Course: Complex Data Managment

Project: Quad Trees vs R-Trees

Name: Konstantinos Vardakas

Student ID: 522

Email: pcs0522@uoi.gr

Code Structure

Files Point.h and Point.cpp

The Point class is the basic data structure for representing points in two-dimensional space and is implemented in the Point.h (header file) and Point.cpp (implementation file) files. It includes two float variables, x and y, which express the coordinates of the point on the plane.

The class provides two basic functions for calculating distances. The distance_to_point() function calculates the Euclidean distance of the point from another point and is used in the k-NN query algorithm for both QuadTrees and R-Trees, applying the Pythagorean theorem formula:

$$distance = \sqrt{(x_2 - x_2)^2 + (y_1 + y_2)^2}$$

The distance_to_rectangle() function calculates the minimum Euclidean distance of the point from a rectangle and is used for the corresponding range queries of the two structures. If the point is within the boundaries of the rectangle, the distance is zero. Otherwise, the function calculates the distance from the nearest edge or corner of the rectangle and returns the corresponding Euclidean value.

Finally, the output operator << is overloaded so that an object of type Point can be printed directly in the form Point(x, y).

Files Rectangle.h and Rectangle.cpp

The Rectangle class, implemented in the Rectangle.h and Rectangle.cpp files, is a basic structure for representing rectangular areas in two-dimensional space. It is used both for implementing range queries and for defining the boundaries of the space occupied by each node in tree structures such as R-Tree and QuadTree. Each object of the class is described by the coordinates of its center (x, y), as well as its width (w) and height (h). These values are used to calculate and store the edges of the parallelogram: left, right, top, and bottom, in order to facilitate the execution of spatial checks.

The class includes three basic methods. The contains() function checks whether a point is within the boundaries of the rectangle, and is used either when inserting a new point into a node (in the case of QuadTree), or when checking the participation of a point in a given range query. The intersects() function examines whether two rectangles overlap and is again used in range queries to omit areas without overlap, thereby reducing computational cost. Finally, the union_with() method of the Rectangle class calculates the minimum rectangle that contains both the original rectangle (this) and a second rectangle given as a parameter. It returns a new Rectangle that fully encompasses both Rectangles and is used to calculate the MBRs of R-trees.

In addition, there is an overload of the operator << so that objects of the Rectangle class can be printed.

File HeapEntry.h

The HeapEntry.h file defines a generic (template) HeapEntry data structure, which is used to store and manage entries in a priority queue. This structure is parameterized with respect to the NodeType node type, so that it can be used for both QuadTree and RTree.

Each HeapEntry contains three variables. First, it contains the distance calculated either by the distance_to_point function or by the distance_to_rectangle function. This distance is used to compare the elements in the priority queue, which is implemented in the k-NN queries of each tree structure. The < operator is overloaded so that the elements with the smallest distance have the highest priority. In case of a tie

in distance, the comparison is made based on a counter, which is introduced exclusively for this purpose. Finally, the third variable is a variant that can store either a point or an index in a NodeType node, which can be either QuadTree or RTree. The two constructors of the structure allow the creation of HeapEntry objects with either point data or node index, respectively.

Files RTree.h and RTree.cpp

The RTree class is implemented in the RTree.h and RTree.cpp files. In the implementation of the R-Tree, data is organized hierarchically into leaves and internal nodes, as indicated by the is_leaf variable. Leaves store points (vector<Point>), while internal nodes contain pointers to subnodes (vector<RTree*> children). Points are initially sorted based on Z-order (Morton) codes to maintain spatial proximity. They are then grouped into sets of size up to max_entries, from which the leaves are constructed, and the intermediate nodes are derived upwards to the root. MBRs are calculated using union_with.

RTree::insert

A bulk loading strategy, similar to the one used in the second exercise, is applied to construct the R-Tree. The data is initially sorted either based on Z-order normalization (Morton order) or following the STR (Sort Tile Recursive) methodology, where it is first separated into strips along the x-axis and then along the y-axis. Next, the sorted points are gradually grouped into nodes, starting from the leaves and proceeding hierarchically to the higher levels until the root of the tree is constructed. The Z-order value for each point is calculated using the external library libmorton*.

*To run the code, the library must be cloned into the code directory as follows: git clone https://github.com/Forceflow/libmorton.

After the calculation, the points are sorted and then divided into consecutive groups, each containing up to max_entries elements. For each such group, a leaf node (with is_leaf = true) is created, which includes the corresponding points. The MBR of

the node is calculated using the compute_boundary function, which locates the minimum and maximum values of the x and y coordinates and forms the smallest possible rectangle that encloses all points. In cases where the last leaf created contains fewer entries than the minimum allowed limit min_entries, certain points are transferred from the previous leaf, and additional processing is applied to meet the lower limit.

After the leaves are created, the upward construction of the tree begins by grouping the leaves into internal nodes. The process follows the same logic, with each parent node containing up to max_entries children. For each node, the MBR that encloses the MBRs of all its children is calculated, again using compute_boundary. The process is repeated, creating levels of internal nodes, until all nodes are merged under a common node.

Finally, when all levels have been constructed, the remaining node is the root of the tree. If it contains direct points, then it is also a leaf; otherwise, it is an internal node that supervises the entire structure.

RTree::range_query

The range_query() function searches for all points within a specific area (rectangle range_rect) and returns a vector<Point> vector with the points that satisfy the query.

First, it checks whether the area covered by the current node (boundary) intersects with the search area range_rect, using intersects(). If there is no overlap, then it is impossible for there to be any relevant points in the subnode, so an empty vector is returned directly. This check is an optimization step, as it avoids unnecessary recursions and comparisons.

Next, if there is overlap and the node is a leaf, all points it contains are checked to see if they are within the range_rect using contains(). Any points that belong to the area are added to the result. Conversely, if the node is internal, the function is called recursively for each of its children. The results from the subtrees are collected in the vector found using insert(), so that a single result is maintained. Finally, the total vector of points within the search area is returned.

RTree::knn_query

The knn_query() function searches for the k nearest neighbors of a query point within the R-Tree. It returns a vector of pairs (pair<Point, float>) that include the point and its distance from the query.

The search is based on a min-heap, which is implemented using a priority_queue with elements of type HeapEntry<RTree>. The < operator of HeapEntry, as mentioned, has been overloaded so that the element with the smallest distance has the highest priority. The first entry in the heap is the root of the tree itself, accompanied by the distance of the query from the root boundary, via the distance_to_rectangle function. The counter is also added, which ensures a stable order among objects with equal distances, as mentioned in the HeapEntry structure.

The main logic of the search is implemented in a while loop, which continues to run until k nearest points are found or until the heap is empty. Each element removed from the top of the heap can be either a point or a tree node, which is managed by the variable variant entry.data. The check is performed using holds_alternative<Point>() to determine what type of data the entry contains.

Thus, if the entry.data element is a point, it is added directly to the results along with its distance, while if it is a node (const RTree*), it is checked to see if it is a leaf or an internal node. If it is a leaf, each point is added to the heap, along with the distance_to_point and the corresponding counter. If it is an internal node, the distance of the query from the boundary of each child is calculated, and each child is added to the heap accordingly. The loop stops as soon as k points are found or the heap is completely empty, and thus all points have been checked before the k neighbors are filled. Finally, the function returns a vector<pair<Point, float>> containing the k closest points to the query.

RTree::print_tree

The print_tree() function implements a recursive printout of the R-Tree, in order to display its structure. First, it prints whether the current node is a leaf (is_leaf) as well as the boundary it covers. If it is a leaf, it prints all the points it contains. Otherwise, the function recursively calls itself for each child, increasing the depth level.

Αρχεία QuadTree.h και QuadTree.cpp

The QuadTree class is implemented in the QuadTree.h and QuadTree.cpp files. Each node of the tree covers a rectangular area, as defined by the boundary variable, and can contain up to a certain number of points, as specified by the capacity variable. The points are initially stored in the node itself, via the points variable. When the capacity limit is exceeded, the node is subdivided into four separate quadrants, northwest, northeast, southwest, and southeast, which are implemented with pointers to corresponding new subnodes. The variable divided indicates whether a node has already been subdivided. The entire implementation of the QuadTree querying functionality (range_query, knn_query) is similar to that of the R-tree.

QuadTree::insert

The insert function is responsible for inserting a new point into the QuadTree structure. The process begins by checking whether the point lies within the rectangle (boundary) that defines the area of the specific node. If the point is outside these boundaries, the insertion is rejected and the function returns false.

Otherwise, if the node has not yet been split (i.e., divided == false) and has free space (i.e., points.size() < capacity), then the point can be added. Before adding, a check is performed to avoid duplicate points by comparing the coordinates of the point with those already stored. If the point is unique, it is added, and the function returns true.

When the node reaches its capacity and has not yet been split, the subdivide function is called, which splits the current node into four separate quadrants: northwest,

northeast, southwest, and southeast. Each new subnode is created with a new rectangle, which covers the corresponding quadrant of the original space. Immediately after the subdivision, all pre-existing points of the node are re-entered into the new subnodes, and the points list is emptied.

Once the split and redistribution are complete, an attempt is made to insert the new point into the four subnodes. The point is inserted into the first subnode that contains it, and if the insertion is successful, true is returned. Otherwise, if no suitable subnode is found or the insertion fails, the function returns false. In reality, the last return false is unreachable and has been inserted for the compiler.

QuadTree::range_query

The logic of the area search is almost identical: it checks whether the boundary of the current node intersects with the search rectangle and, if so, searches for points either in the leaves or in the subnodes. The only difference is that QuadTree always has four fixed subnodes instead of a variable number as in RTree.

QuadTree::knn_query

Finding the k nearest neighbors is almost identical in structure: it uses a min-heap (with HeapEntry), switches between nodes and points via std::variant, and terminates when k points have been found. The difference is only in the traversal: QuadTree always examines four predefined children, while RTree uses a general list of subnodes.

Δομή του Dataset

The datasets used, relate to geographical locations in areas of the United States of America. They contain similar points, but differ in size, with the first T2 file containing 2,280,427 data points and T5 containing 5,043,188 data points. The points in both cases belong to the same range:

- T2: x ∈ [−124.7595, −66.9875], y ∈ [24.5219, 49.1668]
- T5: x ∈ [−124.73, −66.9887], y ∈ [24.5635, 49.1611]

The distributions are not uniform and there is no clear pattern along any axis. A few clusters can be identified, but overall the pattern is quite irregular and scattered, as can be seen in the following diagram:

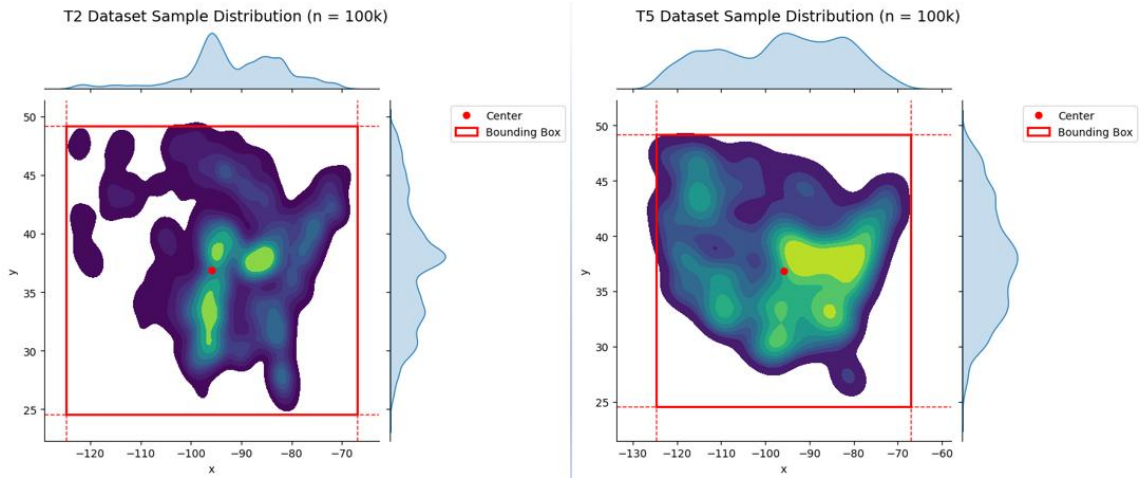

Figure 1: Data Distribution of the Two Datasets

As for the structure of the queries, they are more uniformly distributed and consist of five different files that differ in the size of the range covered by the range query. Each such file consists of 10,000 queries, and the center of each rectangle was used as the point for the k-NN query. The distribution of these centers is shown in the figure below.

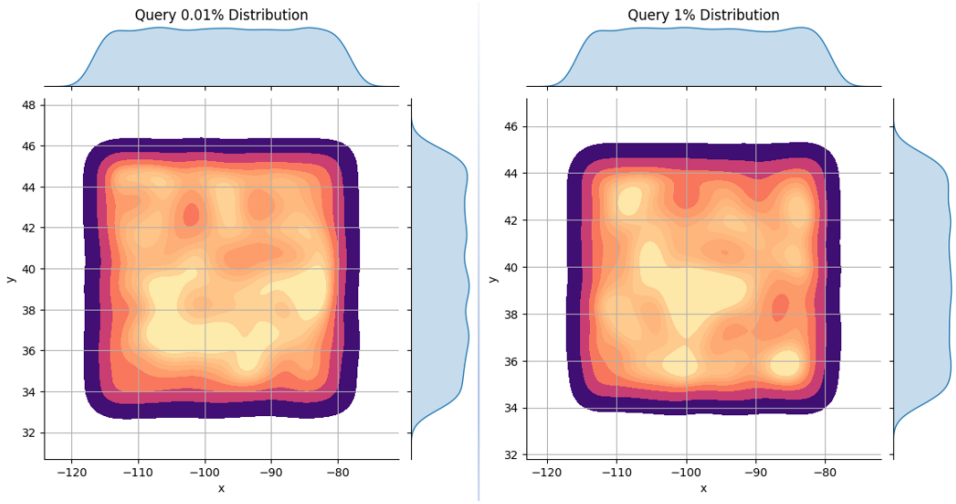

Figure 2: Distribution of the centers of rectangular range queries

The corresponding rectangles are also defined by their width and height beyond their center, and for each queries file, the area fluctuates around a different mean value.

- File 0.01%: average area 0.051
- File 0.05%: average area 0.25
- File 0.1%: average area 0.51
- File 0.5%: average area 2.53
- File 1%: average area 5.07

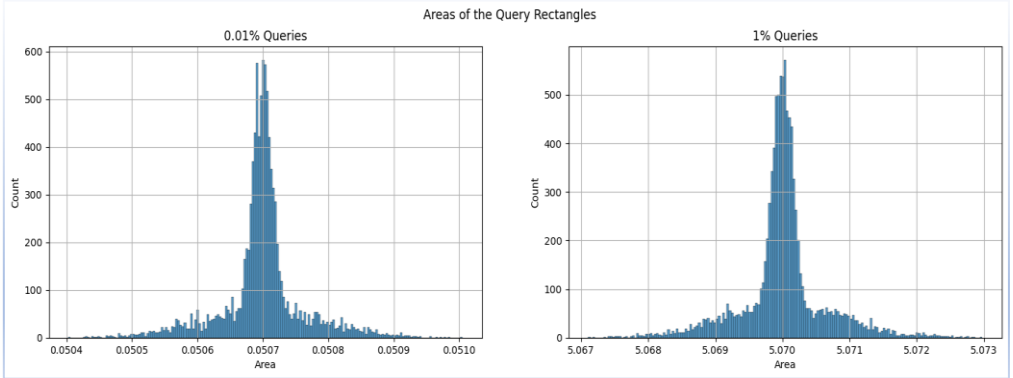

Figure 3: Area distributions of the file with the smallest queries compared to the one with the largest queries

Each query differs in proportions, but the area in each case varies within the same range, as shown in the diagram below. The greater the height, the smaller the width, and vice versa.

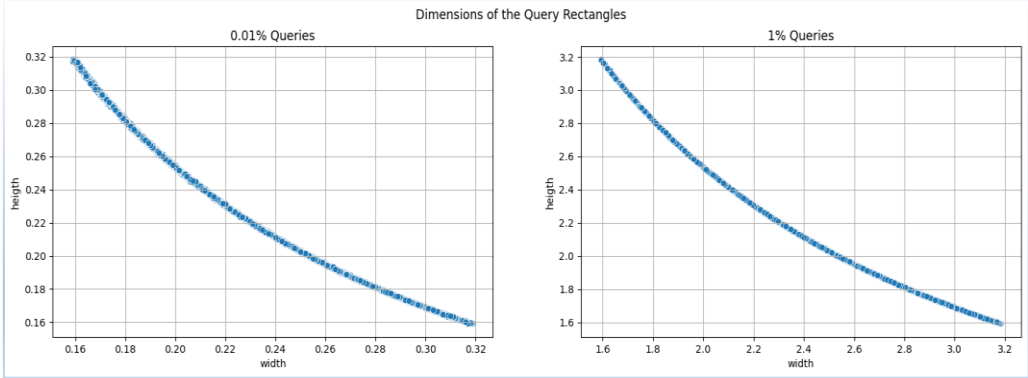

Figure 4: Height x Width of the range of each query for files with the smallest and largest ranges

Results

Optimizing the Capacity value of Quad Tree

Initially, a search was conducted for the appropriate capacity value for the Quad Tree, i.e., the maximum number of points that a leaf can contain before it splits. The T2 dataset with k = 3 was used for these tests, and the goal was to improve the execution time of the k-NN query of the 0.01% file. The optimal value was selected based on the average performance in multiple queries. The results of this tuning are shown in the table below.

capacity	**mean time µs/query**	**Max depth**	**Min depth**	**Avg points per leaf**	**q1**	**q3**	**total leaves**	**internal nodes**
4	86.7533	23	2	1.64924	1	3	1382389	460796
8	68.1143	20	2	3.32551	2	5	685636	228545
16	58.9574	17	2	6.71635	4	10	339502	113167
32	59.8129	15	2	13.30900	8	18	171334	57111
64	72.5485	15	2	26.86070	15	37	84895	28298
128	94.5242	12	2	52.21230	31	71	43675	14558
256	127.5290	11	2	106.27800	61	148	21457	7152

Table 1: Table of time and tree structure results for various capacity values

For this specific dataset and number k, the best results seem to be obtained when the capacity is between 16 and 32.

Optimizing Min and Max Entries values of R-Tree

A similar optimization process was applied to the min and max entries hyperparameters of the R-Tree, examining both implemented data insertion methods. In both cases, the best times for k-NN queries were achieved with min = 4 and max = 8. Furthermore, it was found that the STR insertion method resulted in better data clustering, leading to a generally more efficient response to k-NN queries compared to Z-order sorting. Table 2 shows the relevant times for z-order inserted.

Max entries	**2*min**	**3*min**	**4*min**
Min entries			
4	129.46	165.18	175.72
8	178.472	203.38	244.55
16	271.552	313.52	348.91
32	321.905	409.44	568.93
64	584.56	717.74	884.31

Table 2: Time results table for k-NN, for various min and max entry values for z-order inserted

The following table shows the corresponding results when the data are grouped according to the STR grouping.

Max entries	**2*min**	**3*min**	**4*min**	**5*min**
Min entries				
4	110.748	115.385	112.657	129.127
8	114.713	124.251	142.86	171.055
16	145.113	167.538	199.031	216.589
32	199.24	259.802	296.271	316.802
64	279.377	322.112	349.591	408.514

Table 3: Time results table for k-NN, for various min and max entry values for STR inserted

Comparison of k-NN with baseline methodology

A naïve methodology was also used as a baseline for evaluating the results. Specifically, for each query point, the distances from all points in the T2 dataset were calculated, the distances were sorted, and the k closest points were returned. The performance of each method was compared in terms of the average time per query (in microseconds):

Method	**Naïve**	**Quad Tree***	**R-Tree (z)***	**R-Tree (STR)***
Time (μsec/query)	240.33	44.59	151.38	99.84

*Table 4: Comparison of time per query for each methodology. *The times shown for trees are median values from 10 repetitions.*

The results clearly show that Quad Tree achieved the best performance, being approximately 5.39 times faster than the baseline method. This was followed by R-Tree with STR insertion, which was 2.41 times faster, while R-Tree with Z-order sorting showed a smaller improvement, approximately 1.59 times faster than the naïve approach.

KNN Query Time Analysis by Region

An analysis followed to investigate the existence of "easy" and "difficult" areas in the space, i.e., areas where KNN queries are executed either faster or slower. The study was based on the T2 dataset, with k = 3, using queries from the 0.01% file.

For each query point, 10 repetitions were performed to calculate the statistical mean execution time and standard deviation (mean and std) and to identify possible patterns related to the geographical location of the queries or the local density of the data. The following chart shows the mean and variance for each query point in the 0.01% file.

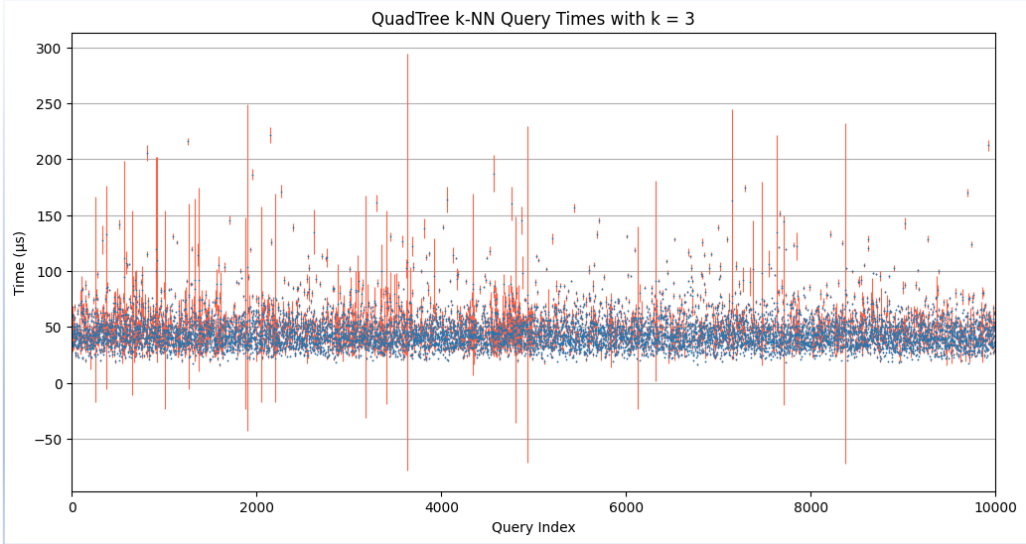

Figure 5: Average time and standard deviation for each query in the 0.01% file, in the T2 dataset

However, since the algorithm is deterministic, points with high std result from outliers due to the machine, as can be seen from point 3630, which is the one with the highest variance.

επανάληψη	1	2	3	4	5	6	7	8	9	10
χρόνος (μsec)	638	77	46	45	46	46	47	47	46	45

Table 5: Query time 3630 in the T2 dataset, for each iteration

Συνεπώς λαμβάνεται η τιμή median, ως αντιπροσωπευτική τιμή για χρόνου για δεδομένο query.

The distribution of median query values appears to follow a right-skewed distribution, suggesting that there are areas in the space where queries take longer to answer, i.e., there are “more difficult” areas.

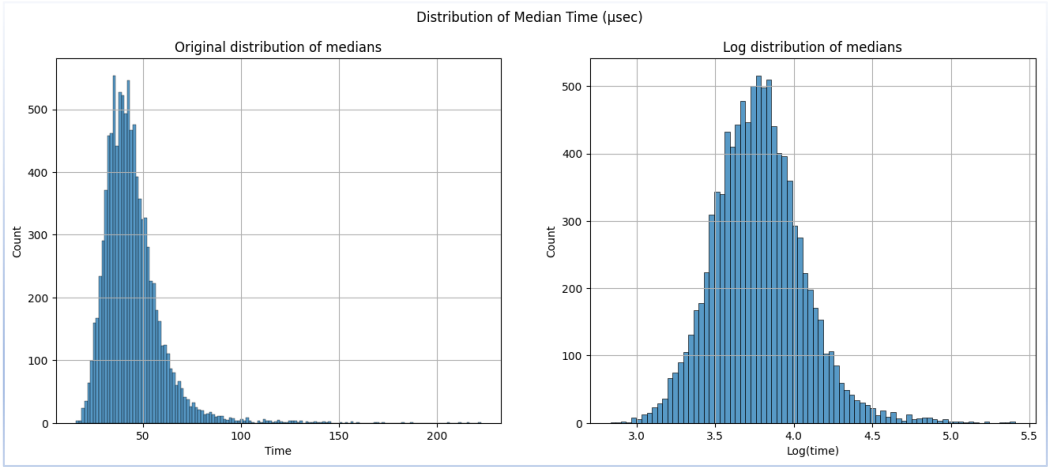

Figure 6: Distribution of median query times and distribution of log(median times)

To better visualize this information, a logarithmic transformation (log transformation) was applied to the time values, and the transformed value was used as a color scale for the corresponding points in the query set (0.01% file). This makes it possible to identify the areas associated with higher computational costs.

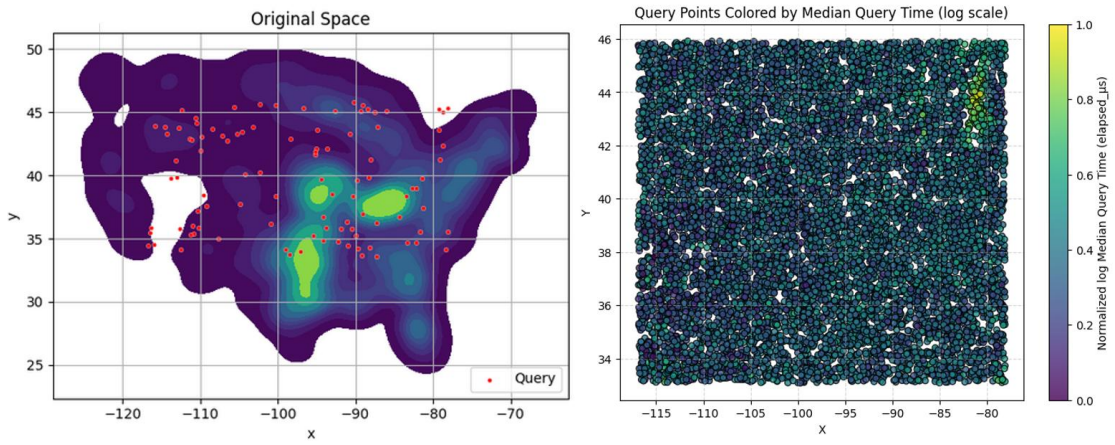

Figure 7: On the left is the T2 dataset with sample query points, and on the right are the query points, with the fastest ones in purple and the slowest ones in light green.

From the right-hand side of Figure 7, it appears that the region responsible for the right skewness of the time distribution approximately belongs to the range $x \in [-81, -80]$, $y \in [42, 45]$.

Space Transformation

Next, an attempt was made to optimize k-NN search times in Quad Tree through appropriate space transformation, with the aim of improving data distribution in the tree. A type of transformation was selected that does not affect the distances between points, in order to maintain the accuracy of the results. Specifically, rotations of the space were used.

The goal was to produce a more evenly balanced Quad Tree, which could reduce the average search time. Initially, PCA (Principal Component Analysis) was applied to align the axes with the greatest variance with the tree's separation axes ($x = 0$, $y = 0$). However, as the rotation matrix resulting from PCA was very close to identity, the alignment it provided was minimal. Therefore, to determine whether there was indeed

a difference with rotation, after PCA, a 45° rotation was performed (original → PCA → 45°).

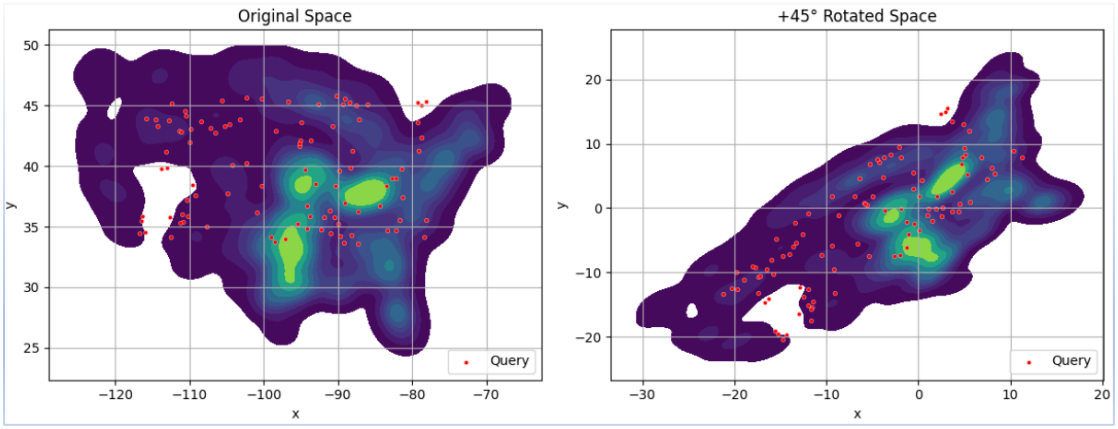

Figure 8: Transformation of space at 45° away from PCA transformation

The transformation of the query points was performed using the same transformation matrix that resulted from the PCA of the data, in order to maintain the relative geometry between queries and dataset. Subsequently, an additional 45° rotation was applied using the corresponding transformation matrix.

The results of this rotation did not show a significant difference in the execution time of k-NN. Specifically, the average time per query decreased marginally, from 44.59 μsec to 43.43 μsec (improvement ~2.5%). To assess whether this difference is statistically significant, the times were transformed using a logarithmic scale so that their distribution approximates the normal distribution. A paired t-test was then applied, which yielded a p-value = 4.4×10^{-15}, indicating that, although the difference is small, it is statistically significant. However, its practical significance remains negligible.

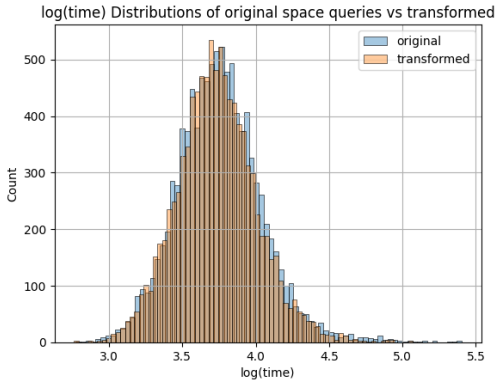

Figure 9: Distributions of log(time) before and after the transformation of space

Experimental Evaluation of Rotations

As a continuation of the previous analysis, the space was rotated from 0° to 90° relative to the PCA axes, in 1° increments. For each rotation angle, a new Quad Tree was constructed and statistical characteristics of its structure were collected, such as: min depth, max depth, average number of points per leaf, standard deviation of leaf population, as well as the 1st and 3rd quartiles (Q1, Q3) of leaf size.

The aim of this process was to investigate the existence of a correlation between the statistical characteristics of the tree and the average execution time of a k-NN query. If such a relationship is found, the second stage examines whether the rotation angle can predict or control these statistics in order to identify desirable rotation angles of the space that optimize the tree structure and, by extension, the efficiency of queries.

The overall approach aims to construct a space alignment strategy prior to tree construction, which leads to shallower and more balanced trees, offering faster response times for repeated k-NN queries.

An initial analysis of the data shows that there is a correlation between certain characteristics and query execution time. Specifically, as the heterogeneity of the tree decreases (e.g., min depth increases and max depth decreases), the average query time increases and outliers decrease.

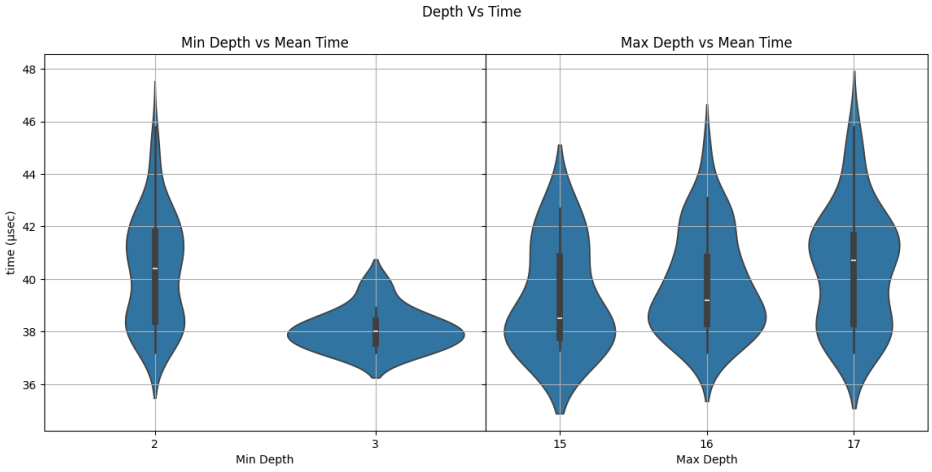

Figure 10: Violin plots of min depth and max depth over time in microseconds

In the next stage, an exhaustive search of features for k-NN time prediction was performed, using a Decision Tree model with max_depth = 4. R2 was used as the evaluation metric, as a positive R2 indicates a reduction in prediction uncertainty

compared to using the average of the time distribution. After this search, the best subset was found to be the max depth, min depth, and the third quartile of the leaf size. The corresponding R2 achieved was 0.161, which is low but confirms that the use of these features improves predictability compared to the baseline model (use of the average without features).

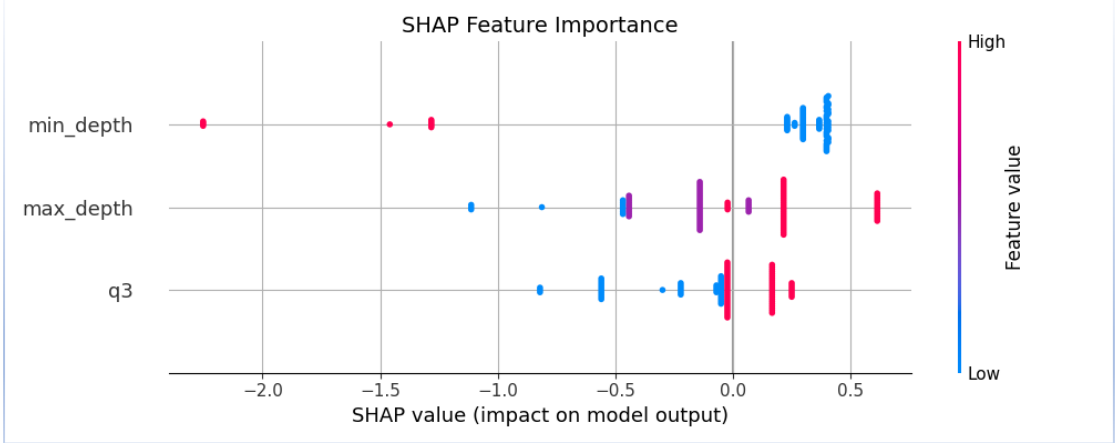

Figure 11: SHAP Values for the Decision Tree with the best subset of features.

The analysis of SHAP values revealed min depth as the most important feature, which is associated with a reduction in execution time when its value is higher (depth 3 instead of 2). This is followed by max depth, which has the opposite effect: the greater the maximum depth, the longer the expected time for a query. Finally, the Q3 feature also appears to be positively correlated with time; as its value increases, so does the predicted time per query.

Quad Tree vs R-Tree k-NN Queries

To compare the two structures in terms of k-NN query time, experiments were conducted using the 0.01% file of centers in both data files. Initially, in the T2 dataset, it appears that for every number k, the Quad Tree is faster than the two R-Trees, and for large values of k, z-order clustering is better than STR clustering. They also follow a linear relationship in the range studied for the trees.

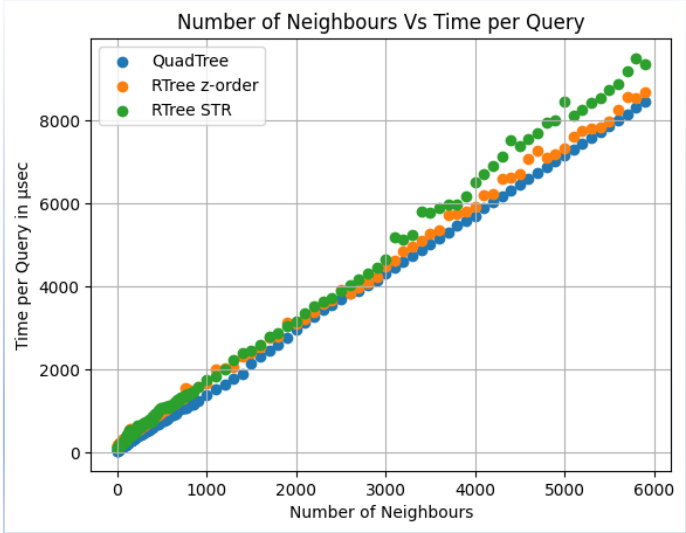

Figure 12: Number k vs. Time per query for the T2 dataset

The table below shows the same result as in the subsection "Optimization of Min and Max Entries in R-Tree" in Tables 2 and 3, where at low k values (close to the intercept), STR grouping was better than z-order grouping. However, as the slopes follow the opposite order, at large k, the speed series of the R-Tree queries are reversed.

	Quad Tree	R-Tree (z-order)	R-Tree (STR)
Intercept (µsec/query)	61.878	228.218	181.087
Slope (µsec/query/k)	1.418	1.440	1.572

Table 6: Table showing the slopes and intercepts of the results for each tree in the T2 dataset

Similar results were obtained in the same study on the T5 Dataset, with the difference that the STR inserted R-Tree was better even at low k than the corresponding tree with z-order data insertion.

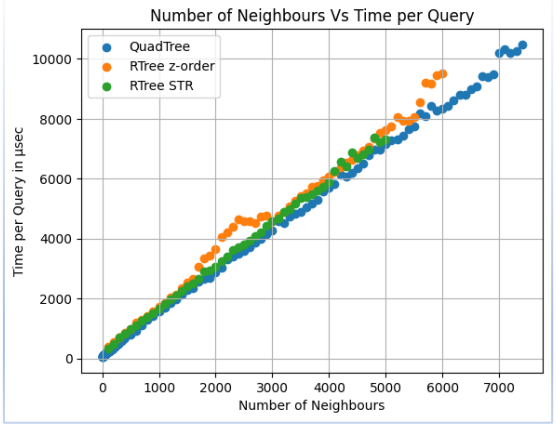

Figure 13: Number k vs. Time per query for the T5 dataset

For the linear fit of the R-Tree z-order results, points ~1700-3000 and 5600+ were excluded, as they appear to be outliers and to have been generated by the computer.

	Quad Tree	R-Tree (z-order)	R-Tree (STR)
Intercept (μsec/query)	89.71	526.07	181.087
Slope (μsec/query/k)	1.397	1.417	1.448

Table 7: Table showing the slopes and intercepts of the results for each tree in the T2 dataset

Range Query Comparison (T2 dataset, all ranges)

With regard to Range Queries, a naïve methodology was also implemented, whereby each point in the dataset is checked to see if it is within the range query. Thus, for each file with queries (0.01%, 0.05%, 0.1%, 0.5%, 1%), the time is constant (with random variation), as the number of operations is the same (for each query range, all points are checked). In contrast to the naïve methodology, the other tree-based methodologies show an increase in execution time as the query range increases.

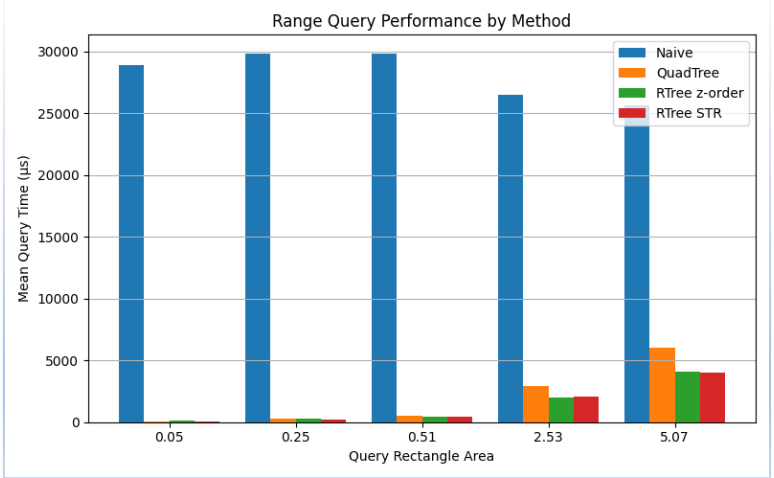

Figure 14: Comparison of range query times for each methodology in each query file

Quad Tree vs R-Tree Range Queries

With regard to the difference between the two trees, the picture obtained is the opposite of that obtained for the speed of k-NN queries. Initially, in the T2 dataset, it appears that the quadtree takes longer as the query range increases, compared to the

two R-Trees. And in the specific case of Range Queries, it seems to follow a linear relationship in this range.

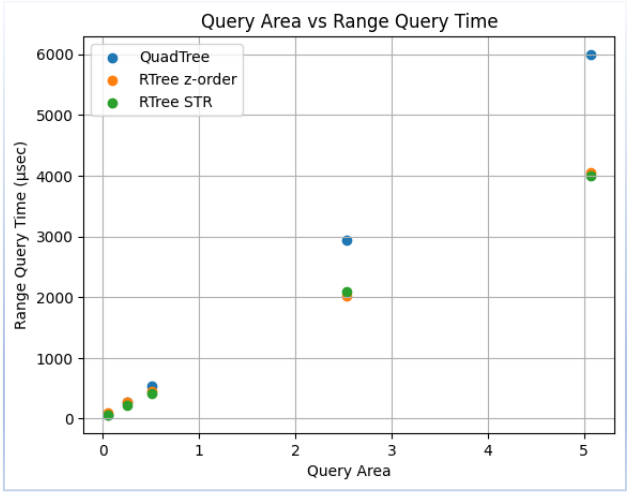

Figure 15: Query range vs. Time per query for the T2 dataset

The slope of the Quad Tree is much greater than that of R-Trees, while in very small query ranges, it appears to be slightly faster than the two R-Trees.

	Quad Tree	R-Tree (z-order)	R-Tree (STR)
Intercept (µsec/query)	-29.62	57.02	32.73
Slope (µsec/query/k)	1184.67	785.89	787.73

Table 8: Table showing the slopes and intercepts of the results for each tree in the T2 dataset

Similar results were obtained when studying the times in the T5 dataset.

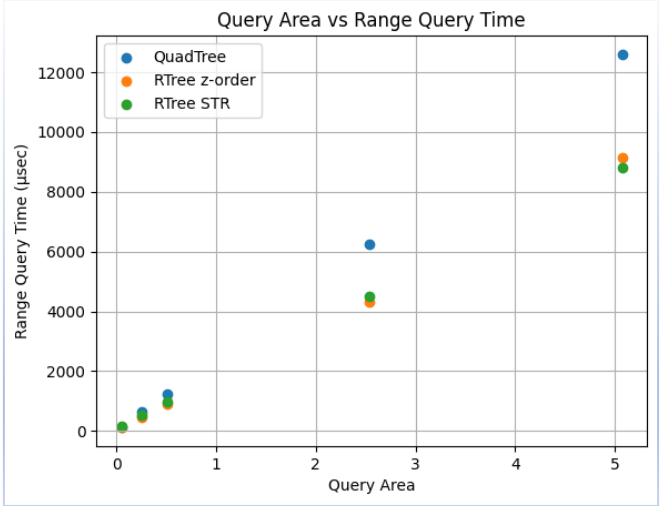

Figure 16: Query range vs. Time per query for the T5 dataset

	Quad Tree	R-Tree (z-order)	R-Tree (STR)
Intercept (µsec/query)	-12.57	-16.03	108.02
Slope (µsec/query/k)	2483.09	1787.79	1717.63

Table 9: Table showing the slopes and intercepts of the results for each tree in the T5 dataset

Conclusions

This paper highlighted the effectiveness of spatial structures (Quad Tree and R-Tree) compared to naive search methodologies. Specifically, significant improvements in execution time were achieved, with Quad Tree being up to 5.39 times faster than the baseline approach, and R-Tree with STR insertion following with 2.41 times better performance for specific dataset conditions and number of neighbors.

It was observed that naive methods have a constant execution time regardless of the query size, while tree structures show a linear correlation between query size (in area) and execution time. At the level of individual methodologies, R-Trees perform better in range queries, especially as the query area increases, while Quad Trees marginally outperform in k-NN queries, mainly due to their simpler and more stable topology.

The attempt to transform the space through rotations revealed that the difference in times was small but statistically significant (p-value ≈ 0.05). However, systematic trends were recorded in the times associated with statistical characteristics of the tree, such as min depth (higher values are associated with lower times), max depth (higher values lead to an increase in time), and Q3 (higher values also lead to a slight increase in time).

Finally, the use of prediction models (Decision Trees) for the execution time of a query, based on tree statistics, showed a low R^2 (~0.16), but revealed a slight improvement in prediction uncertainty compared to prediction in the absence of these characteristics.