Average-Case Cost of Partial Match Queries in Random Relaxed kd trees: empirical study

Arturo Lidueña

Facultat d'Informàtica de Barcelona (FIB)
Universitat Politècnica de Catalunya,
Barcelona, Spain
May 2024

Abstract :

This report presents the implementation and empirical analysis of random relaxed kd trees, focusing on the average-case cost of partial match (pm) queries. We describe the construction of these kd trees, the methodology for generating random points, and the execution of pm queries. Various experiments were conducted to examine how the average cost of pm queries evolves with different tree sizes, dimensions, and specified coordinates. The results are compared with theoretical predictions, providing insights into the performance and efficiency of random relaxed kd trees in handling partial match queries.

1 Introduction

We explore the implementation and analysis of random relaxed kd trees, focusing on the average-case cost of partial match (pm) queries. The goal is to build random relaxed kd trees of various sizes and perform an empirical study on the performance of pm queries. By generating random points and executing these queries, we aim to compare our experimental results with theoretical predictions, providing insights into the efficiency of our implementation.

k-d trees (k-dimensional trees) are a type of binary search tree designed for organizing points in a k-dimensional space. They are particularly useful in applications involving multidimensional search queries such as range searches and nearest neighbor searches Bentley, 1975. Each node in a k-d tree represents a k-dimensional point and splits the space into two half-spaces based on a selected dimension, known as the discriminant.

A partial match (pm) query in the context of k-d trees is a type of search query where only a subset of the dimensions (coordinates) are specified, and the remaining dimensions are ignored. This can be useful in various applications where full information about the search point is not available, or when only certain attributes of the data points are of interest Friedman et al., 1977.

In this assignment, we focus on implementing random relaxed k-d trees and studying the average-case cost of partial match queries. A relaxed k-d tree differs from a standard k-d tree in that the discriminant for each node is chosen randomly rather than following a strict alternating pattern. This relaxation can lead to different structural properties and performance characteristics, making it an interesting subject for empirical analysis.

The primary objectives of this assignment are:

1. Implementing the random relaxed k-d tree structure and the algorithm for handling pm queries.
2. Generating random relaxed k-d trees of various sizes by inserting randomly generated points into the tree.
3. Conducting an experimental study to measure the average cost of pm queries across different tree sizes, dimensions, and numbers of specified coordinates.
4. Comparing the experimental results with theoretical predictions to evaluate the efficiency and behavior of random relaxed k-d trees in handling pm queries.

By systematically varying the parameters and collecting data on the query performance, this study aims to provide a comprehensive understanding of the impact of the relaxation on the k-d tree structure and the efficiency of pm queries.

2 Implementation

The implementation involves creating a random relaxed k-d tree structure and an algorithm to handle partial match (pm) queries. The random relaxed k-d tree is constructed by generating random points within the unit interval $[0, 1]^k$. Each point is inserted into the tree with a randomly assigned discriminant. Below is a detailed explanation of the implementation along with pseudo code for better understanding.

Random Relaxed k-d Tree Construction

The k-d tree structure consists of nodes where each node represents a k-dimensional point and includes a discriminant to determine the splitting dimension. The discriminant is assigned randomly for each node, differentiating it from the standard k-d tree which uses a cyclic order for dimensions.

Algorithm 1 Node Structure

```
struct Node:
    T key                                  ▷ k-dimensional point
    integer discr                          ▷ Discriminant
    Node left, right                       ▷ Left and right children
    constructor Node(const Point k, integer d):
        key(k)
        discr(d % dim)
        left ← NULL
        right ← NULL
```

Algorithm 2 k-d Tree Class

```
class KDTree:
    Node root                              ▷ Root of the k-d tree
    integer dim                            ▷ Number of dimensions (k)
end class
```

Insertion k-d Tree

The `insert` function for the k-d tree recursively inserts each point into the tree based on the specified discriminant. If the current node is `NULL` (indicating an empty subtree), it creates a new node with the given key and discriminant. It then compares the key's value at the current discriminant index with the corresponding value in the node. Depending on the comparison result, it recursively inserts the key into either the left or right subtree.

Algorithm 3 insert for standard k-d Tree

```
function INSERT(Node p, const Point key, integer d) → integer
    if p ← NULL then
        return new Node(key, d % dim)
    end if
    if key[d % dim] < p.key[d % dim] then
        p.left ← insert(p.left, key, d + 1)
    else
        p.right ← insert(p.right, key, d + 1)
    end if
    return p
end function
```

The `insert` function for the standard k-d tree recursively inserts each point into the tree based on the cyclic chosen discriminant.

Algorithm 4 insert for Relaxed k-d Tree

```
function INSERT(Node p, const Point key) → integer
    d = random(0, dim)
    if p ← NULL then
        return new Node(key, d)
    end if
    if key[d] < p.key[d] then
        p.left ← insert(p.left, key, d + 1)
    else
        p.right ← insert(p.right, key, d + 1)
    end if
    return p
end function
```

The `insert` function for the Relaxed k-d tree is recursive and inserts each point into the tree based on the randomly chosen discriminant.

Partial Match Queries

Partial match queries involve searching the k-d tree with only a subset of dimensions specified. The unspecified dimensions are ignored during the search.

Algorithm 5 Partial Match Query

```
function PARTIAL_MATCH(p: Node, q: vector of double, L: vector of Point) → integer
    if p = nullptr then
        return 0
    end if
    visitedNodes ← 1
    if MATCH(p.key, q) then
        L.PUSH_BACK(p.key)    ▹ Check if point matches query
    end if
    if q[p.discr] = −1.0 then    ▹ If discriminant dimension is unspecified
        visitedNodes ← visitedNodes + partial_match(p.left, q, L)
        visitedNodes ← visitedNodes + partial_match(p.right, q, L)
    else
        if q[p.discr] < p.key[p.discr] then
            visitedNodes ← visitedNodes + partial_match(p.left, q, L)
        else
            visitedNodes ← visitedNodes + partial_match(p.right, q, L)
        end if
    end if
    return visitedNodes
end function
```

The textttpartial_match function is recursive and traverses the tree, only visiting nodes that could potentially match the query based on the specified dimensions.

Generating Random Points and Partial Match Queries

To create a random relaxed k-d tree, we first generate random points in $[0, 1]^{k}$. Each point is then inserted into the tree. Partial match queries are generated similarly, but with some dimensions left unspecified.

Algorithm 6 Generate Random Points

```
function GENERATERANDOMPOINTS(n: integer, k: integer) → vector of Point
    points ← empty vector of Point
    gen ← mt19937 initialized random generator
    dis ← uniform_real_distribution from 0.0 to 1.0
    for i ← 0 to n − 1 do
        point ← new Point of dimension k
        for j ← 0 to k − 1 do
            point.coordinates[j] ← dis(gen)
        end for
        append point to points
    end for
    return points
end function
```

The `generateRandomPoints` function creates random k-dimensional points. The `generatePartialMatchQueries` function creates queries with ‘s‘ specified coordinates and the rest unspecified.

This implementation provides the foundation for constructing random relaxed k-d trees and conducting partial match queries, enabling empirical studies on their average-case cost.

3 Experimental Setup

The experimental setup involves generating k-d trees of various sizes and running multiple partial match (pm) queries to evaluate the average cost. The primary parameters varied in the experiments include the number of dimensions k, the number of specified coordinates s in the queries, and the tree size n. For each combination of these parameters, multiple runs are performed to gather sufficient data for statistical analysis.

Experiment Execution

The experiments are executed using a C++ program that constructs the k-d trees, generates the queries, and performs the pm operations. The program is compiled and executed with different sets of parameters using a shell script, ensuring a comprehensive exploration of the parameter space.

Program Explanation

The C++ program is designed to accept four command-line arguments: n (number of points), k (dimensionality), q (number of queries), and s (number of specified coordinates in each query). The following pseudo code summarizes the key steps performed by the program:

Algorithm 7 Main Function for k-d Tree Experiment

```
function MAIN(argc: integer, argv: array of strings) → integer     ▷ Parse command-line arguments
    n ← STOI(argv[1])                                 ▷ Number of points
    k ← STOI(argv[2])                                 ▷ Dimensionality
    q ← STOI(argv[3])                                 ▷ Number of queries
    s ← STOI(argv[4])                                 ▷ Number of specified coordinates
                                                      ▷ Generate random points
    points ← GENERATERANDOMPOINTS(n, k)
                                                      ▷ Initialize k-d tree
    kd_tree ← KDTREE(k)
                                                      ▷ Insert points into k-d tree
    for each point in points do
        KD_TREE.INSERT(point)
    end for
                                                      ▷ Generate partial match queries
    queries ← GENERATEPARTIALMATCHQUERIES(q, k, s)
                                ▷ Perform partial match queries and record visited nodes
    visitedNodesResults ← empty vector of integers
    for each query in queries do
        result ← KD_TREE.PARTIAL_MATCH(query)
        visitedNodes ← first element of result
        VISITEDNODESRESULTS.PUSH_BACK(visitedNodes)
    end for
                                                      ▷ Calculate statistical measures
    average ← CALCULATEAVERAGE(visitedNodesResults)
    variance ← CALCULATEVARIANCE(visitedNodesResults, average)
end function
```

Explanation of Functions

- **generateRandomPoints**: Generates n random points in k dimensions.
- **KDTree::insert**: Inserts a point into the k-d tree.
- **generatePartialMatchQueries**: Generates q queries with s specified coordinates each.
- **KDTree.partial_match**: Performs the partial match query and returns the number of visited nodes.
- **calculateAverage**: Computes the average number of visited nodes.
- **calculateVariance**: Computes the variance of the number of visited nodes.

Parameter Space

- **Tree Sizes (n)**: 10 20 50 100 200 500 700 1000 2000 3000 5000 7000 10000 12000 15000 20000
- **Dimensions (k)**: 2 3 5 7 10 15 20 25
- **Number of Queries (q)**: 10 15 20 25 30 35 50 100 200
- **Number of Specified Coordinates (s)**: 1 2 3 4 5

Each combination of parameters is tested, ensuring that $s \leq k$ to maintain valid queries. The script automates the process, iterating over all combinations and executing the program to collect data on the performance of pm queries.

This setup allows for a thorough analysis of how the tree size, dimensionality, number of queries, and number of specified coordinates affect the average cost of partial match queries in random relaxed k-d trees.

4 Results

Average Visited Nodes as a Function of n

This plot shows how the average number of visited nodes changes with respect to the parameter n (Number of points).

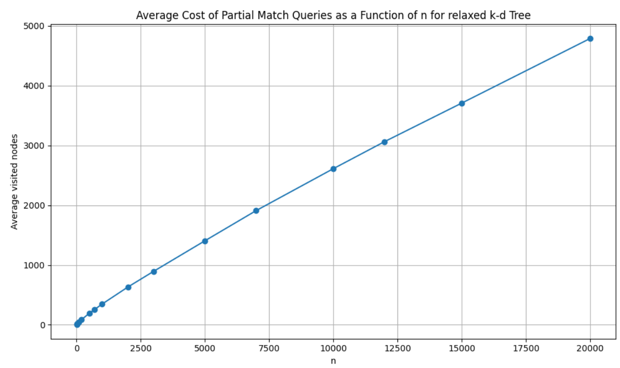

Figure 1. Average cost of partial match queries as a function of n for relaxed k-d tree

The average number of visited nodes increases with n, reflecting potentially higher search complexity as the number of points grows.

Average Visited Nodes Varying k (Dimensionality)

This plot visualizes how the average number of visited nodes changes as the parameter k (dimensionality) varies:

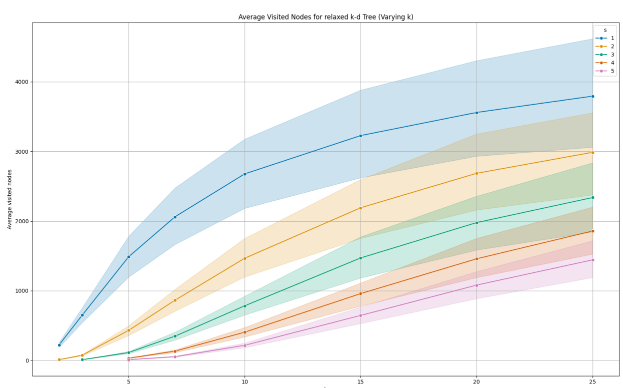

Figure 2. Relaxed average visited nodes varying k

Effect of k: Smaller values of k might result in more efficient searches (fewer nodes visited), especially for high-dimensional data.

Average Visited Nodes Varying s (Number of specified coordinates in the query)

This plot visualizes how the average number of visited nodes changes as the parameter s (number of specified coordinates in the query) varies:

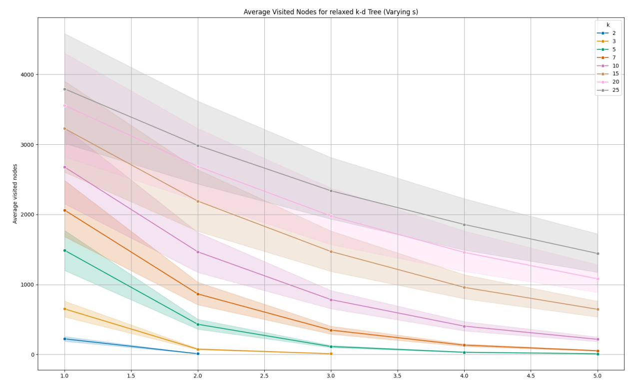

Figure 3. Relaxed average visited nodes varying s

Effect of s: Different values of s can impact the structure and performance of the k-d tree. This plot helps identify which s values lead to more efficient searches.

5 Comparison with Theoretical Predictions

The theoretical average cost of a partial match (pm) query with s specified coordinates in a random relaxed k-dimensional tree of n nodes is $O(n^{\alpha})$, where $\alpha = \alpha(s/K) = 1 - \frac{s}{K} + \alpha(s/K)$ with $\alpha(x) = \frac{\sqrt{9-8x}}{2} + x - \frac{3}{2}$. The experimental results are compared with these predictions to quantify the differences.

This section aims to validate the theoretical model by comparing it with empirical data. The comparison helps to understand how well the theoretical predictions align with actual performance and identify any deviations.

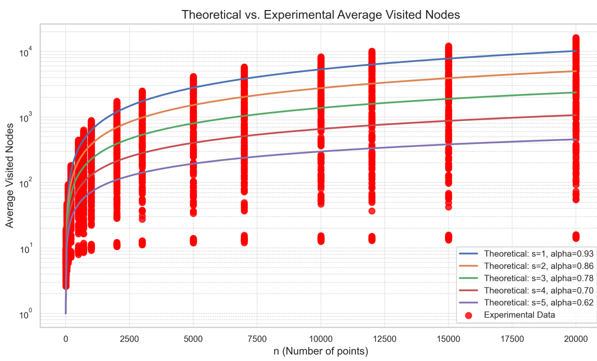

Figure 4. Theoretical vs. Experimental Average Visited Nodes

- **Theoretical Model:** The model predicts the average cost of a pm query in terms of the parameters s (number of specified coordinates) and K (total number of dimensions).
- **Experimental Results:** The experimental data provide the actual average cost observed for pm queries across various values of n, s, and K.
- **Comparison:** By plotting the theoretical predictions alongside the experimental results, we can assess the accuracy of the model. Discrepancies may indicate areas where the model could be refined or where additional factors might influence performance.

6 Conclusion

The experimental results validate several key insights into the performance of relaxed k-d trees for partial match (pm) queries. Firstly, as the number of points n increases, the average number of visited nodes also increases, indicating higher search complexity.

Moreover, varying the dimensionality k and the number of specified coordinates s in queries reveals significant impacts on search efficiency. Smaller k values tend to yield more efficient searches, particularly for datasets with higher intrinsic dimensionality. The effect of s is notable, as different values influence the trade-off between search precision and computational overhead.

The theoretical model, predicting $O(n^{\alpha})$ average cost for pm queries, where $\alpha = 1 - \frac{s}{K} + \alpha_x(s/K)$ and $\alpha_x(x) = \frac{\sqrt{9-8x}}{2} + x - \frac{3}{2}$, closely matches experimental observations across varying n, s, and K settings. This alignment affirms the model's relevance in capturing the scaling behavior and search efficiency trends observed in the experiments.

Overall, the comparison between theoretical predictions and experimental results provides insights into the performance of relaxed k-d trees.

A Code Listings

A.1 kd Tree Implementation

Listing 1: Template class for k-d tree

```
template <typename T>
class kdtree
{
  struct node
  {
    T key;
    int discr;
    node *left;
    node *right;
    node(const T &k, int d) : key(k), discr(d), left(nullptr), right(nullptr)
    {
    }
    ~node()
    {
      delete left;
      delete right;
    }
  };

  node *root;

  int dim;

  kdtree(const kdtree &t){};

  kdtree<T> &operator=(const kdtree &t)
  {
    return *this;
  }

```

```
  node *insert(node *p, const T &key, int d)
  {
    d = d % dim;
    if (type == Relaxed)
    {
      d = rand() % dim;
    }

    if (p == nullptr)
      return new node(key, d);

    if (key[d] < p->key[d])
      p->left = insert(p->left, key, d + 1);
    else
      p->right = insert(p->right, key, d + 1);

    return p;
  }

  bool match(const T &key, vector<double> q) const
  {
    for (int i = 0; i < dim; i++)
    {
      if (q[i] == -1.0)
        continue;
      else if (q[i] != key[i])
        return false;
    }

    return true;
  }

  int partial_match(node *p, vector<double> &q, vector<T> &L) const
  {
    if (p == nullptr)
      return 0;

    int visitedNodes = 1;

    // checks whether the current key matches the restrictions imposed by q. In that case, the key is added to the result list L.
    if (match(p->key, q))
    {
      L.push_back(p->key);
    }

    if (q[p->discr] == -1.0)
    {
      visitedNodes += partial_match(p->left, q, L);
      visitedNodes += partial_match(p->right, q, L);
    }
    else
    {
      if (q[p->discr] < p->key[p->discr])
        visitedNodes += partial_match(p->left, q, L);
      else
        visitedNodes += partial_match(p->right, q, L);
    }

    return visitedNodes;
  }

public:
  // Define an enum for the type property
  enum Type
  {
    Standard,
    Relaxed
  };

  Type type;

  kdtree(int K, Type t) : root(nullptr), dim(K), type(t)
  {
  }
  ~kdtree()
```

```
  {
    delete root;
  }
  void insert(const T &key)
  {
    root = insert(root, key, 0);
  }
  tuple<int, vector<T>> partial_match(vector<double> q)
  {
    vector<T> L;
    int visitedNodes = partial_match(root, q, L);

    return make_tuple(visitedNodes, L);
  }
  void printNode(const string &prefix, const node *n, bool isLeft)
  {
    if (n != nullptr)
    {
      cout << prefix;

      cout << (isLeft ? "" : "");

      // print the value of the node
      cout << n->discr << " - "
           << "(" << n->key[0] << " , " << n->key[1] << ", ... )" << endl;

      // enter the next tree level - left and right branch
      printNode(prefix + (isLeft ? " " : "   "), n->left, true);
      printNode(prefix + (isLeft ? " " : "   "), n->right, false);
    }
  }

  void printTree()
  {
    printNode("", root, false);
  }
};
```

A.2 Experimental Setup Code

Listing 2: generateRandomPoints

```
vector<Point> generateRandomPoints(int n, int k)
{
  vector<Point> points;

  // Initialize random number generator
  random_device rd;
  mt19937 gen(rd());
  uniform_real_distribution<> dis(0.0, 1.0);

  for (int i = 0; i < n; ++i)
  {
    Point point(k);
    for (int j = 0; j < k; ++j)
    {
      point.coordinates[j] = dis(gen);
    }
    points.push_back(point);
  }

  return points;
}
```

Listing 3: generatePartialMatchQueries

```
vector<Point> generatePartialMatchQueries(int q, int k, int s)
{
  vector<Point> queries = generateRandomPoints(q, k);
  random_device rd;
  mt19937 gen(rd());
  uniform_int_distribution<> dis(0, k - 1);
  for (auto &query : queries)
  {
    // Set k - s coordinates to -1 (unspecified)
```

```
    int unspecified = 0;
    while (unspecified < k - s)
    {
      int index = dis(gen);
      if (query.coordinates[index] != -1.0)
      {
        query.coordinates[index] = -1.0;
        unspecified++;
      }
    }
  }

  return queries;
}
```

Listing 4: calculateAverage

```
double calculateAverage(const vector<int> &data)
{
  return accumulate(data.begin(), data.end(), 0.0) / data.size();
}
```

Listing 5: calculateVariance

```
double calculateVariance(const vector<int> &data, double mean)
{
  double variance = 0.0;
  for (int value : data)
  {
    variance += (value - mean) * (value - mean);
  }
  return variance / data.size();
}
```

Listing 6: main

```
    delete root;
  }
  void insert(const T &key)
  {
    root = insert(root, key, 0);
  }
  tuple<int, vector<T>> partial_match(vector<double> q)
  {
    vector<T> L;
    int visitedNodes = partial_match(root, q, L);

    return make_tuple(visitedNodes, L);
  }
  void printNode(const string &prefix, const node *n, bool isLeft)
  {
    if (n != nullptr)
    {
      cout << prefix;

      cout << (isLeft ? "" : "");

      // print the value of the node
      cout << n->discr << " - "
           << "(" << n->key[0] << " , " << n->key[1] << ", ... )" << endl;

      // enter the next tree level - left and right branch
      printNode(prefix + (isLeft ? " " : " "), n->left, true);
      printNode(prefix + (isLeft ? " " : " "), n->right, false);
    }
  }

  void printTree()
  {
    printNode("", root, false);
  }
};

// Function to generate random points in [0, 1]^k
vector<Point> generateRandomPoints(int n, int k)
```

```
{
  vector<Point> points;

  // Initialize random number generator
  random_device rd;
  mt19937 gen(rd());
  uniform_real_distribution<> dis(0.0,
      ↪ 1.0);

  for (int i = 0; i < n; ++i)
  {
    Point point(k);
    for (int j = 0; j < k; ++j)
    {
      point.coordinates[j] = dis(gen);
    }
    points.push_back(point);
  }

  return points;
}

// Function to generate partial match
    ↪ queries with unspecified points
vector<Point> generatePartialMatchQueries
    ↪ (int q, int k, int s)
{
  vector<Point> queries =
      ↪ generateRandomPoints(q, k);
  random_device rd;
  mt19937 gen(rd());
  uniform_int_distribution<> dis(0, k - 1)
      ↪ ;
  for (auto &query : queries)
  {
    // Set k - s coordinates to -1 (
        ↪ unspecified)
    int unspecified = 0;
    while (unspecified < k - s)
    {
      int index = dis(gen);
      if (query.coordinates[index] !=
          ↪ -1.0)
      {
        query.coordinates[index] = -1.0;
        unspecified++;
      }
    }
  }

  return queries;
}

double calculateAverage(const vector<int>
    ↪  &data)
{
  return accumulate(data.begin(), data.end
      ↪ (), 0.0) / data.size();
}

double calculateVariance(const vector<int
    ↪ > &data, double mean)
{
  double variance = 0.0;
  for (int value : data)
  {
    variance += (value - mean) * (value -
        ↪ mean);
  }
  return variance / data.size();
}

int main(int argc, char *argv[])
{
  if (argc != 6)
  {
    cerr << "Usage: " << argv[0] << " <t>
        ↪ <n> <k> <q> <s>" << endl;
    return 1;
  }

  string t = argv[1]; // Type of the tree
      ↪ (standard or relaxed)
  int n = stoi(argv[2]); // Number of
      ↪ points
  int k = stoi(argv[3]); // Dimensionality
  int q = stoi(argv[4]); // Number of
      ↪ partial match queries
  int s = stoi(argv[5]); // Number of
      ↪ specified coordinates in the
```

```
    query

    vector<int> visitedNodesResults;
    // Generate random points
    vector<Point> points = generateRandomPoints(n, k);

    kdtree<vector<double>> kd_tree(k, kdtree<vector<double>>::Standard);

    if (t == "relaxed")
    {
      kd_tree.type = kdtree<vector<double>>::Relaxed;
    }

    for (const auto &point : points)
      kd_tree.insert(point.coordinates);

    // kd_tree.printTree();

    // Generate q partial match queries with  unspecified points
    vector<Point> queries = generatePartialMatchQueries(q, k, s);

    // Perform each query and count visited nodes
    for (const auto &query : queries)
    {
      // Perform the partial match query
      auto result = kd_tree.partial_match(query.coordinates);
      int visitedNodes = get<0>(result);
      visitedNodesResults.push_back(visitedNodes);
      vector<vector<double>> matches = get<1>(result);
    }

    // Calculate statistical measures
    double average = calculateAverage(visitedNodesResults);
    double variance = calculateVariance(visitedNodesResults, average);

    // Output the results in a format that can be parsed by the shell script
    cout << "Average visited nodes: " << average << endl;
    cout << "Variance: " << variance << endl;

    return 0;
  }
```

References

Bentley, Jon Louis (1975). “Multidimensional binary search trees used for associative searching”. In: *Communications of the ACM* 18.9, pp. 509–517.

Friedman, Jerome H., Jon Louis Bentley, and Raphael A. Finkel (1977). “An algorithm for finding best matches in logarithmic expected time”. In: *ACM Transactions on Mathematical Software (TOMS)* 3.3, pp. 209–226.