Complexity Analysis of 2D KD-Tree Construction, Query and Modification Based on Master Theorem and Potential Energy Analysis

***Abstract*—With the increasing popularity of machine learning today, high-dimensional space retrieval has become an extremely important research direction, which has important applications in classification, clustering and database. Among them, KD-Tree is a very important and classical high-dimensional spatial retrieval data structure. This paper introduces the application of master theorem and potential energy analysis in the complexity analysis of 2D KD-Tree, discusses the complexity of KD-Tree construction, nearest neighbor query, deletion and insertion with rebuild, and focuses on the influence of dimension factors in the region query in 2D KD-Tree.**

***Index Terms*—Master theorem, KD-Tree, Amortized Analysis, Potential method**

I. Introduction

KD-Tree is a data structure that organizes points in K-dimensional Euclidean space. It supports a variety of query algorithms such as nearest neighbor search and range search, which has good performance. [1] It plays an important role in machine learning classification and retrieval of special databases. In this paper, the master theorem and potential energy analysis are used to derive the complexity of KD-Tree operations in two dimensions, and a definite bound is given.

II. Related Works

We first introduce some of the methods used to prove complexity related to recursion and data structure maintenance, including the master theorem and potential energy analysis.

A. Time complexity analysis method

1. Master theorem

Consider the complexity of computing the following recursion:

$$T(n) = \begin{cases} O(1), & \text{if } n = 1 \\ aT(\frac{n}{b}) + f(n), & \text{otherwise.} \end{cases} \quad (1)$$

In this recursion, $f(n)$ is a polynomial bounded function, which means that there exists constant c, k, n_0, satisfy $\forall n > n_0,\ f(n) \leq c \times n^k$.

Theorem 1: Solution to Master therom: [2]

$$T(n) = \begin{cases} O(n^k \log_b n), & \text{if } a = b^k \\ O(n^{\log_b a}), & a > b^k \\ O(f(n)), & a < b^k \text{and f(n) satisfy} \\ & \text{Regularity condition} \end{cases} \quad (2)$$

Defination 1: Regularity condition:

If a function $f(n)$, for any large enough n, there exists $c \in (0, 1)$ that satisfy $af(\frac{n}{b}) < cf(n)$, we say that $f(n)$ satisfy Regularity condition.

2. Potential energy analysis

Potential energy analysis belongs to the category of amortization analysis. Amortization also analyzes the average time it takes to perform a particular operation in a sequence of operations used to solve a data structure. It does not involve probability, so the analysis results guarantee average performance in the worst case.

The core idea of potential energy analysis is to store the cost of adjusting data structure in the future as potential energy. Each data structure is mapped to a potential function, so the amortized cost of each operation can be equivalent to the sum of its actual cost and the change in potential energy, which facilitates the analysis of the average cost.

B. A rough time complexity proof

Most existing proofs has proved operations detailly except range query. [3] For those who mention the complexity of range query, they just consider the regions that are crossed by a hyperplane. [4], [5] Actually it's not only a hyperplane that determine the query range. Usually it's a combination of more than one dimension's coordinates constrains. So this kind of proof is neither rigorous nor intuitive at most of the time.

III. Analysis of complexity

A. Build KD-Tree from static point set

The first thing we talked about was how to build a balanced tree. So before we do that, let's first define what a balanced tree is.

Defination 2: Balanced binary tree: If we define height of a binary tree is the maximum number of edges from root to any of it's leaf, a balanced binary tree's height is $\Theta(\log n)$.

When the tree is built, the point set is sorted according to a specified dimension in each round of division, the median is taken as pivot. Points less than pivot on this dimension are divided into left subtree, and the other points are divided into right subtree. The parent node pointer is maintained simultaneously.

But in fact, it does not need to be completely ordered. [6] Careful thinking, this step only needs: firstly, find the median as the pivot for division, which is a very famous sequential order statistics problem. There are quick-select algorithm that can be implemented in liner time complexityn. Secondly, divided sequence into two parts through this pivot. This is another classical problem which can be achieved with the idea of two-pointers. [7] In C++, these two implementations have been unified into the STL template library, namely the nth_element function.

So the pseudo code can be showed as follows.

Algorithm 1 Build KD-Tree

Input: A set of N points in 2-dimension
Output: A well-built KD-Tree

```
BUILD(Node, Dimension, Points)
    if (Points is empty) then
        return
    end if
    divide Points on variable Dimension
    Total ← Points.Size
    Middle ← ⌊(Total+1)/2⌋
    LSet ← NullSet
    RSet ← NullSet
    for (i = 1; i ≤ Middle − 1; i = i + 1) do
        LSet.Insert(Points_i)
    end for
    for (i = Middle + 1; i ≤ Total; i = i + 1) do
        RSet.Insert(Points_i)
    end for
    Node ← Points_Middle
    Dimension ← x if Dimsension is y else y
    BUILD(Node.LeftSon, Dimension, LSet)
    BUILD(Node.RightSon, Dimension, RSet)
```

Let's prove the correctness of build tree from the aspect of tree's height.

Theorem 2: Balanced binary tree: Using the median as the pivot, the height of the KD-Tree with n nodes is built at a time complexity of $O(\log n)$.

Proof: If define $H(n)$ as height of a balanced binary tree with n nodes, we get recursive formula immediately:

$$H(n) = \begin{cases} O(1), & \text{if } n \ = \ 1 \\ H(\frac{n}{2}) + 1, & \text{otherwise.} \end{cases} \tag{3}$$

By using master theorem, it's easy to find $H(n) = \log_2 n + 1$. So the KD-Tree is balance if built by the method below. At the same time, we get complexity in the same way: (If not specifically mentioned, we use $T(n)$ as time complexity.)

$$T(n) = \begin{cases} O(1), & \text{if } n \ = \ 1 \\ 2\ T(\frac{n}{2}) + O(n), & \text{otherwise.} \end{cases} \tag{4}$$

This recursive formula is so classic that it appears in many algorithms' time complexity's proof, such as quick-sort, divide and conquer. The solution to it is $T(n) = O(n \log n)$.

B. Nearest neighbor search

The goal of nearest neighbor search is to query the distance between a given point and the Euclidean closest point in a given static point set in a KD-Tree.

The nearest neighbor search of KD-Tree uses a heuristic algorithm to find the nearest distance. First, the distance between the current node and the query point is compared to update the answer, then the two sub-nodes are compared. If the point is closer to the left sub-node, the left subtree is searched first, otherwise the right subtree is searched first. When a subtree finishs search, if there is no point in the region formed by each dimensiona's upper and lower bound of the point set in another subtree, where the distance between point and the query point is less than the answer, the program can directly prune. Otherwise it needs to recurse into another subtree.

The pseudo code is showed as follows.

Algorithm 2 Adjacent Query

Input: A KD-Tree and a query point
Output: Minimum euclidean distance between query point and points in KD-Tree

```
MD(Node, Point)
    Range ← Node.Range
    return min dist among Point and all points in Range

ADJACENT-QUERY(Node, Point)
    if (Node is Null) then
        return
    end if
    Answer ← MIN{Answer, DIST(Node.point, Point)}
    DistL ← DIST(Node.LeftSon.point, Point)
    DistR ← DIST(Node.RightSon.point, Point)
    if (DistL ≤ DistR) then
        ADJACENT-QUERY(Node.LeftSon, Point)
        if (MD(Node.RightSon, Point)≥Answer) then
            return
        else
            ADJACENT-QUERY(Node.RightSon, Point)
        end if
    else
        ADJACENT-QUERY(Node.RightSon, Point)
        if (MD(Node.LeftSon, Point)≥Answer) then
            return
        else
            ADJACENT-QUERY(Node.LeftSon, Point)
        end if
    end if
```

It is not difficult to find that this process is actually constantly pruning optimization according to the current retrieved answer. The running time will have a lot to do with the data. In the worst case, the program will traverse the entire KD-Tree, so the complexity of the upper bound is $O(n)$. If fortunate enouth, the program gets answer by traverse only a chain from root to any of it's leaf. The lower bound is $O(\log n)$. However,

the actual data distribution is more random. Based on this fact, its performance is also relatively good.

C. Range query

In a given static point set, query the information of the point set that satisfies that each dimension coordinate is within a given range. This is the classic range query problem on KD-Tree.

For instance, we count how many points $(x,\ y)$ satisfy the limitation that $x \in [x_1, x_2]$ and $y \in [y_1, y_2]$, in which x_1, y_1, x_2, y_2 are given in each distinct query.

Therefore, on each KD-Tree node, the maximum and minimum value of each dimension of the hypercube corresponding to the subtree should be recorded, that is, the boundary of the hypercube should be maintained before all queries.

The query operation needs to recursively backtrack on the KD-Tree, and return directly when a subtree is completely included or has no intersection at all. Otherwise continue recursively to the two subtrees.

The pseudo code is showed as follows.

Algorithm 3 Range Query

Input: A KD-Tree and a range
Output: Number of points in this range

```
RANGE-QUERY(Node, Range)
    if (Node is Null) then
        return 0
    end if
    if (Node.Range is in Range) then
        return Node.SubtreeSize
    end if
    if (Node.Range has no intersection with Range) then
        return 0
    end if
    Sum ← 0
    Sum+ = RANGE-QUERY(Node.LeftSon, Range)
    Sum+ = RANGE-QUERY(Node.RightSon, Range)
    return Sum
```

This complexity may seem surprising at first glance, and may become worst-case linear like nearest neighbor search, but it can actually be proven to be $O(\sqrt{n})$.

Theorem 3: Range query's complexity: The RangeQuery complexity of a balanced two-dimensional KD-Tree with n nodes is $O(\sqrt{n})$.

Proof: Let's prove it in three cases. We just need to weaken the condition limitation.

1. Only one dimension has one way limitation.

For example, we just count points whose coordinate (x, y) satisfied $x \leq x'$. This is a kind of query case 1. Similarly, replace the x to y, or replace less than to greater than, which is also included in this case.

To prove it intuitively, we draw the region divide by KD-Tree's node. But we don't draw them all, just 4 regions are needed. They are a single node's son's subtree.

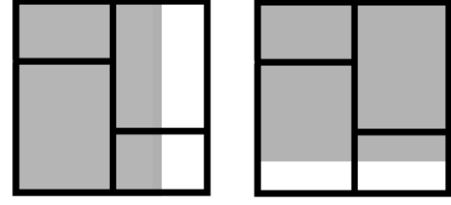

Fig. 1: Samples of only one dimension has one way limitation.

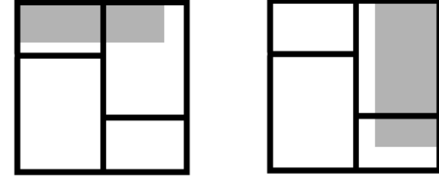

Fig. 2: Samples of two dimensions both has one way limitation, 2 regions have intersection.

Due to all pictures' captions are the same, we explain it here. Black lines form the region, and grey stands for the query range. We use Fig. 1 as the common cases to demonstrate it's time complexity.

Consider that there are at least two regions that do not need to be recursed, either by the restriction on the abscess or the restriction on the ordinate. They are contained by the query region, or do not intersect with it.

We define $T_1(n)$ as the complexity of range query of this case on a KD-Tree with n nodes. Obviously, in each of the 4 regions, thers is less than $\frac{n}{4}$ points. Therefore, it's good to use $\frac{n}{4}$ as approximate estimation of number. And for process a single node's infomation, it costs constant time, which is $O(1)$. The way we describe $T_1(n)$ is as follows:

$$T_1(n) = \begin{cases} O(1), & \text{if } n\ =\ 1 \\ 2\ T_1(\frac{n}{4}) + O(1), & \text{otherwise.} \end{cases} \tag{5}$$

By using master theorem, the solution is $T_1(n) = O(\sqrt{n})$.

2. Two dimensions both has one way limitation.

In this case, we consider the number of intersection regions of query range and 4 regions. Also, we define $T_2(n)$ as time complexity in case 2.

If there are 2 regions has intersection, the behavior is more likely to be showed in Fig. 2.

There is just one region satisfy case 1. Another one can be calcuated recursively. Now we get the $T_2(n)$'s first expression:

$$T_2(n) = \begin{cases} O(1), & \text{if } n\ =\ 1 \\ T_2(\frac{n}{4}) + T_1(\frac{n}{4}) + O(1), & \text{subcase 1.} \end{cases} \tag{6}$$

If there are 3 regions has intersection, the behavior is showed in Fig. 3.

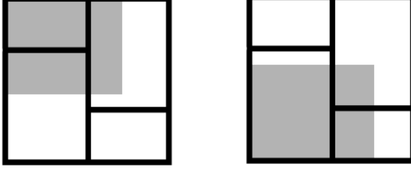

Fig. 3: Samples of two dimensions both has one way limitation, 3 regions have intersection.

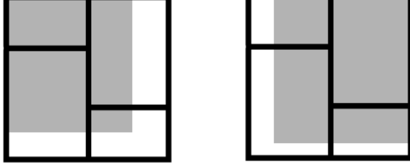

Fig. 4: Samples of two dimensions both has one way limitation, 4 regions all have intersection.

The difference between subcase 1 and subcase 2 is that, one more region satisfy case 1. If one region is fully covered, like the subpicture of left one in Fig. 3, has the same time complexity as subcase 1. To calcuate the right case in Fig.3, the $T_2(n)$ adds a new condition baseed on subcase 1:

$$T_2(n) = \begin{cases} O(1), & \text{if } n \ = \ 1 \\ T_2(\frac{n}{4}) + 2T_1(\frac{n}{4}) + O(1), & \text{subcase 2.} \end{cases} \tag{7}$$

Then we discuss the subcase that 4 regions all have intersection. The possible figure is in Fig. 4.

The $T_2(n)$ can be easily get. It's exactly the same as subcase 2. We don't detaily discuss only one resion has intersection because it's ordinary. Comprehensively, the final expression is like below.

$$T_2(n) = \begin{cases} O(1), & \text{if } n \ = \ 1 \\ T_2(\frac{n}{4}) + O(1), & \text{one resion intersects.} \\ T_2(\frac{n}{4}) + T_1(\frac{n}{4}) + O(1), & \text{subcase 1.} \\ T_2(\frac{n}{4}) + 2T_1(\frac{n}{4}) + O(1), & \text{subcase 2 and 3.} \end{cases} \tag{8}$$

Because of $T_1(n) = O(\sqrt{n})$ has been proved before, we replace it and simplify the formula. Then we get:

$$T_2(n) = \begin{cases} O(1), & \text{if } n \ = \ 1 \\ T_2(\frac{n}{4}) + O(\sqrt{n}), & \text{otherwise.} \end{cases} \tag{9}$$

The reason why we get this is because of asymptotic expression's properties. $O(\sqrt{n})$ always dominates $O(1)$, so that all subcases finally integrates into one expression. The solution to it is $T_2(n) = O(\sqrt{n})$ by using master therom.

3. Two dimensions both has no special limitation.

We still prove it from the aspect of number of regions that intersect. Let $T(n)$ be the time complexity of range query.

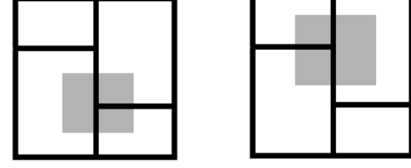

Fig. 5: Samples of 3 regions have intersection.

If number of intersections less than 3, we immediately get:

$$T(n) = \begin{cases} O(1), & \text{if } n \ = \ 1 \\ T(\frac{n}{4}) + O(1), & \text{only 1 region intersects.} \\ 2T(\frac{n}{4}) + O(1), & \text{2 region intersects.} \end{cases} \tag{10}$$

When there are exactly 3 regions have intersection, the figure is nothing else than it shows in Fig. 5.

In Fig. 5, there are exactly 2 regions satisfy case 2. The last region is a sub problem, which can be described by recursion.

When all 4 regions have intersection, there is only one case. It is showed in Fig. 6.

All things considered, the $T(n)$'s recursive formula is:

$$T(n) = \begin{cases} O(1), & \text{if } n \ = \ 1 \\ T(\frac{n}{4}) + O(1), & \text{only 1 intersects.} \\ 2T(\frac{n}{4}) + O(1), & \text{2 region intersects.} \\ T(\frac{n}{4}) + 2T_2(\frac{n}{4}) + O(1), & \text{3 intersects.} \\ 4T_2(\frac{n}{4}) + O(1), & \text{all 4 intersects.} \end{cases} \tag{11}$$

Simplify it with the same method as we use in case 2:

$$T(n) = \begin{cases} O(1), & \text{if } n \ = \ 1 \\ T(\frac{n}{4}) + O(1), & \text{only 1 region intersects.} \\ 2T(\frac{n}{4}) + O(1), & \text{2 region intersects.} \\ T(\frac{n}{4}) + O(\sqrt{n}), & \text{3 intersects.} \\ O(\sqrt{n}), & \text{all 4 intersects.} \end{cases} \tag{12}$$

Then we demonstrate the initial therom: Range query's complexity is $O(\sqrt{n})$.

If we assume that in the resursion, each $O(f(n))$ satiscify:

$$\begin{cases} 0 < O(1) < c_1 \\ 0 < O(\sqrt{n}) < c_2\sqrt{n} \\ 0 < T(n) \leq c_3\sqrt{n} - c_4 \end{cases} \tag{13}$$

According to properties of asymptotic notation, we just need to scale each inequation and find the solution to parameter c_3 c_4:

$$\begin{cases} c_3 - c_4 \geq c_1 \\ c_3\sqrt{n} - c_4 \geq \frac{1}{2}c_3\sqrt{n} - c_4 + c_1 \\ c_3\sqrt{n} - c_4 \geq c_3\sqrt{n} - 2c_4 + c_1 \\ c_3\sqrt{n} - c_4 \geq \frac{1}{2}c_3\sqrt{n} - c_4 + c_2\sqrt{n} \\ c_3\sqrt{n} - c_4 \geq c_2\sqrt{n} \end{cases} \tag{14}$$

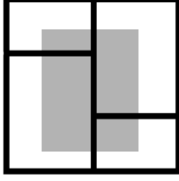

Fig. 6: Samples of all 4 regions have intersection.

Notice that n can be regarded as a very large number. The implicit condition in this is $c_1, c_2 > 0$. So we just select a simple but correct solution: $c_3 = 2c_2 + 2c_1, c_4 = 2c_1$, which can be easily verified.

D. Delete a single point

After the analysis of range query, we are intended to maintain a dynamic point set. Delete operation's task is easy: in a given set of points, delete an existing point. This point is uniquely specified by its Euclidean space coordinates.

Although the form of KD-Tree is a binary tree, but it is different from the binary search tree, there is no fixed keyword, so that the order of the traversal according to the keyword ranking increments, so it's not sensible to use the binary search tree deletion technique directly.

Here static deletion is used to maintain information, also known as lazy deletion, that is, each KD-Tree node maintains a Boolean type tag(in pseudo code, it's 'exist' parameter), indicating whether the point in the space represented by this node has been deleted, if it has not been deleted, its value is true, otherwise it is false.

Since the essence of the KD-Tree is that the node of each tree represents a hyperplane partition in space, we compare the position of the hyperplane and the insertion point for each point to determine which subtree to recursively enter.

Algorithm 4 Delete

Input: A KD-Tree and a point's coordinate
Output: None

```
DELETE(Node, Point, Dimension)
    if (Node is Null) then
        return
    end if
    if (Node.Point equals Point) then
        Node.exist ← False
        return
    end if
    Next ← x if Dimension is y else y
    if (Node.point ≤ Point on Dimension) then
        DELETE(Node.LeftSon, Point, Next)
    else
        DELETE(Node.RightSon, Point, Next)
    end if
```

Obviously, the complexity is proportional to the depth of recursion, which is not higher than the height of the tree. According to therom 1, in a balanced KD-Tree with n nodes, the height is $\Theta(\log n)$.

Since a large number of nodes that have been deleted will still occupy space in the KD-Tree, a garbage node recycling should be carried out at an appropriate time, the specific method will be mentioned in the following.

E. Insert a single point

The task of point insertion is to insert a new point into a given set of points.

In the point deletion section, we mentioned that KD-Tree is not exactly equivalent to binary search trees, so the insertion of KD-Tree cannot exactly copy the algorithm of binary search trees.

Like the way of delete a node, if we meet a null node when recursing, just attach a new node with information initialize by the point's coordinate to it.

The above statement is actually a simple binary search tree insertion method, without maintenance. So in extreme cases the tree height will reach $O(n)$, which will have a serious performance impact on the nearest neighbor search and stack space in the actual recursion. So we need to find a feasible means to maintain the height of the tree.

Here, we introduce the scapegoat tree maintenance method: Periodically refactoring maintenance complexity. If a node's subtree is out of balance too much, we simply rebuild this subtree. The binary tree is now reduced to a balanced binary tree with a balance coefficient α.

Defination 3: α-balanced binary tree

For each node in a α-balanced binary tree, it must satisfy: both son node's subtree size not greater than that node's subtree size $\times\alpha$. Obviously $\alpha \in (0.5, 1)$.

Rebuild the imbalanced subtree just need to traverse the subtree and get all point's coordinate, then execute build. During traverse, nodes with exist tag equals false will be recycled. After rebuild, the whole tree is still α-balanced.

Algorithm 5 Insert

Input: A KD-Tree and a point's coordinate
Output: None

```
INSERT(Node, Point, Dimension)
    if (Node is Null) then
        Initialize by using Point's infomation
        return
    end if
    Next ← x if Dimension is y else y
    if (Node.point ≤ Point on Dimension) then
        INSERT(Node.LeftSon, Point, Next)
    else
        INSERT(Node.RightSon, Point, Next)
    end if
    Maintain balance infomation from subtree
    if (Node's subtree is not balance) then
        Rebuild Node's subtree
        return
    end if
    Maintain infomation from subtree
```

Theorem 4: An α-balanced binary tree's height: A α-balanced binary tree's height is $O(\log n)$.

Proof: We consider a recursion on binary tree. In each step, the node's subtree's size multiplies a constant between $(1-\alpha, \alpha)$, the size becomes to 1 in no more than $-\log_\alpha n$ steps, which means it's a leaf node. The height is no more than $-\log_\alpha n$, which is $O(\log n)$.

Theorem 5: Insertion with rebuild on a α-balanced KD-Tree's average cost: A α-balanced KD-Tree's insertion with rebuild is $O((\log n)^2)$.

Proof: Define a node's potential function:

$$\Delta(node) = |LeftSon.size - RightSon.size| \quad (15)$$

$$\phi(node) = c(2\alpha - 1)\Delta(node)\log node.size \quad (16)$$

And the whole binary tree(described as 'T')'s potential function is just to sum them up, which means $\Phi(T) = \sum_{node \in T} \phi(node)$.

If we just insert the node into the tree, the complexity is no more than height of this tree, which is $O(\log n)$ Consider the insertion operation's effect on T's potential function. Insert a node will increase $\Phi(T)$ at most $height \times \sum_{node \in path} \Delta\phi(node)$. We apply scaling on the expression of potential function, in which we use $\log n$ to replace $\log size$ to make the upper bound more loose. Then $\Delta\phi(node) < \log n$ is get easily. Therefore, the increment in $\Phi(T)$ is less than $height \times O(\log n)$, which is $O((\log n)^2)$.

The maintaince of a KD-Tree uses potential energy stored in potential function to cover the time cost. If a node's subtree is rebuilt, the potential function's value will decrease at least $2(\alpha - 1) size \log size$. And because rebuild's time complexity is $size \log size$, we just need to adjust constant c so that $\Delta\Phi(T)$ is able to cover the cost of rebuild.

Above all, we have proved that insertion with rebuild's average cost is $O((\log n)^2)$.

IV. The influence of dimension

Operations' complexity are not relevant to dimension except range query is easily got after proofs above. We focus on dimension's influence on range query's complexity.

In order to prove range query's complexity in 2-d, 4 regions are showd in each figure. Why we choose 4 as the number of regions? It's because after 2 rounds' division, the standard of division in each round is the same as 2 rounds before, which forms the structure of resursion.

Like our analysis of complexity, in k-d space, 2^k regions are needed to be considerd. We discuss the case that if a dimension has limitation, and in which it's one or both ends. The final conclusion is that in k-d space, range query on KD-Tree's complexity is $O(n^{1-\frac{1}{k}})$ [5]. In 2-d Euclidean space, we have proved it to be true. In high-dimemsion's space, our jub just provided a clear and feasible method. Readers can try it if interested.

V. Conclusion

Each operation's complexity on KD-Tree are listed in tabel 1. One thing to note is that operations except insert and delete are based on a strictly balanced KD-tree.

TABLE I: **Operation's complexity on KD-Tree**

Operation	Complexity	Method of analysis
Build tree	$\Theta(n \log n)$	Master theorem
NN Query	$O(n)$ (worst case)	
Range Query	$O(\sqrt{n})$	Master theorem
Delete	$O(\log n)$	
Insert with Rebuild	$\emptyset((\log n)^2)$ (average)	Potential energy analysis

When we extend the dimension to k, then the range query's complexity will change to $O(n^{1-\frac{1}{k}})$.

Acknowledgments

This paper not only reflects my years of learning and research results, but also more condensed the support and help of family, teachers, classmates and friends. Here I would like to express my heartfelt thanks to them.

References

[1] Chandran, Sharat. Introduction to kd-trees. University of Maryland Department of Computer Science.
[2] Thomas H. Cormen, Charles E.Leiserson, Ronald L.Rivest, Clifford Stein. *Introduction to Algorithms*. 3rd Edition. China Machine Press.
[3] Bentley, J. L. Multidimensional binary search trees used for associative searching. Communications of the ACM, vol. 18, no.9, pp. 509-517. 1975.
[4] Rosenberg, J. B. Geographical data structures compared: A study of data structures supporting region queries. *IEEE transactions on computer-aided design of integrated circuits and systems*, vol. 4, no. 1, pp. 53-67. 1985.
[5] Lee, D. T., & Wong, C. K. Worst-case analysis for region and partial region searches in multidimensional binary search trees and balanced quad trees. *Acta Informatica*, vol. 9, no. 1, pp. 23-29. 1977.
[6] Wald, I., & Havran, V. On building fast kd-trees for ray tracing, and on doing that in $O(N \log N)$. In 2006 *IEEE Symposium on Interactive Ray Tracing* pp. 61-69. Sept, 2006.
[7] Hoare, C. A. R. (1961). "Algorithm 65: Find". Comm. ACM. vol. 4, no. 7, pp. 321–322. doi:10.1145/366622.366647.