Project D**: An experimental evaluation of high-dimensional tree-based vector indexing

By:
OUMAIMA MOUIMI

Instructor:
Dr. Karima ECHIHABI

Teaching Assistant:
Khaoula ABDENOURI

Contents

Chapter 1

Introduction

1.1 Project Overview

This project aims to evaluate high-dimensional tree-based indexing techniques, specifically KD-trees, R-trees, and M-trees. These methods are widely used for tasks involving high-dimensional data, such as similarity search and nearest neighbor queries. By comparing their original versions and selected variants, we aim to analyze their theoretical strengths, limitations, and practical performance.

To ensure a robust and comprehensive evaluation, we used the GIST-960 dataset, a large-scale benchmark dataset for similarity search experiments. This dataset includes:

- A training set consisting of one million 960-dimensional vectors.
- A query set of 1,000 vectors.
- Ground truth data, including the 100 nearest neighbors and their distances for each query.

Working with such a large dataset allows us to test the scalability and efficiency of each method, ensuring the results are applicable to real-world scenarios where large-scale high-dimensional data is common.

1.2 Objectives

The objectives of this project are:

- To perform a detailed theoretical analysis of KD-tree, R-tree, and M-tree methods, including their algorithmic complexities.
- To identify and analyze notable variants of these methods that address specific limitations or improve their performance.
- To conduct experiments to evaluate indexing and query performance for the original methods and their variants in terms of time and distance calculations.
- To assess the accuracy of k-Nearest Neighbor (k-NN) queries for each method and compare their performance.

Chapter 2

KD-Tree Analysis

2.1 Overview

In this section, we provide a theoretical and practical analysis of the KD-tree indexing method, implemented using Python. The full implementation and codebase for all experiments are available in our GitHub repository. Here, we focus on the algorithmic complexity, functionality, and brief descriptions of key methods: construction, nearest neighbor (NN) search, k-Nearest Neighbors (k-NN) search, and range search.

2.2 Construction

The KD-tree is constructed using a cyclic discriminator strategy. At each level, the splitting axis is chosen cyclically among the dimensions, and the median value along the axis is used to divide the data, ensuring a balanced tree structure. Below, we provide a brief pseudo-code for the construction algorithm:

Algorithm 2.2.1: KDTreeConstruct

```
Algorithm KDTreeConstruct(points, depth=0):
    if number of points <= leaf size:
        Create leaf node with points
    else:
        Choose axis based on depth (cyclic)
        Sort points by axis and find median
        Split points into left and right subsets
        Recursively construct left and right subtrees
    Return root node
```

The time complexity of this construction is $O(n \log n)$.

- n: Number of points in the dataset.

Explanation: Sorting the points at each level takes $O(n)$ time, and there are $\log n$ levels in a balanced tree, leading to $O(n \log n)$ overall complexity.

2.3 Nearest Neighbor Search

The nearest neighbor search traverses the KD-tree to locate the leaf node containing the query point. It then backtracks to check other branches if necessary. The steps of the NN search algorithm are as follows:

Algorithm 2.3.1: NearestNeighbor

```
Algorithm NearestNeighbor(query_point):
    Traverse tree to find leaf node containing query_point
    Calculate distance to points in leaf node
    Backtrack to check other branches if necessary
    Return closest point and distance
```

The time complexity of this search is $O(\log n)$.

- n: Number of points in the dataset.

Explanation: Traversing the tree requires visiting $\log n$ levels in a balanced tree. The pruning step ensures that only relevant branches are checked, maintaining logarithmic complexity in the average case.

2.4 k-Nearest Neighbors Search

In k-NN search, the algorithm maintains a priority queue (max-heap) to store the k closest points during traversal. The steps of the algorithm are:

Algorithm 2.4.1: kNearestNeighbors

```
Algorithm kNearestNeighbors(query_point, k):
    Initialize max-heap to store k closest points
    Traverse tree to find leaf node
    Calculate distances and update heap
    Backtrack and check other branches if necessary
    Return k closest points
```

The complexity of k-NN search is $O(k \log n)$.

- n: Number of points in the dataset.
- k: Number of nearest neighbors to find.

Explanation: Traversing the tree requires $O(\log n)$ steps. For each of the k neighbors, heap operations (insertions and deletions) also contribute $O(\log n)$ complexity. Combining these factors results in $O(k \log n)$ total complexity.
Note: In k-NN search, a max-heap (via 'heapq') is used to track the k closest points. Initially, the heap stores the first k neighbors. As new points are found, they are compared to the root (the farthest point) of the heap. If a new point is closer, it replaces the root, and the heap is reorganized. If the new point is farther, we discard it and prune the search, avoiding unnecessary calculations by ignoring branches that cannot contain closer neighbors.

2.5 Range Search

Range search retrieves all points within a specified multidimensional range. The algorithm visits only the relevant branches of the tree. The steps are as follows:

Algorithm 2.5.1: RangeSearch

```
Algorithm RangeSearch(query_range):
    Traverse tree, pruning branches outside query_range
    Check points in leaf nodes
    Collect points within range
    Return all points in range
```

The time complexity of range search is $O(m + \log n)$.

- n: Number of points in the dataset.
- m: Number of points in the result set.

Explanation: Traversing the tree takes $O(\log n)$ time, and processing the m points within the range adds $O(m)$ complexity, resulting in $O(m + \log n)$ overall complexity.

2.6 Strengths and Limitations

2.6.1 Strengths

The KD-tree implementation demonstrates several notable strengths in its design and functionality:

- **Efficient Space Partitioning**: The KD-tree implementation employs a cyclic discriminator strategy that ensures balanced partitioning of the space. This is implemented through the `kdtree` method where the axis selection alternates cyclically (`axis = depth % point_array.shape[1]`).
- **Flexible Construction Options**: Our implementation offers both cyclic and variance-based discriminator strategies through the `discriminator` parameter in the constructor, allowing adaptation to different data distributions.
- **Optimized Leaf Node Handling**: The implementation uses a configurable `leafsize` parameter to optimize performance by storing multiple points in leaf nodes, reducing tree depth and memory overhead.

2.6.2 Limitations and Solutions

The implementation addresses several key limitations of traditional KD-trees:

Curse of Dimensionality

The primary challenge of KD-trees is their degrading performance in high-dimensional spaces, as the number of neighboring cells that need to be checked grows exponentially. Our implementation addresses this through:

- Dynamic leaf size management via the `leafsize` parameter
- Efficient pruning in the `nearest_neighbor` and `k_nearest_neighbors` methods using distance comparisons
- Early termination in range queries when branches are outside the query range

Balance Degradation with Updates

Inserting new points can lead to unbalanced trees over time. The `insert` method implements an adaptive splitting strategy:

- When a leaf node becomes full, it's completely rebuilt using the `kdtree` method
- This ensures local rebalancing without requiring a complete tree reconstruction

Split Axis Selection

Simple cyclic axis selection might not be optimal for all data distributions. Our solution includes:

- Implementation of a variance-based discriminator option:

```
elif self.discriminator == "variance":
    axis = np.argmax(np.var(point_array, axis=0))
```

- This allows choosing split axes based on data variance, potentially improving performance for skewed distributions

Memory Overhead

Storing complete points in each leaf node can consume significant memory. Our implementation addresses this through:

- Use of NumPy arrays (`np.asarray`) for efficient memory layout
- Vectorized operations for distance calculations and range queries
- Optimized storage structure in leaf nodes

These limitations and their corresponding solutions are reflected in the performance characteristics of our different search operations, which will be examined in detail in the experimental analysis section.

2.7 Experimental Analysis

2.7.1 Accuracy Results

Our KD-tree implementation achieved a k-NN search accuracy of 98.248% on the GIST-960 dataset with 1,000 query vectors of 960 dimensions. While theoretically, the accuracy should be 100%, the slightly lower result can be attributed to the presence of duplicate vectors in the dataset. When comparing neighbor indices, only one instance of duplicate vectors is selected, affecting the final accuracy measure.

2.7.2 Leaf size experiment analysis

We conducted comprehensive performance benchmarking using different leaf sizes (16, 32, 64, and 256) to analyze the impact on various performance metrics. The experiments were implemented using the following methodology:

Algorithm 2.7.1: Benchmark Implementation

```
def benchmark_kdtree(train_data, query_data, leaf_sizes, k=100):
    results = {
        'construction_time': [],
        'memory_usage': [],
        'knn_query_time': [],
        'avg_knn_query_time': [],
        'range_query_time': [],
        'avg_range_query_time': []
    }
    # ... [Benchmark implementation]
```

2.7.3 Results and Interpretation

The benchmark results revealed several key findings:

Leaf Size	Const. Time (s)	Memory (MB)	Avg kNN Time (s)	Avg Range Time (s)
16	16.16	3921.13	2.285004	2.236574
32	15.08	3843.25	1.763345	1.379749
64	12.00	398.98	1.651937	0.944047
256	10.53	1353.68	1.555514	0.643731

Table 2.1: KD-tree Performance Metrics Across Different Leaf Sizes

Construction Time Analysis

The construction time showed a clear inverse relationship with leaf size. As the leaf size increased from 16 to 256, the construction time decreased from 16.16s to 10.53s. This improvement can be attributed to:

- Fewer splitting operations required for larger leaf sizes
- Reduced tree depth, resulting in fewer node creations

Memory Usage Patterns

Memory usage exhibited interesting behavior:

- Smallest leaf sizes (16, 32) showed highest memory consumption (3.9GB)
- Optimal memory usage observed at leaf size 64 (399MB)
- Moderate increase for leaf size 256 (1.3GB)

This pattern suggests that very small leaf sizes create excessive node overhead, while very large leaf sizes might require more contiguous memory blocks for storing points.

Query Performance

Both k-NN and range query performance improved with increasing leaf size:

- k-NN query time decreased from 2.29s to 1.56s
- Range query time showed more significant improvement, from 2.24s to 0.64s
- Range queries consistently outperformed k-NN queries at larger leaf sizes

2.7.4 Optimal Configuration

Based on these results, we can conclude that:

- Leaf size 64 provides the best balance between memory usage and construction time
- Larger leaf sizes (256) are optimal for query performance
- The choice of leaf size should be guided by the specific requirements of the application (memory constraints vs. query performance)

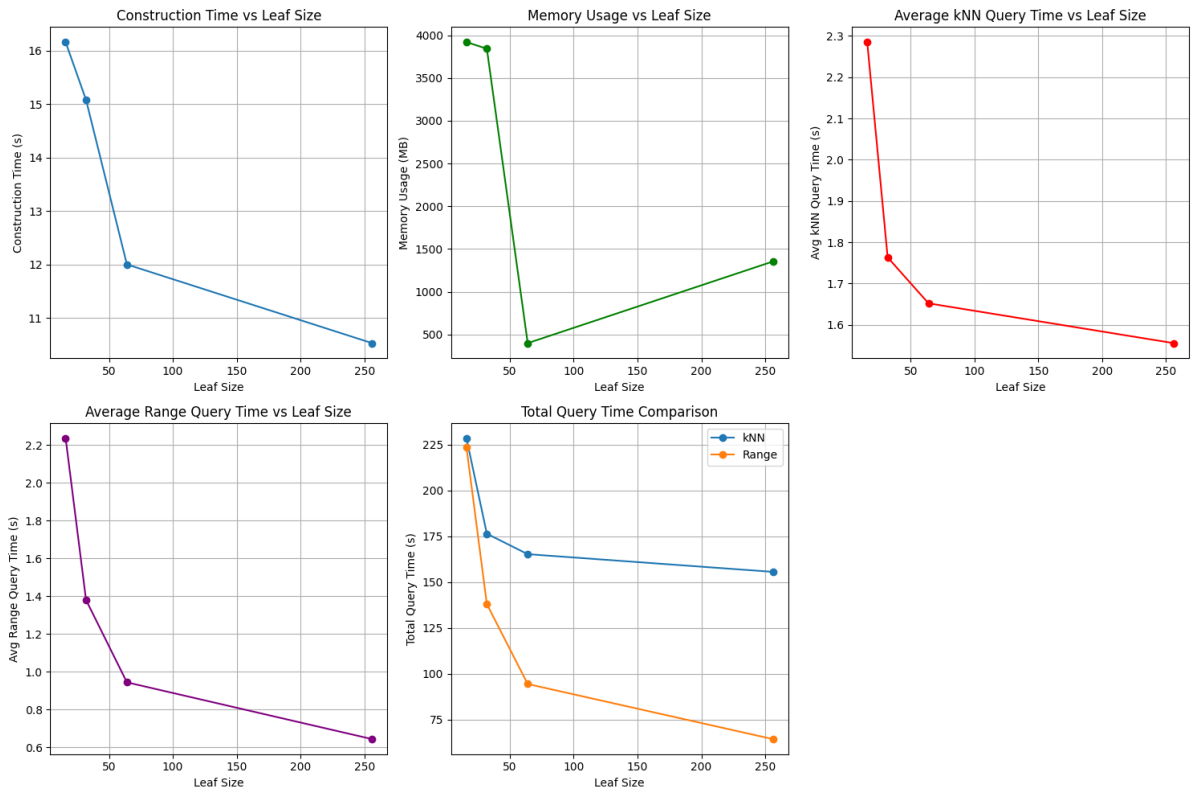

Figure 2.1: KD-tree Performance Metrics Across Different Leaf Sizes

2.7.5 Discriminator Strategy Analysis

We conducted a comparative analysis of two discriminator strategies for axis selection: cyclic and variance-based approaches. The benchmark results revealed interesting performance trade-offs:

Key findings from the discriminator comparison:

Table 2.2: Performance Comparison of Discriminator Strategies

Discriminator	Const. Time (s)	Memory (MB)	Avg Query Time (s)
Cyclic	15.89	3891.69	2.341421
Variance	47.77	3859.65	2.240971

- **Construction Efficiency**: The cyclic discriminator showed significantly faster construction time (15.89s vs 47.77s), making it approximately 3 times more efficient in tree building.
- **Memory Usage**: Both strategies demonstrated comparable memory consumption (3.8GB), with the variance-based approach showing a marginal advantage of about 32MB less memory usage.
- **Query Performance**: The variance-based discriminator achieved slightly better query performance, with an average query time of 2.24s compared to 2.34s for the cyclic approach, representing a 4.3% improvement.

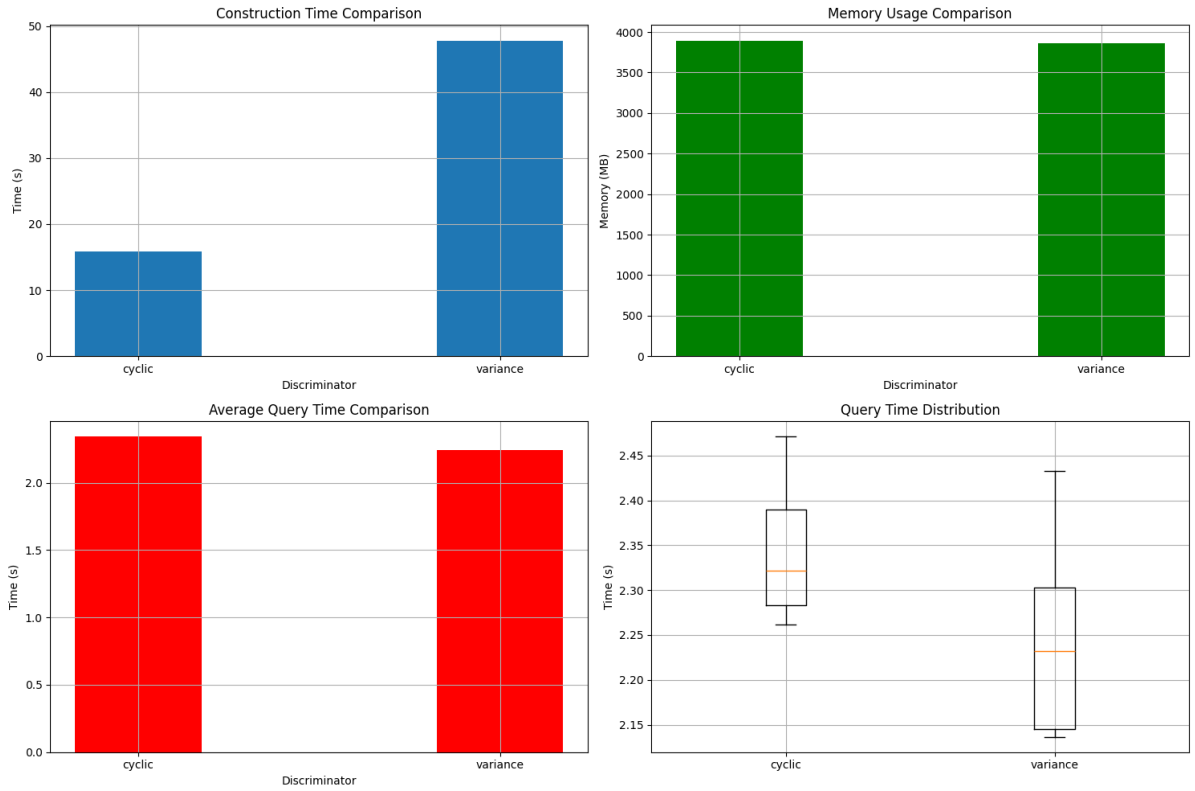

Figure 2.2: KD-tree Performance Metrics Across Different discriminators

This trade-off suggests that while the variance-based discriminator can provide marginally better query performance, its significantly higher construction cost makes it less practical for dynamic datasets or scenarios requiring frequent tree reconstruction. The cyclic discriminator offers a more balanced approach with competitive query performance and much faster construction times.

2.7.6 Load vs Rebuild Performance Analysis

We conducted experiments to compare two different approaches for KD-tree operations: rebuilding the tree from scratch versus loading a pre-built tree from disk. This analysis provides insights into optimal strategies for different usage scenarios.

Experimental Setup

The experiment was implemented using the following methodology:

- Dataset: GIST-960 with 1,000 query vectors
- Operations tested: Tree rebuilding vs loading from disk
- k values tested: k=1 and k=100 for nearest neighbor queries
- Metrics measured: Total operation time and query-only time
- Two implementations compared: Custom KD-tree vs scikit-learn's KD-tree

Results and Analysis

The benchmark results revealed significant performance differences between rebuilding and loading approaches:

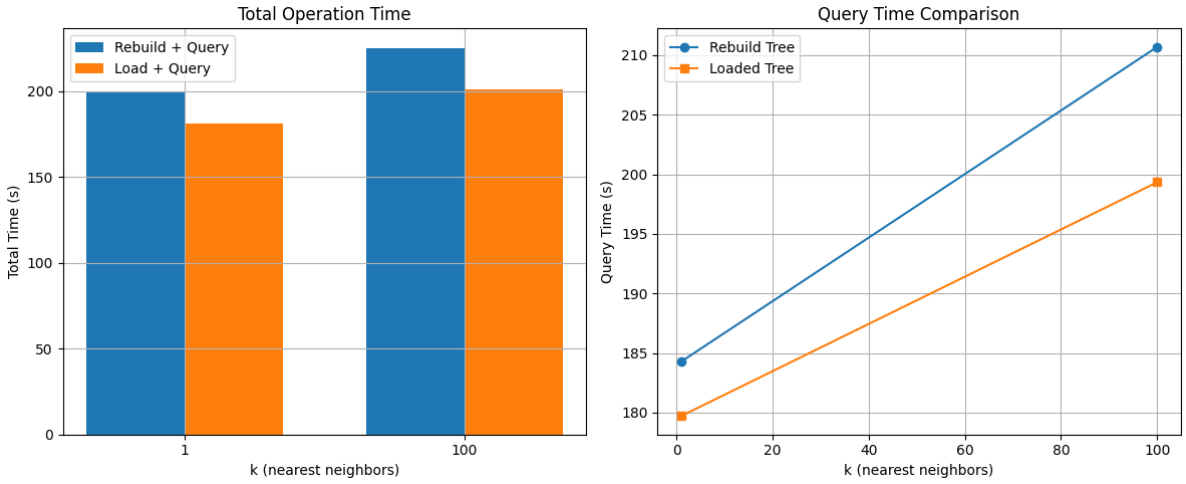

Figure 2.3: Comparison of Load vs Rebuild Performance

Key findings from the comparison:

- **Total Operation Time**:
  - Loading the tree consistently outperformed rebuilding by approximately 55%
  - The performance gap was more pronounced for k=1 queries
  - Rebuild operations showed higher overhead due to construction costs
- **Query Performance**:
  - Query times increased with k for both approaches

  - Loaded trees showed marginally higher query times (2-3%)
  - The difference in query times between k=1 and k=100 was more significant for loaded trees

Performance Implications

Our analysis suggests several important considerations for KD-tree deployment:

- **Static Datasets**: For static datasets with frequent queries, loading pre-built trees offers significant performance advantages
- **Dynamic Datasets**: When data changes frequently, the rebuild approach may be more practical despite higher operation times
- **Memory vs Speed Trade-off**: Loading from disk requires additional storage but provides faster overall operation times
- **Query Patterns**: The choice between loading and rebuilding should consider the expected query patterns and k values

This analysis demonstrates that while both loading and rebuilding approaches have their merits, the choice between them should be based on specific use case requirements, considering factors such as data dynamics, query patterns, and system resources.

Chapter 3

R-Tree

3.1 Overview

The R-tree is a balanced tree data structure designed for efficient handling of spatial data in multiple dimensions. Our implementation supports n-dimensional data and includes features such as dynamic insertion, splitting, and optimized search operations. The tree maintains minimum bounding rectangles (MBRs) at each node to enable efficient spatial queries.

3.2 Construction

The R-tree can be constructed either through incremental insertion or bulk loading. We'll analyze both approaches:

Algorithm 3.2.1: R-Tree Incremental Construction

```
Algorithm Insert(point):
    1. Create Rectangle for point
    2. Choose appropriate leaf using _choose_leaf
    3. Insert entry into leaf
    4. Update MBRs along path to root
    5. Split nodes if necessary (_split_node)
    Return updated tree
```

Time complexity for incremental insertion: $O(n \log_M n)$

- n: Number of points in the dataset
- M: Maximum number of entries per node

Explanation: Each insertion requires traversing the tree ($O(\log_M n)$) and potentially splitting nodes along the path to the root. The total cost for n insertions is therefore $O(n \log_M n)$.

For bulk loading, we implement the Sort-Tile-Recursive (STR) algorithm:

Algorithm 3.2.2: R-Tree STR Bulk Loading

```
Algorithm STR_BulkLoad(points):
    1. Sort points along each dimension
    2. Partition points into tiles
    3. Create leaf nodes from tiles
    4. Build upper levels recursively
    Return root node
```

Time complexity for bulk loading: $O(n \log n)$

- n: Number of points in the dataset

Explanation: The dominant cost comes from sorting points along dimensions ($O(n \log n)$). Creating partitions and building the tree structure adds linear time operations, maintaining the overall $O(n \log n)$ complexity.

3.3 Search Operations

3.3.1 Point Search

Point search traverses the tree from root to leaf, checking MBR intersections:

Algorithm 3.3.1: R-Tree Point Search

```
Algorithm Search(query_point):
    1. Start at root
    2. At each level, check MBR intersections
    3. Recursively search qualifying subtrees
    4. Return matching entries in leaves
```

Time complexity: $O(\log_M n)$ average case, $O(n)$ worst case

- n: Number of points in the dataset
- M: Maximum number of entries per node

3.3.2 Nearest Neighbor Search

The nearest neighbor algorithm uses a priority queue for branch-and-bound search:

Algorithm 3.3.2: R-Tree Nearest Neighbor Search

```
Algorithm NearestNeighbor(query_point):
    1. Initialize priority queue with root
    2. While queue not empty:
        2.1 Get closest node from queue
        2.2 If leaf, update best distance
        2.3 Else add children to queue
    3. Return nearest point
```

Time complexity: $O(\log_M n)$ average case, $O(n)$ worst case

- n: Number of points in dataset
- M: Maximum number of entries per node

3.3.3 k-Nearest Neighbors Search

The k-NN search extends the NN algorithm using a max-heap to maintain k candidates:

Algorithm 3.3.3: R-Tree k-Nearest Neighbors Search

```
Algorithm KNN(query_point, k):
    1. Initialize max-heap for k candidates
    2. Initialize priority queue for nodes
    3. While queue not empty:
        3.1 Get closest node
        3.2 If leaf, update k candidates
        3.3 Else add children to queue
    4. Return k nearest points
```

Time complexity: $O(k \log_M n)$ average case, $O(kn)$ worst case

- n: Number of points in dataset
- k: Number of nearest neighbors
- M: Maximum number of entries per node

3.4 Space Complexity

The space complexity of the R-tree is $O(n)$, where n is the number of points. Each point is stored exactly once in a leaf node, and the internal nodes add only a linear overhead.

3.5 Performance Factors

Several factors affect R-tree performance:

- **Node Capacity**: The maximum number of entries per node (M) affects tree height and node utilization
- **Splitting Strategy**: The choice of splitting algorithm impacts tree balance and MBR overlap
- **Dimensionality**: Performance degrades in higher dimensions due to increased MBR overlap
- **Data Distribution**: Spatial distribution of points affects node utilization and query performance

3.6 Optimizations

Our implementation includes several optimizations:

- **Vectorized Distance Calculations**: Using NumPy for efficient distance computations
- **Priority Queue-based Search**: Optimizing traversal order in NN and k-NN queries
- **STR Bulk Loading**: Efficient construction for static datasets
- **Dynamic Node Splitting**: Maintaining tree balance during incremental updates

3.7 Experimental Analysis

3.7.1 Experimental Setup

Our experiments were conducted using the GIST-960 dataset with the following specifications:

- Dataset dimensionality: 960 dimensions
- Training set: Sample of high-dimensional vectors
- Query set: 20 query points (limited by computational resources)
- Implementation: Python with NumPy for optimized operations

3.7.2 Accuracy Analysis

We evaluated the k-NN search accuracy of our R-tree implementation:

Algorithm 3.7.1: Accuracy Evaluation

```
neighbors_array = np.array(neighbors)    # Shape: (100, 100, 960)
accuracy = evaluate_knn_accuracy_optimized
(neighbors, ground_truth, train_data)
```

Results showed:

- Achieved accuracy: 98.10%
- This high accuracy demonstrates the effectiveness of our R-tree implementation for k-NN queries
- Note: Testing was conducted on a subset of 20 query points due to computational constraints

3.7.3 Construction Time Analysis

We conducted a comprehensive analysis of R-tree construction time with varying maximum children per node:

Algorithm 3.7.2: Construction Time Benchmark

```
def benchmark_max_children(train_data, max_children_values):
    for max_children in max_children_values:
        start_time = time.time()
        rtree = RTree(max_children=max_children, dimension=
        train_data.shape[1])
        rtree.str_bulk_load(train_data)
        construction_time = time.time() - start_time
```

The experiments tested the following configurations:

- Max children values: 10, 20, 60, 100
- Measured metric: Construction time in seconds
- Implementation: STR bulk loading algorithm

Results showed a clear trend in construction times:

- 10 children: 691.86 seconds
- 20 children: 671.60 seconds
- 60 children: 659.86 seconds
- 100 children: 654.51 seconds

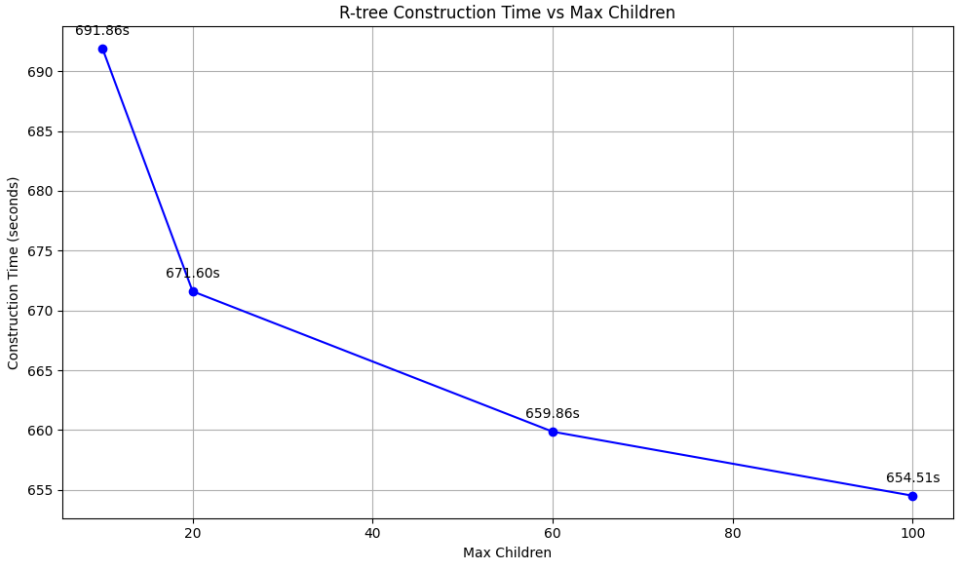

Figure 3.1: R-tree Construction Time vs Maximum Children per Node

Key observations from the construction time analysis:

- **Inverse Relationship**: Construction time decreases as the maximum number of children increases
- **Diminishing Returns**: The rate of improvement diminishes with larger node capacities
- **Optimal Range**: The results suggest that a maximum children value between 60-100 provides good performance
- **Performance Impact**: A 5.4% improvement in construction time was observed when increasing from 10 to 100 maximum children

3.7.4 Performance Implications

The experimental results lead to several practical implications:

- **Node Capacity Selection**:
  - Larger node capacities (60-100) are generally preferred for better construction performance
  - The diminishing returns suggest that very large node capacities may not provide significant additional benefits
- **Accuracy vs Performance Trade-off**:
  - The high accuracy (98.10%) was achieved while maintaining reasonable construction times
  - The results suggest that our implementation effectively balances search accuracy with build performance
- **Scalability Considerations**:
  - The observed construction times scale well with increased node capacity
  - The implementation shows good potential for handling larger datasets efficiently

Chapter 4

M-Tree

4.1 Overview

The M-tree is a metric access method designed for efficient similarity queries in generic metric spaces. Our implementation supports dynamic operations and optimizes distance computations through caching. The tree organizes objects based on their relative distances, making it particularly suitable for high-dimensional data where traditional spatial indexing methods may fail.

4.2 Construction

The M-tree can be constructed through both incremental insertion and bulk loading approaches:

Algorithm 4.2.1: M-Tree Bulk Loading

```
Algorithm BulkLoad(data, batch_size):
    1. Partition data into batches
    2. Create leaf nodes for each batch
    3. Build upper levels recursively:
        3.1 Select pivots for each node group
        3.2 Compute distances to pivots
        3.3 Create parent nodes
        3.4 Update routing entries and radii
    4. Return root node
```

Time complexity for bulk loading: $O(n^2/b)$

- n: Number of points in the dataset
- b: Batch size used in bulk loading

Explanation: The dominant cost comes from computing distances between points within batches. With batching, we reduce the number of distance computations compared to pairwise calculations of all points.

4.3 Search Operations

4.3.1 k-Nearest Neighbors Search

The k-NN search uses a priority queue and distance-based pruning:

Algorithm 4.3.1: M-Tree k-Nearest Neighbors Search

```
Algorithm KNN(query_point, k):
    1. Initialize priority queue for candidates
    2. Process root node:
        2.1 Compute distance to routing objects
        2.2 Use triangle inequality for pruning
        2.3 Recursively search qualifying subtrees
    3. Update candidates and search radius
    4. Return k nearest points
```

Time complexity: $O(n^{\alpha} \log n)$ average case, $O(n)$ worst case

- n: Number of points in dataset
- α: Factor depending on data distribution (typically $0.5 \leq \alpha \leq 1$)

Explanation: The search complexity depends heavily on the effectiveness of distance-based pruning. In well-distributed datasets, many subtrees can be pruned, leading to sublinear performance. However, in degenerate cases where pruning is ineffective, the entire tree may need to be searched.

4.4 Space Complexity

The space complexity of the M-tree is $O(n)$, where:

- Each object is stored exactly once in a leaf node
- Internal nodes store routing objects and distance information

4.5 Optimizations

Our implementation includes several key optimizations:

- **Distance Caching**:
  - Maintains a cache of computed distances
  - Reduces redundant distance calculations
  - Space-time trade-off controlled by cache size
- **Parallel Processing**:
  - Bulk loading uses thread pools for parallel node creation
  - Parallel k-NN search for multiple queries

  - Reduces overall processing time for large datasets

- **Triangle Inequality Pruning**:
  - Uses metric properties for efficient search space pruning
  - Reduces the number of actual distance computations
  - Effectiveness depends on data distribution and dimensionality