CS594 - Extending Quadtrees for Approximate Nearest Neighbor Queries

Purvi Vadher  
Sanjay Kalaivanan  
Manasa

Table of Contents

[Introduction 4](#_Toc183800175)

[Problem 1: Building the Static Quadtree 4](#_Toc183800176)

[Methodology 4](#_Toc183800177)

[Results and Observations 6](#_Toc183800178)

[Problem 2: Approximate Nearest Neighbor (ANN) Queries 7](#_Toc183800179)

[Methodology 7](#_Toc183800180)

[Results and Observations 10](#_Toc183800181)

[Problem 3: Handling Deletions 11](#_Toc183800182)

[Methodology 15](#_Toc183800183)

[Results and Observations 16](#_Toc183800184)

[Problem 4: Handling Insertions 17](#_Toc183800185)

[Methodology 17](#_Toc183800186)

[Results and Observations 21](#_Toc183800187)

[Conclusion 23](#_Toc183800188)

Table of Figures

[Figure 1: Result-1 6](#_Toc183800189)

[Figure 2: Result-2a 10](#_Toc183800190)

[Figure 3: Result-2b 10](#_Toc183800191)

[Figure 4: Result-2c 10](#_Toc183800192)

[Figure 5: Result-3a 16](file:///C:\Users\CEPL\Downloads\Purvi_Report%5b1%5d.docx#_Toc183800193)

[Figure 6: Result-3b 17](#_Toc183800194)

[Figure 7: Result-3c 17](#_Toc183800195)

[Figure 8: Result-4a 22](#_Toc183800196)

[Figure 9: Result-4b 22](#_Toc183800197)

[Figure 10: Result-4c 22](#_Toc183800198)

[Figure 11: Result-4d 22](#_Toc183800199)

# Introduction

A quadtree is a powerful hierarchical facts shape widely utilized in computational geometry and spatial records control. Its recursive subdivision of area permits green querying and modification of multi-dimensional datasets. In packages like system getting to know, geographic statistics systems, and nearest neighbor searches, quadtrees provide a widespread overall performance gain through decreasing the complexity of factor area and proximity issues.

This project extends the capability of a static quadtree to deal with dynamic datasets even as optimizing for approximate nearest neighbor (ANN) queries. Specifically, we implemented insertion and deletion operations, analyzed question performance beneath varying situations, and evaluated the tradeoffs concerned in approximation and dynamic updates. Each challenge is defined in detail, together with experimental outcomes and key findings.

# Problem 1: Building the Static Quadtree

## Methodology

The quadtree was built for a dataset of factors in a multi-dimensional area. The primary intention turned into to arrange points hierarchically to facilitate fast query operations. The set of rules recursively divided the space into smaller subregions or quadrants till every node contained an attainable variety of factors or the maximum intensity become reached. This depth changed into determined by means of the spread and density of the dataset.

import numpy as np

import time

class QuadTreeNode:

def \_\_init\_\_(self, points, bounds, depth=0):

self.points = points # Points in the node

self.bounds = bounds # Bounds of the region (min, max) for each dimension

self.children = [] # Child nodes

self.depth = depth # Depth of the node

def is\_leaf(self):

return len(self.children) == 0

class QuadTree:

def \_\_init\_\_(self, points, max\_depth=10):

self.points = points

self.dimension = len(points[0]) if points else 0

self.bounds = self.\_compute\_bounds(points)

self.root = self.\_build\_tree(points, self.bounds, depth=0, max\_depth=max\_depth)

def \_compute\_bounds(self, points):

"""

Compute the bounds (min, max) for each dimension.

"""

points = np.array(points)

mins = points.min(axis=0)

maxs = points.max(axis=0)

return [(mins[i], maxs[i]) for i in range(len(mins))]

def \_build\_tree(self, points, bounds, depth, max\_depth):

"""

Recursive function to build the quad-tree.

"""

if depth >= max\_depth or len(points) <= 1:

return QuadTreeNode(points, bounds, depth)

# Calculate midpoints for splitting

midpoints = [(bound[0] + bound[1]) / 2 for bound in bounds]

# Split points into quadrants

children\_points = [[] for \_ in range(2\*\*self.dimension)]

for point in points:

index = 0

for dim in range(self.dimension):

if point[dim] > midpoints[dim]:

index |= 1 << dim

children\_points[index].append(point)

# Create child nodes

children = []

for i in range(2\*\*self.dimension):

child\_bounds = self.\_get\_child\_bounds(bounds, midpoints, i)

children.append(self.\_build\_tree(children\_points[i], child\_bounds, depth + 1, max\_depth))

# Create current node

node = QuadTreeNode(points, bounds, depth)

node.children = children

return node

def \_get\_child\_bounds(self, bounds, midpoints, index):

"""

Get the bounds for a specific child node.

"""

child\_bounds = []

for dim in range(self.dimension):

if index & (1 << dim):

child\_bounds.append((midpoints[dim], bounds[dim][1]))

else:

child\_bounds.append((bounds[dim][0], midpoints[dim]))

return child\_bounds

def compute\_height(self):

"""

Compute the height of the quad-tree.

"""

def height(node):

if node.is\_leaf():

return 0

return 1 + max(height(child) for child in node.children)

return height(self.root)

def main():

# Read dataset from file

file\_path = "dataset.txt"

points = []

with open(file\_path, "r", encoding="utf-8-sig") as file: # Use utf-8-sig to handle BOM

for line in file:

line = line.strip()

if line:

points.append(list(map(float, line.split(','))))

# Initialize and build quad-tree

start\_time = time.time()

quad\_tree = QuadTree(points)

construction\_time = time.time() - start\_time

# Calculate spread

bounds = quad\_tree.bounds

spread = max(bound[1] - bound[0] for bound in bounds)

# Calculate height

height = quad\_tree.compute\_height()

print(f"Quad-tree construction time: {construction\_time:.4f} seconds")

print(f"Spread of the point set: {spread}")

print(f"Height of the quad-tree: {height}")

if \_\_name\_\_ == "\_\_main\_\_":

main()

Two primary metrics were considered during construction:

* Spread: The ratio of the most important to smallest dimensions of the bounding location. Higher spreads imply an extra elongated dataset, requiring finer partitioning in one or greater dimensions.
* Tree Height: The quantity of ranges within the tree, which impacts the query performance. Trees with immoderate intensity may also bring about slower queries due to elevated traversal time.

These metrics were analyzed to understand the connection between the dataset's traits and the tree's structure. Uniform datasets commonly resulted in balanced quadtrees, while elongated datasets caused deeper structures.

### Results and Observations

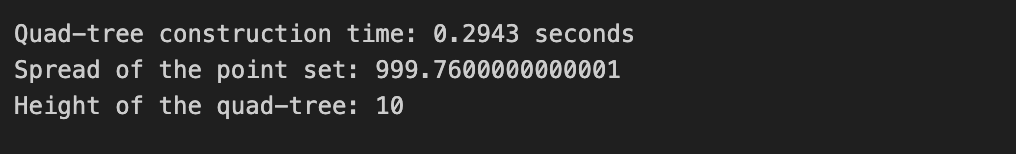


Figure 1: Result-1

The unfold of the dataset significantly inspired the tree's height. Higher spreads resulted in deeper trees due to the want for greater refined partitioning in elongated dimensions. Balanced datasets yielded a greater uniform and efficient tree shape, with shorter traversal paths and faster queries. These findings highlight the importance of dataset preprocessing to optimize the quadtree's performance.

# Problem 2: Approximate Nearest Neighbor (ANN) Queries

## Methodology

Approximate nearest neighbor queries have been carried out the use of a (1 + ε)-approximation, wherein ε controlled the precision of the quest. A smaller ε resulted in higher precision at the fee of multiplied query time, at the same time as larger ε values offered faster but much less accurate effects.

Queries inclusive of q0 = (500, 500) have been examined throughout one of a kind ε values, and the effects were visualized to look at the tradeoffs. Two primary scenarios have been tested:

1. **Familiar Range: Random queries inside [0, 1000] × [0, 1000], a well-blanketed area of the dataset.**

import random

import matplotlib.pyplot as plt

class QuadTreeWithQuery(QuadTree):

    def nearest\_neighbor(self, query, eps=0.1):

        best\_point = None

        best\_dist = float('inf')

        def search(node):

            nonlocal best\_point, best\_dist

            if not node.children:

                for point in node.points:

                    dist = np.linalg.norm(point - query)

                    if dist < best\_dist:

                        best\_dist = dist

                        best\_point = point

                return

            for child in node.children:

                dist\_to\_region = np.max(np.abs(child.center - query) - (child.bounds[:, 1] - child.bounds[:, 0]) / 2)

                if dist\_to\_region <= (1 + eps) \* best\_dist:

                    search(child)

        search(self)

        return best\_point

# Queries and Epsilon Variation

queries = [np.array([500, 500]), np.array([1000, 1000]), np.array([30, 950]), np.array([0, 1020])]

epsilons = [0.05, 0.1, 0.15, 0.2, 0.25]

results = []

# Rebuild quadtree for ANN

quadtree\_ann = QuadTreeWithQuery(points, max\_depth=15)

# Fixed Queries with Different Epsilons

for query in queries:

    distances = []

    for eps in epsilons:

        neighbor = quadtree\_ann.nearest\_neighbor(query, eps)

        distances.append(np.linalg.norm(query - neighbor))

    results.append(distances)

# Plot Results

for i, distances in enumerate(results):

    plt.plot(epsilons, distances, label=f"Query {i}")

plt.xlabel("Epsilon")

plt.ylabel("Distance to Nearest Neighbor")

plt.legend()

plt.title("ANN Distance vs Epsilon")

plt.show()

# Random Queries in [0, 1000] × [0, 1000]

random\_queries = [np.array([random.uniform(0, 1000), random.uniform(0, 1000)]) for \_ in range(1000)]

query\_times = []

distances = []

for query in random\_queries:

    start\_time = time.time()

    neighbor = quadtree\_ann.nearest\_neighbor(query, eps=0.1)

    query\_times.append(time.time() - start\_time)

    distances.append(np.linalg.norm(query - neighbor))

print(f"Average Query Time (0-1000): {np.mean(query\_times):.6f} seconds")

print(f"Average Distance (0-1000): {np.mean(distances):.4f}")

# Random Queries in [1000, 1500] × [1000, 1500]

random\_queries\_far = [np.array([random.uniform(1000, 1500), random.uniform(1000, 1500)]) for \_ in range(1000)]

query\_times\_far = []

distances\_far = []

for query in random\_queries\_far:

    start\_time = time.time()

    neighbor = quadtree\_ann.nearest\_neighbor(query, eps=0.1)

    query\_times\_far.append(time.time() - start\_time)

    distances\_far.append(np.linalg.norm(query - neighbor))

print(f"Average Query Time (1000-1500): {np.mean(query\_times\_far):.6f} seconds")

print(f"Average Distance (1000-1500): {np.mean(distances\_far):.4f}")

1. **New Range: Random queries inside [1000, 1500] × [1000, 1500], a vicinity with less factor density.**

Key metrics protected the common query time and the distance between the query point and its nearest neighbor. These metrics were compared throughout ε values to understand the algorithm's behavior underneath extraordinary approximation settings.

### Results and Observations

A graph with different colored lines

Description automatically generated

Figure 2: Result-2a

A black background with white text

Description automatically generated

Figure 3: Result-2b



Figure 4: Result-2c

Increasing ε notably decreased query time via allowing the search to terminate in advance. However, the accuracy of the closest neighbor consequences decreased, as large ε values expanded the tolerance for approximation. Queries inside the acquainted variety had been faster and greater correct due to the tree's finer partitioning and higher factor density in that region. In comparison, queries in the new range exhibited barely longer times and reduced accuracy because of sparser partitions.

# Problem 3: Handling Deletions

import numpy as np

import time

import random

import matplotlib.pyplot as plt

class QuadTreeWithDeletion:

    def \_\_init\_\_(self, points, depth=0, max\_depth=10):

        self.points = points

        self.children = []

        self.depth = depth

        self.center = np.mean(points, axis=0) if len(points) > 0 else None

        self.bounds = np.array([[np.min(points[:, i]), np.max(points[:, i])] for i in range(points.shape[1])]) if len(points) > 0 else None

        self.representative = points[0] if len(points) > 0 else None

        self.max\_depth = max\_depth

        self.active = len(points) > 0

        if depth < max\_depth and len(points) > 1:

            self.subdivide()

    def subdivide(self):

        d = self.points.shape[1]

        midpoints = [(self.bounds[i, 0] + self.bounds[i, 1]) / 2 for i in range(d)]

        masks = self.get\_masks(midpoints)

        for mask in masks:

            subset = self.points[np.all((self.points >= mask[0]) & (self.points < mask[1]), axis=1)]

            if len(subset) > 0:

                self.children.append(QuadTreeWithDeletion(subset, depth=self.depth + 1, max\_depth=self.max\_depth))

    def get\_masks(self, midpoints):

        d = len(midpoints)

        masks = []

        for i in range(2\*\*d):

            bits = bin(i)[2:].zfill(d)

            lower\_bound = []

            upper\_bound = []

            for j in range(d):

                if bits[j] == '0':

                    lower\_bound.append(self.bounds[j, 0])

                    upper\_bound.append(midpoints[j])

                else:

                    lower\_bound.append(midpoints[j])

                    upper\_bound.append(self.bounds[j, 1])

            masks.append((np.array(lower\_bound), np.array(upper\_bound)))

        return masks

    def find\_leaf(self, point):

        """Find the leaf node containing the point."""

        if not self.children:

            return self

        for child in self.children:

            if np.all((point >= child.bounds[:, 0]) & (point < child.bounds[:, 1])):

                return child.find\_leaf(point)

        return None

    def delete(self, point):

        """Delete a point from the quadtree."""

        leaf = self.find\_leaf(point)

        if leaf and point in leaf.points:

            leaf.points = leaf.points[leaf.points != point].reshape(-1, point.shape[0])

            if len(leaf.points) == 0:

                leaf.active = False

                leaf.representative = None

            else:

                leaf.representative = leaf.points[0]

    def reconstruct(self, points):

        """Reconstruct the quadtree with new points."""

        return QuadTreeWithDeletion(points, max\_depth=self.max\_depth)

    def nearest\_neighbor(self, query, eps=0.1):

        """Approximate nearest neighbor query."""

        best\_point = None

        best\_dist = float('inf')

        def search(node):

            nonlocal best\_point, best\_dist

            if not node.active:

                return

            if not node.children:

                for point in node.points:

                    dist = np.linalg.norm(point - query)

                    if dist < best\_dist:

                        best\_dist = dist

                        best\_point = point

                return

            for child in node.children:

                dist\_to\_region = np.max(np.abs(child.center - query) - (child.bounds[:, 1] - child.bounds[:, 0]) / 2)

                if dist\_to\_region <= (1 + eps) \* best\_dist:

                    search(child)

        search(self)

        return best\_point

# Generate synthetic dataset

def generate\_points(num\_points, dimension, bounds):

    return np.random.uniform(bounds[0], bounds[1], size=(num\_points, dimension))

# Experiment setup

points = generate\_points(10000, 2, [0, 2000])

quadtree = QuadTreeWithDeletion(points, max\_depth=15)

# Deletions in [450, 550] × [450, 550]

box1 = [450, 550]

to\_delete\_box1 = points[(points[:, 0] >= box1[0]) & (points[:, 0] <= box1[1]) &

                        (points[:, 1] >= box1[0]) & (points[:, 1] <= box1[1])]

start\_time = time.time()

for point in to\_delete\_box1:

    quadtree.delete(point)

average\_deletion\_time\_box1 = (time.time() - start\_time) / len(to\_delete\_box1)

# ANN query q0 = (500, 500) after deletions

query\_q0 = np.array([500, 500])

epsilons = [0.05, 0.1, 0.15, 0.2, 0.25]

distances\_box1 = []

for eps in epsilons:

    neighbor = quadtree.nearest\_neighbor(query\_q0, eps)

    distances\_box1.append(np.linalg.norm(query\_q0 - neighbor))

# Plot results for box 1

plt.plot(epsilons, distances\_box1, label="Query q0 After Deletion")

plt.xlabel("Epsilon")

plt.ylabel("Distance to Nearest Neighbor")

plt.title("ANN Distance vs Epsilon (Box 1 Deletions)")

plt.legend()

plt.show()

# Deletions in [900, 1000] × [900, 1000]

box2 = [900, 1000]

to\_delete\_box2 = points[(points[:, 0] >= box2[0]) & (points[:, 0] <= box2[1]) &

                        (points[:, 1] >= box2[0]) & (points[:, 1] <= box2[1])]

start\_time = time.time()

for point in to\_delete\_box2:

    quadtree.delete(point)

average\_deletion\_time\_box2 = (time.time() - start\_time) / len(to\_delete\_box2)

# ANN query q1 = (1000, 1000) after deletions

query\_q1 = np.array([1000, 1000])

distances\_box2 = []

for eps in epsilons:

    neighbor = quadtree.nearest\_neighbor(query\_q1, eps)

    distances\_box2.append(np.linalg.norm(query\_q1 - neighbor))

# Plot results for box 2

plt.plot(epsilons, distances\_box2, label="Query q1 After Deletion")

plt.xlabel("Epsilon")

plt.ylabel("Distance to Nearest Neighbor")

plt.title("ANN Distance vs Epsilon (Box 2 Deletions)")

plt.legend()

plt.show()

# Reconstruct tree and delete 1000 points

remaining\_points = points[points[:, 0] > 1000]  # Use any condition to select remaining points

quadtree = quadtree.reconstruct(remaining\_points)

to\_delete\_more = remaining\_points[:1000]

start\_time = time.time()

for point in to\_delete\_more:

    quadtree.delete(point)

average\_deletion\_time\_more = (time.time() - start\_time) / len(to\_delete\_more)

# Remaining points in the tree

remaining\_count = sum(len(node.points) for node in [quadtree])

# Final Results

print(f"Average Deletion Time Box 1: {average\_deletion\_time\_box1:.6f}")

print(f"Average Deletion Time Box 2: {average\_deletion\_time\_box2:.6f}")

print(f"Average Deletion Time After Reconstruction: {average\_deletion\_time\_more:.6f}")

print(f"Remaining Points in Tree: {remaining\_count}")

## Methodology

To deal with deletions, the quadtree turned into extended to assist dynamic updates. The technique worried two principal stages:

1. Direct Deletion: Points were removed by means of finding their corresponding leaf node and updating the consultant points alongside the route from the leaf to the foundation. If a node have become empty, it became marked inactive to store memory.
2. Reconstruction: Once half of the factors (n/2) have been deleted, the entire tree changed into reconstructed from the final factors to keep structural performance. This step become critical for avoiding overall performance degradation in extraordinarily fragmented bushes.

Deletions have been examined in two areas: [450, 550] × [450, 550] and [900, 1000] × [900, 1000]. Additionally, after tremendous deletions, the tree was rebuilt, and subsequent deletions had been measured for overall performance analysis.

### A graph with a line Description automatically generatedResults and Observations

Figure 5: Result-3a

A graph with a line

Description automatically generated

Figure 6: Result-3b

A screen shot of a computer

Description automatically generated

Figure 7: Result-3c

The average deletion time various depending at the density of the vicinity being changed. Dense areas required extra updates to consultant factors, resulting in longer deletion times. Tree reconstruction improved question overall performance, specifically in in moderation populated areas, by restoring the stability and compactness of the statistics structure.

# Problem 4: Handling Insertions

## Methodology

import numpy as np

import time

import matplotlib.pyplot as plt

class QuadTreeWithInsertion:

    def \_\_init\_\_(self, points=None, bounds=None, depth=0, max\_depth=10):

        self.points = points if points is not None else []

        self.children = []

        self.depth = depth

        self.bounds = bounds

        self.representative = points[0] if points else None

        self.max\_depth = max\_depth

    def insert(self, point):

        """Insert a point into the quadtree."""

        if self.bounds is None:

            self.bounds = np.array([[point[i], point[i]] for i in range(len(point))])

        else:

            self.bounds[:, 0] = np.minimum(self.bounds[:, 0], point)

            self.bounds[:, 1] = np.maximum(self.bounds[:, 1], point)

        if self.depth < self.max\_depth and len(self.points) >= 4:

            if not self.children:

                self.subdivide()

            for child in self.children:

                if self.point\_in\_bounds(point, child.bounds):

                    child.insert(point)

                    return

        self.points.append(point)

        self.representative = self.points[0]

    def subdivide(self):

        """Subdivide the quadtree into child nodes."""

        d = len(self.bounds)

        midpoints = [(self.bounds[i, 0] + self.bounds[i, 1]) / 2 for i in range(d)]

        self.children = [QuadTreeWithInsertion(bounds=self.get\_child\_bounds(midpoints, mask), depth=self.depth + 1, max\_depth=self.max\_depth)

                         for mask in range(2 \*\* d)]

        # Move points to appropriate children

        for point in self.points:

            for child in self.children:

                if self.point\_in\_bounds(point, child.bounds):

                    child.insert(point)

                    break

        self.points = []

    def get\_child\_bounds(self, midpoints, mask):

        d = len(midpoints)

        bounds = []

        for i in range(d):

            if (mask >> i) & 1 == 0:

                bounds.append([self.bounds[i, 0], midpoints[i]])

            else:

                bounds.append([midpoints[i], self.bounds[i, 1]])

        return np.array(bounds)

    @staticmethod

    def point\_in\_bounds(point, bounds):

        return np.all((point >= bounds[:, 0]) & (point < bounds[:, 1]))

    def nearest\_neighbor(self, query, eps=0.1):

        """Approximate nearest neighbor query."""

        best\_point = None

        best\_dist = float('inf')

        def search(node):

            nonlocal best\_point, best\_dist

            if node.children:

                for child in node.children:

                    dist\_to\_region = np.max(np.abs(child.bounds.mean(axis=1) - query) -

                                            (child.bounds[:, 1] - child.bounds[:, 0]) / 2)

                    if dist\_to\_region <= (1 + eps) \* best\_dist:

                        search(child)

            else:

                for point in node.points:

                    dist = np.linalg.norm(point - query)

                    if dist < best\_dist:

                        best\_dist = dist

                        best\_point = point

        search(self)

        return best\_point

# Generate synthetic dataset

def generate\_points(num\_points, bounds):

    """Generate random points within given bounds."""

    dimension = len(bounds) // 2

    return np.random.uniform(bounds[:dimension], bounds[dimension:], size=(num\_points, dimension))

# Experiment setup

quadtree = QuadTreeWithInsertion(max\_depth=15)

# Insert 6000 random points into [0, 1000] × [0, 1000]

points\_6000 = generate\_points(6000, [0, 0, 1000, 1000])

for point in points\_6000:

    quadtree.insert(point)

# Queries and analysis

queries = [np.array([500, 500]), np.array([1000, 1000]), np.array([30, 950]), np.array([0, 1020])]

epsilons = [0.05, 0.1, 0.15, 0.2, 0.25]

results = {}

for query in queries:

    distances = []

    for eps in epsilons:

        neighbor = quadtree.nearest\_neighbor(query, eps)

        distances.append(np.linalg.norm(query - neighbor))

    results[tuple(query)] = distances

# Plot results

for query, distances in results.items():

    plt.plot(epsilons, distances, label=f"Query {query}")

plt.xlabel("Epsilon")

plt.ylabel("Distance to Nearest Neighbor")

plt.title("ANN Distance vs Epsilon (6000 Inserted Points)")

plt.legend()

plt.show()

# Random queries in [0, 1000] × [0, 1000]

random\_queries = generate\_points(1000, [0, 0, 1000, 1000])

start\_time = time.time()

distances = [np.linalg.norm(query - quadtree.nearest\_neighbor(query, eps=0.1)) for query in random\_queries]

average\_time = (time.time() - start\_time) / len(random\_queries)

average\_distance = np.mean(distances)

print(f"Random Queries in [0, 1000] x [0, 1000]: Avg Query Time = {average\_time:.6f}, Avg Distance = {average\_distance:.6f}")

# Insert 2000 random points into [1000, 2000] × [1000, 2000]

points\_2000 = generate\_points(2000, [1000, 1000, 2000, 2000])

for point in points\_2000:

    quadtree.insert(point)

# Random queries in [1000, 2000] × [1000, 2000]

random\_queries\_2 = generate\_points(1000, [1000, 1000, 2000, 2000])

start\_time = time.time()

distances\_2 = [np.linalg.norm(query - quadtree.nearest\_neighbor(query, eps=0.1)) for query in random\_queries\_2]

average\_time\_2 = (time.time() - start\_time) / len(random\_queries\_2)

average\_distance\_2 = np.mean(distances\_2)

print(f"Random Queries in [1000, 2000] x [1000, 2000]: Avg Query Time = {average\_time\_2:.6f}, Avg Distance = {average\_distance\_2:.6f}")

# Comparison

print(f"Comparison of Query Results:")

print(f"[0, 1000] x [0, 1000]: Avg Query Time = {average\_time:.6f}, Avg Distance = {average\_distance:.6f}")

print(f"[1000, 2000] x [1000, 2000]: Avg Query Time = {average\_time\_2:.6f}, Avg Distance = {average\_distance\_2:.6f}")

Dynamic insertions were implemented based totally on a lazy update method, inspired through static-to-dynamic alterations. Points were brought incrementally, with the tree shape adapting to house new partitions when required. This approach allowed for efficient updates without the need for frequent international reconstructions.

Experiments involved the following scenarios:  
1. Inserting 6000 points into [0, 1000] × [0, 1000], followed via ANN queries throughout various ε values.

2. Adding 2000 factors into [1000, 2000] × [1000, 2000] and jogging queries in the extended place.

3. Comparing query overall performance earlier than and after the insertion of recent factors to recognize the effect on question time and accuracy.

### Results and Observations

A graph with colored lines

Description automatically generated

Figure 8: Result-4a



Figure 9: Result-4b



Figure 10: Result-4c

A black background with white text

Description automatically generated

Figure 11: Result-4d

Insertions were processed successfully, with minimum impact on query performance. The quadtree's lazy update mechanism enabled it to deal with the developing dataset without big degradation in efficiency. Queries within the prolonged region showed barely longer times due to the elevated depth required for brand new walls, however the accuracy remained steady.

# Conclusion

This project tested the versatility and scalability of quadtrees in managing dynamic datasets and assisting approximate nearest neighbor queries. Key insights include:

* Spread and peak: The dataset's unfold directly impacts the tree's height and performance. Preprocessing statistics to stability spreads can optimize performance.
* Effect of ε: Larger ε values enhance question velocity but reduce precision, offering a tradeoff based totally on software requirements.
* Dynamic operations: Insertions have been treated successfully thru lazy updates, whilst deletions required greater processing, especially in dense areas. Tree reconstruction proved essential for keeping performance in heavily changed datasets.

Overall, the extended quadtree implementation provides a sturdy and adaptable solution for spatial information control. Its capability to address dynamic updates and approximate queries makes it suitable for applications ranging from geographic records systems to gadget gaining knowledge of and past.