cell at level* 0 *(corresponding* C_0*) is always instantiated and serves as the root of the PK-tree.*
2. *Except the root, every node in the PK-tree is mapped one-to-one with a* K*-instantiable cell.*
3. *For any two nodes* C_1 *and* C_2 *in the 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 the PK-tree such that* $C_1 \subset C_3$ *and* $C_3 \subset C_2$.

Note that we refer to "cells" in a PK-tree to mean the node associated with that cell. 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 $C \subset C_2$. The center location (x, y) and the level l of each non-leaf cell are stored as $C.x$, $C.y$, and $C.level$ respectively. In addition, the number of data points contained in C, $C.datanum$, is also stored in each non-leaf node of PK-tree. All point cells serve as leaf nodes in a PK-tree. For the leaf nodes, the $C.x$ and $C.y$ are the coordinates of the point instead of the center location of the point cell. Moreover, a special number is used for $C.level$, (e.g., -1) in leaf nodes. The range of the X and Y dimensions of C_0 and the rank of the tree are stored in the root of the PK-tree.

Figure 3(a) illustrates a set of data points. Figure 3(b) and (c) show the possible intermediate nodes. Figure 3(d) is the PK-tree of rank 3 built on this data set with dividing ratio $r_x = 2$ and $r_y = 2$.

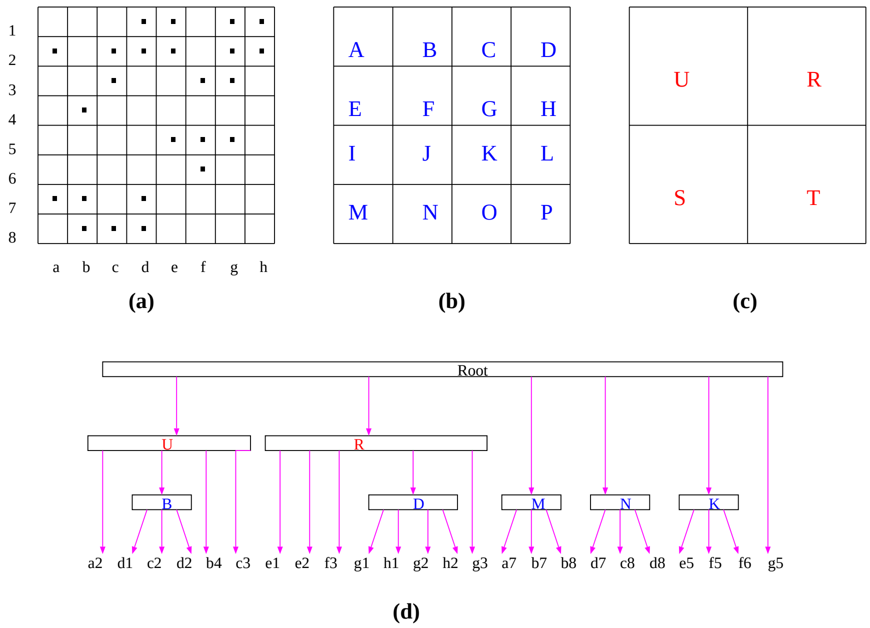


Figure 3: An Example of a PK-tree of rank 3

Intuitively, a PK-tree can be constructed by the following method. Let T_p be a pyramid tree for index space C_0, but with all empty cells and point cell descendants removed. Then all non K-instantiable nodes are removed from T_p. When a non-K-instantiable node is removed, its children became children of the parent of the removed node. The result is the PK-tree. In some of the discussion to follow we will use this relationship between T_p and the PK-tree.

Theorem 3.2 *For a given finite set of points D of cardinality N, in an index space C_0, and dividing ratio $R = (r_x, r_y)$, there exists a unique PK-tree of rank K $(K > 1)$.*

Proof. The pyramid tree T_p is unique for a given finite data set D, index space C_0, dividing ratio R, and rank K $(K > 1)$. The set of K-instantiable cells is unique as proven in Theorem 3.1, and in turn the set of nodes to be removed from T_p is unique. If before removing a node, the structure is a tree, then the structure remains a tree after removing the node. A simple interchange argument on the sequence of nodes to remove can be used to prove that the resulting PK-tree is unique. Therefore, the theorem holds. □

4 Properties

Lemma 4.1 *For any two nodes C_1 and C_2 in a PK-tree, if $C_1 \subset C_2$ then there exists a unique child C_3 of C_2 such that $C_1 \subseteq C_3$.*

Proof. If this lemma holds for any two nodes in a tree before removing a node, then the lemma is still true after removing a node. It holds for T_p and the PK-tree results from a sequence of deletions. Therefore, this lemma holds. □

Lemma 4.2 *For any intermediate node C in a PK-tree and any data point $d \in D$, if $d \in C$, then there exists a unique child C_1 of C such that $d \in C_1$.*

Proof. For any intermediate cell C and data point $d \in D$, let C_d be the point cell such that $d \in C_d$. If $d \in C$, then $C_d \subset C$. By Lemma 4.1, there must exist a unique child C_1 of C such that $C_d \subseteq C_1$. Thus, $d \in C_1$. Therefore, this lemma is proved. □

Theorem 4.3 *For a given index space C_0 and dividing ratio $R = (r_x, r_y)$, each intermediate node in a PK-tree of rank K ($K > 1$) has at least K and at most $(K-1) \times r_x \times r_y$ children.*

The proof of this theorem is in Appendix A. Theorem 4.3 gives us the range of the number of children of an intermediate node in a PK-tree. This range facilitates the choice of dividing ratio R of the space and the rank K of the PK-tree to be built according to some goal. For example, one reasonable consideration is to put all children of an intermediate node in one page to improve the retrieval time.

Theorem 4.4 *Any PK-tree of rank K for a finite set points D, of cardinality N, index space C_0 and 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.*

Proof. Let X be 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 Theorem 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:

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

Solving this equation, we find the range of the total 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 theorem holds. □

Statistical Properties:

Let $D = \{x_1, x_2, \ldots, x_N\}$ be a random sample of N points in an index space C_0 where $P_D(x_1, x_2, \ldots, x_N)$ is the probability distribution for D. For a given C_0, K, 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 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 those cells that are never K-instantiable for any sample data set; any cell for which $P_i = 1$ only contains points in one subcell at the next level and is not K-instantiable.) For a given sub-path sequence ρ', the first q elements in ρ' is called the *q-sub-path sequence* and denoted by ρ_q. If ρ' has less than q elements, then ρ_q is the same as ρ'. It is easy to see that there exists at most r^q distinct q-sub path sequences for any data set where $r = r_x \times r_y$. (Note that if a node C_i is dropped out of a path sequence, because $P_i = 1$ then in any other cell sequence containing C_i, $P_i = 1$ or 0.)

Let $f(N) = O(\log N)$. For a given sub-path sequence ρ', let ψ_f be the smallest real number such that $|\{P_j \mid P_j > \psi_f\}| \leq f(N)$, and let Ψ be the least upper bound of ψ_f over all sub-path sequences. Also, let $\alpha_{\rho'} = |\{P_j \mid P_j > \Psi\}|$ 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, (i.e., each point is independently distributed uniformly over the region) $\Psi = \frac{1}{r_x \times r_y}$. For other general classes of distributions of points, simple but sufficient conditions for $\Psi < 1$ are described in [Wan97]. When $\Psi < 1$, we are able to show that the average height of the PK-tree is $O(\log N)$. This is the subject of the remainder of this section.

Lemma 4.5 *Given a PK-tree of rank K ($K > 1$) and height h ($h > 1$) for a data set D, there exists a sub-path sequence ρ' such that, for any integer l ($1 \leq l \leq h$), the lth element of ρ', denoted $C(\rho', l)$ contains at least $(K-1) \times (h-l) + 1$ data points in D.*

[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.

Proof. Any path from the root of a PK-tree to a leaf node (minus the root node) is a subsequence of some sub-path sequence. Let PATH be the longest path from the root to a leaf (not including the root node). $PATH$ is a subsequence of some sub-path sequence ρ'. The lth node in $PATH$ is the $(l+\delta)$th element in ρ' where $\delta \geq 0$ is an integer and $C(\rho', l+\delta)$ contains at least $(K-1)(h-l)+1$ data points. Therefore, $C(\rho', l)$ contains at least $(K-1)(h-l)+1$ data points in D. □

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

Proof. Let $R = (r_x, r_y)$ be the dividing ratio of the space and $r = r_x \times r_y$. For a sub-path sequence ρ' and an integer $l > 0$, the lth element of ρ is denoted by $C(\rho', l)$. For a given sub-path sequence ρ', the probability that at least $(K-1) \times (H-l)+1$ data points are in $C(\rho', l)$ is

$$p_l \leq \sum_{k=(K-1)(H-l)+1}^{N} \binom{N}{k} \left(\Psi^{\max(l-\alpha,0)}\right)^k \left(1 - \Psi^{\max(l-\alpha,0)}\right)^{N-k},$$

the probability that there exists at least one sub-path sequence ρ' such that at least $(K-1)(H-l)+1$ data points are in $C(\rho', l)$ is

$$\begin{aligned}
P_l &< r^l \times p_l \\
&\leq r^l \times \sum_{k=(K-1)(H-l)+1}^{N} \binom{N}{k} \left(\Psi^{\max(l-\alpha,0)}\right)^k \left(1 - \Psi^{\max(l-\alpha,0)}\right)^{N-k} \\
&< r^l \times \sum_{k=(K-1)(H-l)+1}^{N} \binom{N}{k} \left(\Psi^{\max(l-\alpha,0)}\right)^k
\end{aligned}$$

since there are at most r^l distinct l-sub-path sequences. We want to calculate the probability that the height of the PK-tree is larger than H. In this case, by Lemma 4.5, for each l $(1 \leq l \leq H)$, there must exist a sub-path sequence ρ' such that there are at least $(K-1)(H-l)+1$ data points in $C(\rho', l)$. Therefore, the probability

$$P(h > H) \leq \min_{1 \leq l \leq H} P_l \leq P_l$$

for any $1 \leq l \leq H$. Let's take $l = \lfloor \frac{H}{c_2} \rfloor$ where $1 < c_2 < c$ is a small constant. In turn, $l > \alpha$. We have $P(h > H) \leq P_l$ and

$$P_l < r^l \times \sum_{k=(K-1)(H-l)+1}^{N} \binom{N}{k} \left(\Psi^{l-\alpha}\right)^k.$$

Through some algebraic manipulation, from the above expression, it can be shown that $P_l = O\left(\Psi^{H^2}\right)$. (The details are in Appendix A.) Given this result, we have $P(h > H) = O\left(\Psi^{H^2}\right)$. □

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

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

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

Thus, this theorem holds. □

When Ψ is bounded by a constant less than 1, $N \times \Psi^{\log^2 N}$ approaches 0 for large N. As a result, the expected height of the tree is $O(\log N)$ when $\Psi < 1$. If Ψ is a function of N, then a sufficient condition for the expected height of a PK-tree to be $O(\log N)$ is $\Psi = O\left(\sqrt[\log^2 N]{\frac{\log N}{N}}\right)$.

5 PK-tree Generation

There is a trivial PK-tree generation algorithm: inserting point by point into a PK-tree. The complexity of this point-by-point algorithm is $O(N \log N)$ on average, but $O(N^2)$ in the worst case. In this section, we present an efficient algorithm to generate a PK-tree, which has complexity $O(N \log N)$ in the worst case. The algorithm has two phases. First the set of points D is sorted to into a certain order, which we call PK-order. The second phase is to generate a PK-tree from the sorted data.

5.1 PK-order

Definition 5.1 *For a given data set D, a data visiting sequence of D, denoted by $seq(D)$, is a permutation of all data points in D.*

Definition 5.2 *For a given data set D of cardinality N, a data visiting sequence $seq(D) = d_1, d_2, \ldots, d_N$ is a PK-ordered sequence iff $\nexists$ three data points $d_i, d_j, d_k \in D$ $(1 \leq i < j < k \leq N)$ and a grid cell C in a pyramid tree such that $d_i, d_k \in C$ but $d_j \notin C$.*

The key idea behind this definition is that if we visit data points (or point cells) in PK-order, then for any cell at any level, once we visit a data point within this cell, we will continue to visit all other data points within this cell (in some order) before visiting any data point outside this cell. As a result, for any given cell, the data points covered by that cell all appear together in the PK-order sequence. Some well-known space filling curves [Fal89][Jag90] such as Z-order [Ore84][Ore86], the Gray-code curve [Fal88], and Hilbert curve [Kam94] are all special cases of the PK-order. For example, we have a data set D containing 8 data points whose locations are $(15, 30)$, $(40, 30)$, $(15, 70)$, $(55, 10)$, $(95, 90)$, $(70, 130)$, $(80, 110)$, and $(95, 175)$ respectively in a two dimensional space with dividing ratio $R = (2, 2)$. The spatial area is $X_{min} = 0, Y_{min} = 0, X_{max} = 100, Y_{max} = 200$. The visiting sequence illustrated in Figure 4(a) and (b) are in PK-order, but the visiting sequence in Figure 4(c) is not PK-order because there exist three points $(55, 10)$, $(80, 110)$, $(95, 90)$ and the shaded cell C such that $(55, 10), (95, 90) \in C$, $(80, 110) \notin C$ and $(80, 110)$ is visited after $(55, 10)$ and before $(95, 90)$.

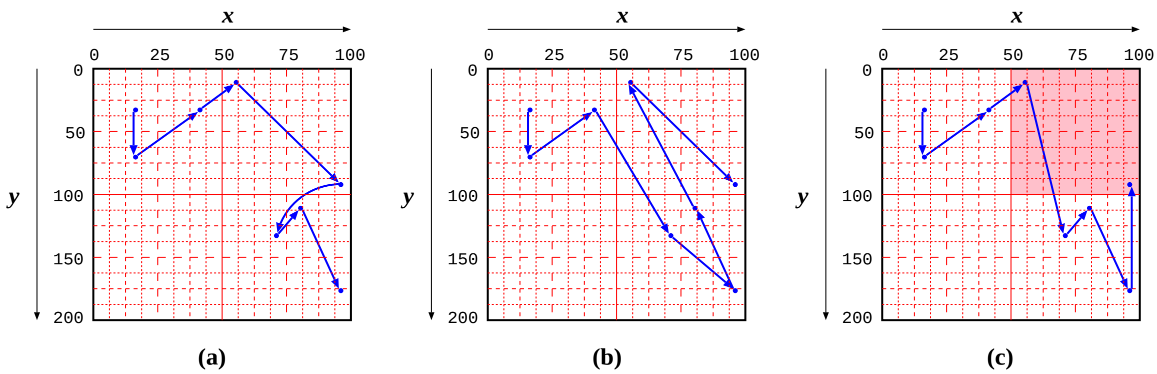


Figure 4: Examples of Visiting Curve

Due to space limitations, we do not present a sorting algorithm for PK-order in this paper. However, it is trivial that there exists a sorting algorithm for PK-order with worse case complexity $O(N \log N)$.

5.2 PK-tree Generation from Sorted Data

We will describe a PK-tree generation algorithm whose complexity is $O(N \log N)$ in the worst case. For a set of N data points over a spatial area, we first find a PK-order sequence of the input data set. We use a stack and a queue.

Initially the stack is empty and the queue contains the sorted data points. We take an element C_d from the head of the queue, then see whether there at least K elements in the stack. If so, we pop K elements from the stack, and find the minimum bounding cell C for those K elements. If $C_d \subset C$, we push back all K elements (in the reverse order they were popped out from the stack) and then push C_d onto the stack. The reason we can not instantiate any nodes at this point is that there could be more data points belonging to C that we have not yet fetched from the queue. Therefore, we can not decide whether or not C is K-instantiable. On the other hand, if $C_d \not\subset C$, then we keep popping elements from the stack until an element is reached which is not a proper subset of C. (We know for sure C is K-instantiable and we want to find all children of C.) We make all elements children of C except the last one. Then we push the last popped out cell, push C, then push C_d onto the stack. We continue this procedure until the queue is empty. Then we instantiate the root and make all the elements in the stack the children of the root. The formal description of this algorithm is in Appendix B.

Now we analyze the complexity of this algorithm. After taking a node C_d from the head of the queue, K nodes in the stack are popped. If $C_d \not\subset C$, then we say it is a *successful pull*. Otherwise, we say it is a *unsuccessful pull*. A successful pull results a creation of an intermediate node and the new intermediate node is pushed onto the stack. Therefore, the number of successful pulls is equal to the number of intermediate nodes, which is $O(N)$. During a successful pull, at most $(K-1) \times r_x \times r_y + 1$ nodes are popped and two nodes (the new created node and the last popped node) are pushed back onto the stack. Thus, a successful pull can be done in constant time. Moreover, each node in the queue starts at most one unsuccessful pull. During an unsuccessful pull, K nodes are popped form the stack and $K+1$ nodes are pushed back onto the stack. So the complexity of unsuccessful pulls is also $O(N)$. Therefore, the overall complexity of this algorithm is $O(N)$. Thus, the complexity for generating a PK-tree from a finite set of unsorted points is $O(N \log N)$ in the worst case.

6 Operations on PK-tree

In this section, we discuss three kinds of operations: insertion, deletion, and search for a point. For a given PK-tree T of rank K, let R_{PK} denote the root of T. Let d denote the data point that is to be inserted into T or deleted from T and let C_d be the point cell that contains d.

6.1 Insertion

The insertion of a data point d into the PK-tree T is achieved by inserting the corresponding leaf node C_d into T. This can be done in two phases: the first phase is to follow the path from the root R_{PK} down to the leaf level to locate all (potential) ancestors of C_d while the second phase is 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 as a PK-tree. The formal description of the insertion algorithm is presented in [Wan97].

The computational complexity of data point insertion is linearly proportional to the height of the PK-tree. Since the average height of the PK-tree is $O(\log N)$, the average computational complexity is $O(\log N)$, where N is the number of data points in the PK-tree.

6.2 Deletion

The deletion of a data point d from the PK-tree T is achieved by deleting the corresponding leaf node C_d from T. Similar to insertion, this can also be done in two phases: in the first phase is to follow the path from the root R_{PK} down to the leaf level to locate all ancestors of C_d and remove C_d while the second phase is to go 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 as a PK-tree. The formal description of the deletion algorithm is also presented in [Wan97]. 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.

6.3 Point Search

Given a location (x, y) and a PK-tree, the algorithm $SINGLEPOINT(R_{PK}, x, y)$ returns the data point d locating at (x, y) if there is one. Otherwise, nil will be returned.

Algorithm 6.1 *Single Point Retrieval*

SINGLEPOINT(C, x, y)
{
  if C *is a leaf node*
  then return $C.data$
  else if $\exists$ *a child* C' *of* C *such that* $(x, y) \in C'$
    then *SINGLEPOINT*(C', x, y)
    else return *nil*
}

It is easy to see that the computational complexity of Algorithm 6.1 is $O(h)$ since the algorithm only visits at most all the nodes along one path from the root to one of its leaf once where h is the height of the PK-tree. Thus, the average complexity is $O(\log N)$ where N is the size of the data set.

7 Conclusion

In this paper, we propose a new dynamic spatial indexing structure, called the PK-tree. In some sense, it can be conceived as a variation of PR quad-trees. However, it differs from all existing trees by employing two special restrictions. 1) Only a node that contains some data points are instantiated. 2) A intermediate node is forced to have certain number of children. The combination of these two constraints is unique. These two restrictions ensure that the number of the total node in a PK-tree is $O(N)$ and the average height of a PK-tree is $O(\log N)$ under some general conditions.

We also proved a set of properties of PK-trees. These properties include: non-overlapping of sibling nodes, uniqueness of the PK-tree for a given set of data points, and so on. R-trees lack these properties so that the performance of R-trees is difficult to predict. Empirical studies indicate that the PK-tree outperforms SR-tree and X-tree by a large margin [Wan97]. Therefore, we see the PK-tree as a preferable indexing method.

References

[Abr95] M. Abrash. BSP Trees. *Dr. Dobbs Sourcebook*, 20(14), 49-52, 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.

[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.

[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.

[Fal88] Christos Faloutsos. Gray codes for partial match and range queries. *IEEE Trans. on Software Engineering*, 14(10):1381-1393, 1988.

[Fal89] Christos Faloutsos and Shari Roseman. Fractals for secondary key retrieval. *Proc. 8th ACM SIGART-SIGMOD Sym. on Principles of Database Systems (PODS)*, 247-252, 1989.

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

[Jag90] H. V. Jagadish. Linear clustering of objects with multiple attributes. *Proc. ACM SIGMOD Conf. on Management of Data*, 332-342, 1990.

[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.

[Kam94] Ibrahim Kamel and Christos Faloutsos. Hilbert R-tree: an improved R-tree using fractals. *Proc. 20th Intl. Conf. on Very Large Data Bases (VLDB)*, 500-509, 1994.

[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.

[Ore84] Jack A. Orenstein and T. H. Merrett. A class of data structures for associative searching. *Proc. 3rd ACM SIGART-SIGMOD Sym. on Principles of Database Systems (PODS)*, 181-190, 1984.

[Ore86] Jack A. Orenstein. Spatial query processing in an object-oriented database system. *Proc. ACM SIGMOD Conf. on Management of Data*, 326-336, 1986.

[Ohs83] Yutaka Ohsawa and Masao Sakauchi. Multidimensional data management structure with Efficient Dynamic Characteristics. *Systems Computers Controls*, vol. 14, no.5, 1983.

[Sam90] Hanan Samet. The design and analysis of spatial data structures. *Addison-Wesley Publishing Company*, 1990.

[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.

[Wan97] Wei Wang, Jiong Yang, and Richard Muntz. PK-tree: a dynamic spatial index structure for large data sets. *UCLA Computer Science Department Technical Report #970039*, 1997. http://dml.cs.ucla.edu/~weiwang/paper/TR-97039.ps

A Proof of PK-tree Properties

Theorem 4.3 For a given two dimensional space with dividing ratio $R = (r_x, r_y)$, each intermediate node in an PK-tree of rank K ($K > 1$) have at least K but at most $(K-1) \times r_x \times r_y$ children.

Proof. First, we prove that each intermediate node in an PK-tree of rank K must have at least K children by contradiction. Assume that there exists an intermediate node C that has less than K children. By Lemma 4.2 and Definition 3.3, C is not K-instantiable. This contradicts that all intermediate nodes in a PK-tree of rank K are K-instantiable cells.

Secondly, we prove that each intermediate node in an PK-tree of rank K can have at most $(K-1) \times r_x \times r_y$ children by contradiction. Assume that there exists an intermediate cell C that has more than $(K-1) \times r_x \times r_y$ children. Then, by the Pigeonhole Theorem, there exists at least one uninstantiated proper sub-cell Cs of C such that at least K of C's children, say $C_1, \ldots, C_K$, are K-instantiable proper sub-cells of Cs. Since Cs is not K-instantiable, there exist at most $K-1$ K-instantiable proper sub-cells of Cs, namely $Cs_1, \ldots, Cs_{K-1}$ such that $\forall d \in D(d \in Cs \Longrightarrow d \in Cs_1 \vee \ldots \vee d \in Cs_{K-1})$. Then there must exist one of them, say Cs_j $(1 \leq j \leq K-1)$ and two cells from C_i $(1 \leq i \leq K)$, say C_{i1} and C_{i2} $(1 \leq i1, i2 \leq K)$, such that both $C_{i1} \subseteq Cs_j$ and $C_{i2} \subseteq Cs_j$. Since C_{i1} and C_{i2} disjoint, we have $C_{i1} \subset Cs_j$ and $C_{i2} \subset Cs_j$. Then C_{i1} and C_{i2} can not be children of C because Cs_j is an K-instantiable proper sub-cell of C and $C_{i1} \subset Cs_j$, $C_{i2} \subset Cs_j$. This contradicts the assumption that C_{i1} and C_{i2} are children of C. Therefore, we prove that each intermediate node in an PK-tree of rank K has at most $(K-1) \times r_x \times r_y$ children. $\Box$

Algebraic Manipulation of P_l

Now we calculate an upper bound for $P_l < r^l \times \sum_{k=(K-1)(H-l)+1}^{N} \binom{N}{k} \left(\Psi^{l-\alpha}\right)^k$ where $l = \frac{H}{c_2}$. Since $0 < \Psi < 1$ and $\alpha = O(\log N)$, we have $\left(\frac{1}{\Psi}\right)^{l-\alpha} = \left(\frac{1}{\Psi}\right)^{\Theta(H)-O(\log N)} = \left(\frac{1}{\Psi}\right)^{\Theta(\Omega(\log N))-O(\log N)} = \left(\frac{1}{\Psi}\right)^{\Omega(\log N)} = \Omega(N) \geq c_1 \times N$ where $c_1 > 0$ is a constant. Then

$$\begin{aligned}
\binom{N}{k+1} \left(\Psi^{l-\alpha}\right)^{k+1} &= \frac{N-k}{(k+1) \times \left(\frac{1}{\Psi}\right)^{l-\alpha}} \binom{N}{k} \left(\Psi^{l-\alpha}\right)^k \\
&< \frac{1}{c_1 \times (k+1)} \binom{N}{k} \left(\Psi^{l-\alpha}\right)^k
\end{aligned}$$

$$
\begin{aligned}
&\leq \frac{1}{c_1 \times ((K-1)(H-l)+2)} \binom{N}{k} \left(\Psi^{l-\alpha}\right)^k \\
&< \frac{\binom{N}{(K-1)(H-l)+1} \left(\Psi^{l-\alpha}\right)^{(K-1)(H-l)+1}}{\left(c_1 \times ((K-1)(H-l)+2)\right)^{k-(K-1)(H-l)-1}}.
\end{aligned}
$$

In this case,

$$
\begin{aligned}
P_l &< r^l \times \sum_{k=(K-1)(H-l)+1}^{N} \frac{\binom{N}{(K-1)(H-l)+1} \left(\Psi^{l-\alpha}\right)^{(K-1)(H-l)+1}}{\left(c_1 \times ((K-1)(H-l)+2)\right)^{k-(K-1)(H-l)-1}} \\
&= r^l \times \binom{N}{(K-1)(H-l)+1} \left(\Psi^{l-\alpha}\right)^{(K-1)(H-l)+1} \times \\
&\quad \sum_{k=(K-1)(H-l)+1}^{N} \frac{1}{\left(c_1 \times ((K-1)(H-l)+2)\right)^{k-(K-1)(H-l)-1}} \\
&= r^l \times \binom{N}{(K-1)(c_2-1)l+1} \left(\Psi^{l-\alpha}\right)^{(K-1)(c_2-1)l+1} \times \\
&\quad \sum_{k=(K-1)(c_2-1)l+1}^{N} \frac{1}{\left(c_1 \times ((K-1)(c_2-1)l+2)\right)^{k-(K-1)(c_2-1)l-1}}.
\end{aligned}
$$

When $l \leq \frac{1-2c_1}{c_1(c_2-1)(K-1)}$ we have $c_1((K-1)(c_2-1)l+2) \leq 1$ and since $r^l = r^{\Omega(\log N)} = \Omega(N)$, we have $N = O(r^l) < c_3 \times r^{\frac{1-2c_1}{c_1(c_2-1)(K-1)}}$ where c_3 is a constant. Then

$$
\begin{aligned}
&\sum_{k=(K-1)(c_2-1)l+1}^{N} \frac{1}{\left(c_1 \times ((K-1)(c_2-1)l+2)\right)^{k-(K-1)(c_2-1)l-1}} \\
<\; &\sum_{k=(K-1)(c_2-1)l+1}^{c_3 \times r^{\frac{1-2c_1}{c_1(c_2-1)(K-1)}}} \frac{1}{\left(c_1 \times ((K-1)(c_2-1)l+2)\right)^{k-(K-1)(c_2-1)l-1}}
\end{aligned}
$$

which is a constant upper bound.

On the other hand, if $l > \frac{1-2c_1}{c_1(c_2-1)(K-1)}$, then $c_1((K-1)(c_2-1)l+2) > 1$ and

$$
\sum_{k=(K-1)(c_2-1)l+1}^{\infty} \frac{1}{\left(c_1 \times ((K-1)(c_2-1)l+2)\right)^{k-(K-1)(c_2-1)l-1}}
$$

is a converged geometric sum. We have

$$
\begin{aligned}
&\sum_{k=(K-1)(c_2-1)l+1}^{N} \frac{1}{\left(c_1 \times ((K-1)(c_2-1)l+2)\right)^{k-(K-1)(c_2-1)l-1}} \\
<\; &\sum_{k=(K-1)(c_2-1)l+1}^{\infty} \frac{1}{\left(c_1 \times ((K-1)(c_2-1)l+2)\right)^{k-(K-1)(c_2-1)l-1}} \\
=\; &\frac{c_1((K-1)(c_2-1)l+2)}{c_1((K-1)(c_2-1)l+2)-1}.
\end{aligned}
$$

We take c_4 as the larger one of those two constants.

$$
\begin{aligned}
c_4 &= \max\left(\sum_{k=(K-1)(c_2-1)l+1}^{c_3 \times r^{\frac{1-2c_1}{c_1(c_2-1)(K-1)}}} \frac{1}{\left(c_1 \times ((K-1)(c_2-1)l+2)\right)^{k-(K-1)(c_2-1)l-1}}, \right. \\
&\quad \left. \frac{c_1((K-1)(c_2-1)l+2)}{c_1((K-1)(c_2-1)l+2)-1} \right)
\end{aligned}
$$

Then

$$
\begin{aligned}
P_l &< c_4 \times r^l \times \binom{N}{(K-1)(c_2-1)l+1} \left(\Psi^{l-\alpha}\right)^{(K-1)(c_2-1)l+1} \\
&= \frac{c_4 \times r^l \times \binom{N}{(K-1)(c_2-1)l+1}}{\left(\frac{1}{\Psi}\right)^{(l-\alpha)((K-1)(c_2-1)l+1)}} \\
&< \frac{c_4 \times r^l \times N^{(K-1)(c_2-1)l+1}}{\left(\frac{1}{\Psi}\right)^{(l-\alpha)((K-1)(c_2-1)l+1)}} \\
&= c_4 \times r^{l+((K-1)(c_2-1)l+1)\log_r N-(l-\alpha)((K-1)(c_2-1)l+1)\log_r\left(\frac{1}{\Psi}\right)} \\
&= \frac{c_4}{r^{l(l-\alpha)(K-1)(c_2-1)\log_r\left(\frac{1}{\Psi}\right)+(l-\alpha)\log_r\left(\frac{1}{\Psi}\right)-l-((K-1)(c_2-1)l+1)\log_r N}}
\end{aligned}
$$

Since $l = \frac{H}{c_2}$, $\alpha < \frac{H}{c}$, we have $l-\alpha > \frac{H}{c_2} - \frac{H}{c} = \frac{(c-c_2)\times H}{c \times c_2} = \Omega(H)$ on one hand and $l - \alpha < l = \Theta(H)$ on the other hand. So, $l-\alpha = \Theta(H)$. Thus, $l(l-\alpha)(K-1)(c_2-1)\log_r\left(\frac{1}{\Psi}\right)$ is the dominate term in $l(l-\alpha)(K-1)(c_2-1)\log_r\left(\frac{1}{\Psi}\right) + (l-\alpha)\log_r\left(\frac{1}{\Psi}\right) - l - ((K-1)(c_2-1)l+1)\log_r N$. So,

$$
\begin{aligned}
& l(l-\alpha)(K-1)(c_2-1)\log_r\left(\frac{1}{\Psi}\right) + (l-\alpha)\log_r\left(\frac{1}{\Psi}\right) - l - ((K-1)(c_2-1)l+1)\log_r N \\
= {}& \Theta\left(l(l-\alpha)(K-1)(c_2-1)\log_r\left(\frac{1}{\Psi}\right)\right) \\
= {}& \Theta\left(H^2 \log_r\left(\frac{1}{\Psi}\right)\right).
\end{aligned}
$$

In turn,

$$
\begin{aligned}
P_l &< \frac{1}{r^{\Theta\left(H^2\log_r\left(\frac{1}{\Psi}\right)\right)}} \\
&= O\left(\Psi^{H^2}\right).
\end{aligned}
$$

B PK-tree Generation Algorithm

Algorithm B.1 *Build PK-tree*

BUILDPKTREE$(D, N, K, X_{min}, X_{max}, Y_{min}, Y_{max}, r_x, r_y)$
{
  $E \leftarrow PK\_SORT(D, N, X_{min}, X_{max}, Y_{min}, Y_{max}, r_x, r_y)$
  a stack S is initiated to be $\emptyset$
  for $j \leftarrow 1$ **to** N
    $d \leftarrow E[j]$
    instantiate the corresponding leaf node C_d
    BUILD-SUB-TREE$(S, C_d, K, X_{min}, X_{max}, Y_{min}, Y_{max}, r_x, r_y)$
  pop(S, C)
  if C *is at level* 0
  then $R_{PK} \leftarrow C$
  else
    instantiate root R_{PK}
    make C a child of R_{PK}
    while $S \neq \emptyset$
      pop(S, C)
      make C a child of R_{PK}
  return R_{PK}
}

```
BUILD-SUB-TREE(S, C_d, K, X_min, X_max, Y_min, Y_max, r_x, r_y)
{
    i ← 1
    while i ≤ K  and  S ≠ ∅
        pop(S, Cells[i])
        i ← i + 1
    if i = K + 1
    then
        C ← FIND-MIN-CELL(Cells, K, X_min, X_max, Y_min, Y_max, r_x, r_y)
        if C_d ⊂ C
        then
            for k ← K downto 1
                push(S, Cells[k])
        else
            instantiate C
            for k ← 1 to K
                make Cells[k] a child of C
            pop(S, C')
            while C' ⊂ C
                make C' a child of C
                pop(S, C')
            push(S, C')
            BUILD-SUB-TREE(S, C_d, K, X_min, X_max, Y_min, Y_max, r_x, r_y)
}

FIND-MIN-CELL(Cells, K, X_min, X_max, Y_min, Y_max, r_x, r_y)
{
    l ← DETERMINE_LEVEL(Cells, K, X_min, X_max, Y_min, Y_max, r_x, r_y)
    x_unitsize ← (X_max − X_min)/r_x^Level)
    y_unitsize ← (Y_max − Y_min)/r_y^Level)
    for i ← 1 to K
        D[i].x ← ⌊(A[i].x − X_min)/x_unitsize⌋
        D[i].y ← ⌊(A[i].y − Y_min)/y_unitsize⌋
        D[i].xy ← BIT_INTERLEAVE (D[i].x, r_x, D[i].y, r_y)
    i ← 0
    b ← max(r_x^2, r_y^2)
    while true
        if CHECK_DIGIT(D, K, b, i)
        then break
        else i ← i + 1
    C_level ← l − i
    x_unitsize ← ⌊(X_max − X_min)/r_x^(l−i)⌋
    y_unitsize ← ⌊(Y_max − Y_min)/r_y^(l−i)⌋
    C.x ← ⌊(D[1].x − X_min/X_unitsize⌋ × X_unitsize + X_min + X_unitsize/2
    C.y ← ⌊(D[1].y − Y_min/Y_unitsize⌋ × Y_unitsize + Y_min + Y_unitsize/2
    return C
}

CHECK_DIGIT(D, K, b, i)
{
    for j ← 1 to i
        if ⌊D[1].xy/b⌋ ≠ ⌊D[j].xy/b⌋
            then return false
    return true
}
```