GEOMETRIC DATA STRUCTURES FROM THEORY TO APPLICATIONS

Bao Ngo*
Department of Computer Science
University of Manitoba
Winnipeg, Manitoba, Canada
ngot1@myumanitoba.ca

Jared Rost*
Department of Computer Science
University of Manitoba
Winnipeg, Manitoba, Canada
rostj@myumanitoba.ca

1 INTRODUCTION

In the past decade, there has been a surge in interest and tremendous progress made in the fields of Deep Learning (DL) and Machine Learning (ML). ML and DL have become superior technologies for automating complex tasks, uncovering hidden patterns in vast datasets, and powering many aspects of modern society, from web searches to recommendations on e-commerce websites. The contributing factor to the success of DL models is their ability to learn a mapper function $f(x)$ that transforms raw data into a high-dimensional vector representation, known as an embedding, which can be represented as a point in high-dimensional space. Recent advances in DL architecture designs, availability of computing resources and optimization techniques have enabled models to effectively approximate any complex mappers $f(x)$ and construct meaningful embeddings that facilitate various downstream prediction tasks. In Natural Language Processing (NLP), word2vec (Mikolov et al., 2013) is trained to encode each word in the English dictionary into a low-dimensional embedding that well preserves semantic information of words, Figure 1. In chemistry, where data is rarely plentiful as it is generated from hard-to-gather measurements (Kläser et al., 2024), Graph Neural Networks (GNNs) based models (Kläser et al., 2024; Beaini et al., 2024) learned to embed structure and properties of atoms into meaningful high-dimensional representations, which are used to predict molecules' properties.

The advancement in representation learning has provided an effective method to transform raw data into high-dimensional embeddings in a latent space, ensuring that data points with similar properties are close to each other. One application involves leveraging ML models to learn representations of all products on e-commerce markets. When a user searches for a product, the system can search for other related products on our platform according to their learned representation and recommend them to the user. A naive approach to the problem of searching for related products is to compute the similarity between the representation of the query product and that of every product. Unfortunately, the time complexity of the naive approach scales linearly with the number of stored elements, making it infeasible for large-scale datasets. To improve the time complexity of this naive approach, if a set of possible queries is finite, one can store the answer to each possible query in a dictionary keyed by the query; this improvement, however, requires space that scales linearly to the number of points in the datasets. The expensive time and space complexity of naive approaches have stressed the high demand for scalable and efficient similarity search data structures for high-dimensional data.

Inspired by the problem of efficient similarity search on high-dimensional representation obtained from ML, in this report, we aim to investigate geometric data structures, which are a class of data structures designed for modelling spatial data and supporting efficient similarity search queries. To accomplish that, we select the two geometric data structures and study their time and space complexity by performing extensive experiments on large-scale and high-dimensional datasets.

*Equal contribution

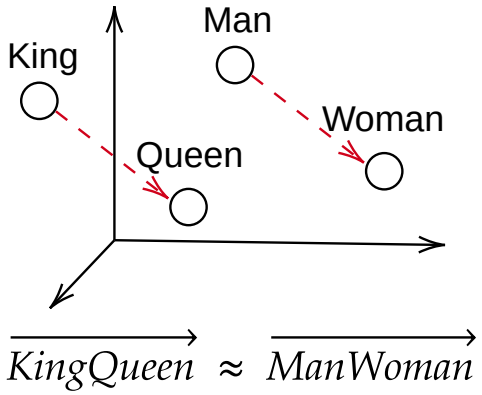


Figure 1: An example of representations of English words. Advancements in ML allow learning representations as high-dimension vectors (embeddings) that preserve semantic information for each word. In this example, the distance and direction of vector $\overrightarrow{KingQueen}$ is relatively the same as vector $\overrightarrow{ManWoman}$

2 Preliminaries and Related Work

Let $\mathcal{P}$ = $\{x_1, x_2, x_3, ..., x_n\}$ denote a set of points such that each point $x_i \in \mathbb{R}^M$,$1 \leq i \leq n$, is the M-dimensional vector of point x_i. Let $n = |\mathcal{P}|$ be the number of points in $\mathcal{P}$. The problem of finding the Nearest Neighbours (NN) and the top K Nearest Neighbours (K-NN) are defined as follows.

Definition 1 (Nearest Neighbours)**.** *Given a query point x_q, $q \in [1, n]$, the nearest neighbours problem is the problem of finding a point that is closest to x_q according to a defined distance metric (e.g., Euclidean distance, Manhattan distance).*

Definition 2 (K Nearest Neighbours)**.** *Given a query point x_q, $q \in [1, n]$, the K nearest neighbours problem is the problem of finding a set of K nearest neighbours to x_q according to a defined distance metric (e.g., Euclidean distance, Manhattan distance).*

Although it is conceptually simple, efficient nearest neighbours search is a non-trivial problem and is studied extensively in data management communities (Chávez et al., 2001; Naidan et al., 2015). Existing nearest neighbours search algorithms can be categorized into exact search algorithms and approximation algorithms.

Various methods have been proposed over the last two decades regarding exact nearest neighbours search algorithms. Most of these methods relied on leveraging space-partitioning to construct a search tree data structure to support exact neighbours search algorithms. The root partition contains the whole search space, and each branch of the tree splits its partition into two sub-partitions, a hierarchy of partitions (Dolatshah et al., 2015). Applying this strategy, Finkel & Bentley proposed the first tree-based geometric data structure, quadtree, where each internal node partitions a 2-D space by recursively subdividing it into four quadrants or regions. Extending quadtree to 3-D spaces, octree (Tamminen & Samet, 1984) recursively subdivides it into eight quadrants or regions. The number of sub-regions divided by each internal node scale exponentially, 2^M, to the dimension M of each point. To address that, Bentley introduced the well-known geometric data structure, K-D tree, where at each depth level, nodes are split into left or right child according to a selected dimension of the point x_i. This improvement made K-D tree work well on moderate dimensional data, but K-D tree still suffers from the "curse of dimensionality" (Malkov & Yashunin, 2020) when working with high dimensional data. K-D tree has shown to eventually visit nearly every node as dimension increases (Yianilos, 1993). Ball-tree is constructed using hyperspheres to partition the space, addressing the limitations of axis-aligned partitioning in high dimensions. The Ball* tree algorithm (Dolatshah et al., 2015) further improves upon traditional ball trees by incorporating Principal Component Analysis (PCA) to identify optimal splitting directions. This approach enables Ball* tree to adapt to the underlying data distribution, resulting in more balanced partitioning and

reduced overlap between nodes. By leveraging the triangle inequality for pruning during searches, Ball* tree significantly reduces the number of distance computations required in high-dimensional spaces, making it more efficient than K-D tree as dimensionality increases (Dolatshah et al., 2015).

Exact solutions for NN and K-NN may offer a substantial search speedup only in case of relatively low to moderate dimensional data and become expensive when working with high dimensional data due to the "curse of dimensionality". To overcome this, an approximate nearest neighbours search has been proposed. By relaxing the condition of exact search, approximate techniques may return non-optimal neighbours in some cases, but can be orders of magnitude faster than exact search. Combining K-D tree random forest and priority search k-means tree algorithm, Muja & Lowe proposed a scalable approximate version of the tree algorithm. Another well-known approach is product quantization (Jégou et al., 2011) that decomposes the search space into low-dimensional subspaces and represents the data sets' joints by compact codes computed as quantization indices in these subspaces. On the other hand, hashing-based nearest neighbours techniques (Andoni & Indyk, 2008; Lv et al., 2007; Bawa et al., 2005) leverage different hash functions to hash similar elements close to each other. The performance of hashing-based nearest neighbours techniques, however, highly depends on the quality of hashing functions. Another nearest neighbours approximate approach is a graph-based strategy that builds a graph of vertices representing points and edges connecting each point to its nearest neighbours (Sebastian & Kimia, 2002; Hajebi et al., 2011; Wang et al., 2012). Despite being faster than exact search, nearest neighbours search methods are not guaranteed to find correct solutions. Hence, in this report, we focus on exact search methods, K-D tree and Ball* tree in particular, which are also well-aligned with the scope of our course.

3 METHODOLOGY

In this report, we focus on evaluating the performance of K-D tree and Ball* tree on five main operations: tree construction, insert, delete, get nearest neighbours and get K-NN. We exclude range query operation of these data structures from this report since, from the ML perspective, the representation of data learned by ML models is mostly not interpretable; therefore, range query operation may not be useful for an ML practitioner. In this section, we briefly discuss the algorithms of all operations of K-D tree, Section 3.1, and Ball* tree, Section 3.2.

3.1 K-D TREE

A K-D tree is a binary search tree where each node in the tree represents a K-dimensional point. The K in K-D tree stands for the dimension of each point x_i that the tree contains. As a result, a M-D tree is a K-D tree consisting of x_i from $\mathcal{P}$ such that $x_i \in \mathbb{R}^M$. In the K-D tree, given a parent node, the discriminator defines the dimension used to split nodes into left or right subtrees at a given node. All nodes at the same depth level have the same discriminator, and the discriminators are different between two consecutive depth levels. Figure 2 illustrates an example of a 2-D tree. In this example, the discriminator of nodes at a depth level d is defined by the depth level modulus of the dimension of points, $d \ mod \ 2$. The root node at a depth level of 0 partitions nodes into two subbranches according to the first dimension. All nodes in the left subtree have the first dimension less than the first dimension of the root node, while the right subtree contains nodes having greater first dimension. Nodes at depth level 1, $(2, 22)$ and $(20, 4)$, with a discriminator of 1, adopt dimension at index 1 to further partition nodes into left and right subtree.

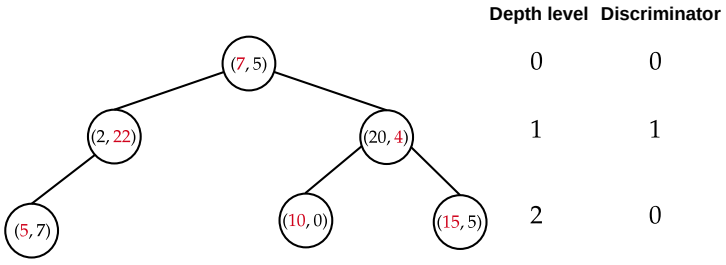


Figure 2: An example of K-D tree for 2 dimensional points (2-d tree)

Tree construction. To construct a K-D tree given an initial set of point $\mathcal{P}$, one naive approach is loop over $\mathcal{P}$ and call insert operation on each point x_i. However, in some circumstances, this may result in an imbalanced binary search tree, which affects the performance of K-D tree in general. To address this issue, Bentley introduced a strategy to construct an optimal K-D tree that is described

in Algorithm 1. Finding the median of a list can be obtained with time complexity of $O(n)$ by using the algorithm proposed by Blum et al.. The number of recursion depth is bounded at $O(log_2 n)$. Hence, the general time complexity of construction operation is $O(nlogn)$.

Insert. Insert on a K-D tree is similar to a binary tree with the time complexity of $O(logn)$. Algorithm 2 illustrates such an insert algorithm, where we keep traversing down and add a new inserting point as a leaf node. During tree traverse, a comparison between the current node and the inserting node at the dimension defined by the discriminator determines whether to traverse to the right or left subtree.

Delete. Detail of the deletion algorithm is described in Algorithm 3. To delete a node x from a given K-D tree, we first need to traverse the tree and find if point x we want to delete exists in the tree. If x exists in the tree, we first replace it with the node having the minimum value on the right subtree or the node having the maximum value on the left subtree, then we recursively delete the node that we replace the target node x with from the subtree where it is found. Similar to insertion, deletion operation can be accomplished in $O(logn)$.

Get K-NN. A full description of the get K-NN operation is provided in Algorithm 4. Given a query point x_q, the main idea of the algorithm is similar to inserting that we need to traverse a given K-D tree from the root node to a leaf node. However, to keep track of the top K nearest neighbours to x_q, we need to maintain a priority queue of nearest points to x_q. Assume that all operations on priority queue can be performed in $O(1)$. Whenever we find a point u having a distance to x_q smaller than the distance between the K^{th} closest point v so far encountered and x_q, we remove v from the queue and add u to the queue, Line 9-11 Algorithm 4. The time complexity to traverse from the root to a leaf node in K-D tree can be done in $O(logn)$.

During traversing the K-D tree, the next subtree to traverse is determined whether left or right according to the comparison as described in the insert algorithm. In addition to that, a test needs to be made to determine if it is necessary to consider another subtree. It is necessary to consider another subtree if the geometric boundaries delimiting those subtrees' nodes overlap the ball centred at the query point with a radius equal to the dissimilarity to the K^{th} closest point so far encountered. This is referred to as the bounds overlap-ball test (Friedman et al., 1977), Line 20 Algorithm 4. If the bounds-overlap-ball test fails, then none of the nodes on the other subtree can be among the K^{th} to the query point x. If the bounds overlap the ball, then the nodes of other subtrees must be considered, and the function is called recursive for the node representation of that subtree (Friedman et al., 1977). Therefore, the get-known's overall time complexity can be scaled up to a multiple of $O(logn)$, $O(klogn)$. Assume the K-D tree is a balance, which can be obtained by using the optimized tree construction method mentioned above, Friedman et al. proved k to be independent of n. As a result, the expected time complexity of getting K-NN is $O(logn)$ and $O(n)$ in the worst-case scenario. Given the algorithm of get K-NN, we can perform the get nearest neighbours operation by simply calling get K-NN with $K = 1$.

3.2 BALL* TREE

A Ball* tree is a binary tree where each node represents a hypersphere (ball) containing a subset of points from $\mathcal{P}$. Each ball node stores a center point, a radius that completely encompasses all contained points, and pointers to left and right child nodes (or the actual points if it's a leaf node). Unlike K-D tree, which uses axis-aligned hyperplane partitioning, Ball* tree partitions space with hyperspheres whose centers can be positioned arbitrarily in the feature space, not constrained to axis-aligned positions. This flexibility allows the structure to better adapt to the underlying data distribution.

The distinction between left and right children is determined through PCA. At each split, the algorithm:

1. Identifies the principal axis (direction of maximum variance) in the data
2. Projects all points onto this axis
3. Places points with projection values less than or equal to the median in the left child
4. Places points with projection values greater than the median in the right child

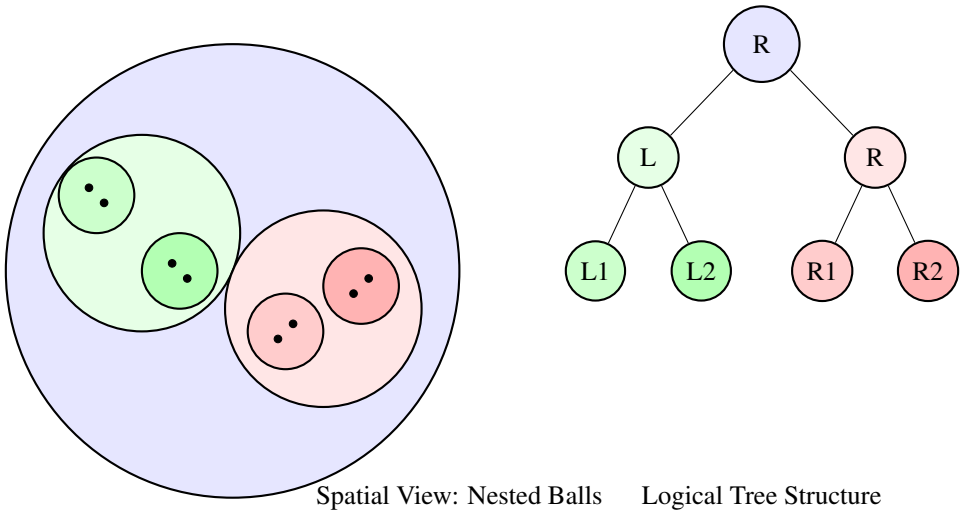


Figure 3: Ball* tree representation showing both spatial organization of points and the corresponding tree structure. The root ball contains all points and is recursively partitioned into smaller balls until leaf nodes are reached.

This creates a hierarchical structure of nested balls where each parent ball completely contains its children's balls. Internal nodes contain two child balls that recursively partition the space, while leaf nodes store up to 50 points directly. 50 was chosen based on the scikit-learn developers implementation of the Ball Tree that uses a default leaf size of 40, slightly increased due to the size of the datasets used in the experiments. Figure 3 illustrates this nested ball hierarchy. This PCA-based splitting approach is particularly effective for high-dimensional data where axis-aligned partitioning becomes inefficient, as the spherical regions can more compactly represent clusters of points regardless of their distribution in the feature space. It also enables efficient search operations through distance-based pruning: during queries, entire subtrees can be skipped when the minimum possible distance from a query point to any point in a node exceeds the current best candidate distance.

Tree construction. To construct a Ball* tree from an initial set of points $\mathcal{P}$, we use a top-down recursive approach as shown in Algorithm 5. Unlike K-D tree which splits along coordinate axes, Ball* tree uses PCA to identify the direction of maximum variance in the data. The algorithm begins by computing the center of all points, then calculates the covariance matrix to find its principal eigenvector. Points are projected onto this principal axis and split at their median projection value, creating balanced left and right subsets. For each node, we compute a bounding radius that encompasses all contained points. If the number of points at any node falls below a predefined leaf size threshold (we used 50), or if the split would result in an empty partition, we create a leaf node storing those points directly. This construction approach adapts to the actual data distribution rather than being constrained to axis-aligned partitioning. The recursion depth is bounded by $O(\log_2 n)$, and since each level requires $O(n)$ work for the PCA computation and partitioning, the overall time complexity is $O(n \log n)$.

Insert. The Ball* tree insertion algorithm has time complexity $O(\log n)$ in the average case. Algorithm 6 shows the process, which begins at the root and recursively traverses to a suitable leaf node. For internal nodes, we update the bounding ball's radius if necessary, and then select the child whose center is closer to the new point. Upon reaching a leaf node, if it contains fewer than `leaf_size` points, we simply add the point and update the ball's center and radius. If the leaf is full, we rebuild the subtree with all points including the new one. This approach maintains the Ball* tree property that each ball completely encompasses all points in its subtree, while ensuring balanced tree growth through occasional rebuilding of densely populated subtrees.

Delete. The Ball* tree deletion algorithm has time complexity $O(\log n)$ in the average case but may degrade to $O(n \log n)$ when restructuring is needed. Algorithm 7 outlines the process, which begins by recursively searching for the target point. When the point is found in a leaf node, it is removed from the node's point list and the ball's center and radius are recalculated. Unlike K-D tree, which

replaces deleted nodes with successors, Ball* tree handles deletions by rebuilding affected subtrees from their remaining points. If a deletion occurs in either child subtree of an internal node, we first collect all remaining points in that subtree and then reconstruct it using the same PCA-based partitioning method used during original construction. This rebuilding approach maintains the Ball* tree's balanced structure but at the cost of higher computational complexity when deleting points near the root.

Lazy Delete. Since the regular Delete operation was too computationally inefficient to be completed in a reasonable amount of time, we also implemented an efficient lazy deletion strategy that significantly reduces the amortized computational cost compared to standard deletion. Algorithm 9 illustrates this approach, which maintains a set of deleted points rather than immediately restructuring the tree. When a point is deleted, it is first verified to exist in the tree, then simply marked as deleted by adding it to a deletion tracking set. Search operations transparently skip these marked points during traversal. To prevent excessive memory overhead, the algorithm tracks the ratio of deleted to total points, triggering a complete tree rebuild only when this ratio exceeds a specified threshold (in the code it is set to 25%, but this was arbitrarily chosen since it was never triggered in our testing as the tree was rebuilt after each delete). This lazy strategy transforms the worst-case $O(n \log n)$ complexity of regular deletion into an amortized $O(1)$ operation in most cases, with occasional $O(n \log n)$ rebuilds. The tradeoff is slightly increased memory usage and potentially reduced search performance as the number of deleted points grows, but for high-dimensional data and frequent modification operations, this approach offers superior overall performance.

Get K-NN. The Ball* tree K-NN search leverages the hierarchical ball structure to efficiently find the K nearest neighbors of a query point x_q, as described in Algorithm 8. The algorithm maintains a max-heap priority queue of the K nearest neighbors found so far. At each node, the search computes a distance lower bound using the triangle inequality: the minimum possible distance between x_q and any point in the ball is $\max(0, d(center, x_q) - radius)$. If this lower bound exceeds the distance to the current K^{th} nearest neighbor, the entire subtree can be safely pruned without examining any contained points, as illustrated in Figure 3. When a leaf node is reached, all contained points are directly compared against the query point and the priority queue is updated accordingly. For internal nodes, the algorithm sorts child balls by their distance lower bounds and visits them in ascending order to maximize pruning opportunities. Unlike K-D tree which uses axis-aligned pruning, Ball*-tree's sphere-based pruning adapts better to high-dimensional data by providing tighter bounds regardless of data orientation. The expected time complexity is $O(\log n)$ for well-distributed data, though it can degrade to $O(n)$ in the worst case when many branches cannot be pruned. The get-nearest operation is simply implemented as get-K-NN with $K = 1$.

4 Experiments Setup

This report studies the performance of two selected geometric data structures from two perspectives. In the first experiment, we investigate the scalability of K-D tree and Ball* tree in terms of time and space complexity when the number of data points n increases while the dimension of each point M is fixed. We conduct the first experiments on three real-world datasets, including Cifar[1], Sun[2], Gauss (Li et al., 2020). In the second experiment, we investigate the "curse of dimension" problem, in particular, how the increase in the dimension of each data point affects the performance of K-D tree and Ball* tree. To accomplish that, we report the time and space complexity of these data structures in the first 50,000 data points from a set of real-world datasets with various sizes of dimensions, including Sift[3], Enron, and Trevi[4]. Further information about datasets used in this report is provided in Appendix A.

For our baseline model, we employ a naive approach that adopts a set data structure to store all points in $\mathcal{P}$. Deletion and insertion operation of our baseline data structure can be done in $O(1)$ on average. In addition, get K-NN operation can be done by computing the distances between query point x_q and all x_i in $\mathcal{P}$, and keeping track of the top K nearest neighbours, resulting in the time complexity $O(n)$.

[1] http://www.cs.toronto.edu/~kriz/cifar.html

[2] http://groups.csail.mit.edu/vision/SUN/

[3] http://corpus-texmex.irisa.fr

[4] http://phototour.cs.washington.edu/patches/default.htm

To evaluate time complexity, we evaluate each operation of K-D tree and Ball* tree using a unified framework of evaluation. We select 200 distinct query points x_i and performed each operation 3 times per point (due to resource limitations, some operations had to be reduced to 1 trial), yielding 600 total measurements. Results are reported as mean execution time with standard deviation across all trials. For the K-nearest neighbors search, we fixed $K = 5$ across all experiments.

In this report, we implement K-D tree and Ball* tree in Python (version 3.11) and utilize Numpy (version 1.26.4) to optimize matrix operations. The computational environment is a Linux cluster equipped with an Intel(R) Xeon(R) Gold 6338, 2.00GHz, processor and 251 GB RAM. Our implementation of K-D tree and Ball* tree is publicly available at `https://github.com/benjaminnNgo/4420`.

5 Results and Discussion

The results of our experiments largely aligned with the theoretical expectations in the literature, particularly regarding the comparative performance of insert, delete, and K-NN operations. However, we observed two noteworthy deviations: dimensionality had less impact than anticipated, while data distribution played a significantly larger role.

Theoretical Expectations. Theoretically, Ball* tree should require more computational resources for construction, insertion, and deletion operations, but offer superior performance for K-NN and nearest-neighbor queries compared to K-D tree and the Brute Force approach. Our experimental results confirm these expectations across datasets of varying sizes and dimensionalities. The Ball* tree consistently underperforms both the Brute Force approach and K-D tree for modification operations—often by substantial margins. For insertions and construction, Ball* tree is typically 10-100 times slower as seen in Figures 4c, 5c, 4a, and 5a, while the deletion operation proves so computationally intensive that we implement a Lazy Delete strategy as an alternative which is seen in Figures 4d and 5d. The Ball* tree's poor delete performance stems from its need to rebuild sub-trees to maintain its structure, as seen in line 32 of Algorithm 7.

K-D tree performs comparably to the Brute Force approach for insertions and deletions as seen in Figures 4c, 5c, 4d, and 5d while requiring approximately three times more time for construction, as seen in Figures 4a and 5a. For search operations, results align with theoretical predictions—the Ball* tree demonstrates superior performance across all datasets as seen in Figures 4e, 5e, 4f, and 5f, with the advantage becoming more pronounced as dataset size increased. K-D tree significantly outperformed the Brute Force approach, particularly as dimensionality or dataset size grew.

Scale Dimensionality. Figure 5 shows the performance of insert, tree construction, delete, space, get K-NN and get NN operations as dimensionality increases while keeping the number of points the same; enabling us investigate the "curse of dimensionality". We tested Sift with 128 dimensions, Enron with 1,369 dimensions, and Trevi with 4,096 dimensions. All three of the datasets had 50,000 points. Contrary to theoretical expectations, dimensionality does not yield the anticipated performance divergence between data structures. While literature suggests Ball* tree should demonstrate increasingly significant performance advantages over K-D tree in the K-NN and nearest operations as dimensionality increases, our experiments show only marginal improvements across all dimensionalities such as in Figures 5e and 5f. Meanwhile, Ball* tree insert, construction, and delete gets significantly worse with larger dimensionalities as seen in Figures 5a, 5d, and 5c; which is expected given the complex matrix operations needed to balance the Ball* tree.

Scale Size. Figure 4 shows the performance of insert, tree construction, delete, space, get K-NN and get NN operations as the number of points increases while keeping the dimensionality the same. The experiment is conducted on three following datasets: Cifar with 50,000 points, Sun with 79,000 points, and Gauss with 2,000,000 points. All three of the datasets had 512 dimensions. Unlike the experiments on varying dimensionality, our results using different dataset sizes align more closely with our expectations. As the dataset size increased, a larger performance gap emerges between Ball* tree and K-D tree for both K-NN and nearest neighbor queries. In our largest dataset, Ball* tree outperforms K-D tree by roughly a factor of 7, while in the smallest dataset, the performance advantage is only about 1.5 times, as illustrated in Figures 4e and 4f. Similar to our dimensionality results, as the dataset size increases, the construction, insertion, and deletion operations of Ball* tree degrades more rapidly than those of K-D tree, as shown in Figures 4a, 4d, and 4c.

Variability. Regarding variability, K-D tree exhibits moderate inconsistency in search operations (K-NN and Nearest) on high-size datasets. This aligns with expectations, as its traversal patterns depend on query-dependent backtracking that could be worsened by having more branches as seen on line 28 of Algorithm 8. Ball* tree shows significant variability in modification operations like in Figures 4c, 5c, 4d, and 5d, likely due to how it makes the most use of multi-threading in its operations and is thus influenced by the availability of resources. The Brute Force implementation shows some variance in high-dimensionality datasets like Figures 5f and 5e, which is harder to explain. While there is minor randomness inherent to Python sets and different queries can cause slightly different priority queue updates, that should not cause the differences we are seeing.

Dataset Distribution. Perhaps most intriguing are the counterintuitive results with Ball* tree on our Gaussian dataset. Despite the Gauss dataset (2,000,000 points) being substantially larger than Cifar (50,000) and Sun (79,000), Ball* tree demonstrates better space efficiency and construction performance than expected on Gauss, as seen on Figures 4a and 4b. While some performance gains can be explained by multi-threading randomness, it is possible the structure of the data played a role. Gauss is our only synthetic dataset, which we generate from a modified normal distribution, while the other datasets were images or text data. The Gaussian distribution provides ideal conditions for the PCA-based splitting algorithm used in our Ball* tree implementation, resulting in well-balanced, efficiently-packed nodes. In contrast, the more irregular distributions in image and text datasets created suboptimal partitioning and partially filled nodes, reducing efficiency.

In conclusion, our experimental results largely confirm theoretical expectations while highlighting important nuances: the impact of dimensionality is less significant than predicted, while data distribution characteristics prove unexpectedly influential in determining real-world performance. Furthermore, the modest impact of dimensionality and the scalability of K-D tree and Ball* tree, as demonstrated by our empirical results, highlight the potential of geometric data structures in ML and DL applications. Given meaningful representations of raw data learned by ML and DL models, geometric data structures can be effectively leveraged for NN and K-NN searches, facilitating various downstream tasks such as labelling unknown data and identifying similar products in recommend systems.

ACKNOWLEDGMENTS

This research project uses the ICLR 2021 Conference submission template for formal formatting. We would like to thank Dr. Jennifer Vaughan for her guidance and her constructive feedbacks on our manual script.

REFERENCES

Alexandr Andoni and Piotr Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. *Commun. ACM*, 51(1):117–122, 2008. doi: 10.1145/1327452.1327494. URL https://doi.org/10.1145/1327452.1327494.

Mayank Bawa, Tyson Condie, and Prasanna Ganesan. LSH forest: self-tuning indexes for similarity search. In Allan Ellis and Tatsuya Hagino (eds.), *Proceedings of the 14th international conference on World Wide Web, WWW 2005, Chiba, Japan, May 10-14, 2005*, pp. 651–660. ACM, 2005. doi: 10.1145/1060745.1060840. URL https://doi.org/10.1145/1060745.1060840.

Dominique Beaini, Shenyang Huang, Joao Alex Cunha, Zhiyi Li, Gabriela Moisescu-Pareja, Oleksandr Dymov, Samuel Maddrell-Mander, Callum McLean, Frederik Wenkel, Luis Müller, Jama Hussein Mohamud, Ali Parviz, Michael Craig, Michal Koziarski, Jiarui Lu, Zhaocheng Zhu, Cristian Gabellini, Kerstin Klaser, Josef Dean, Cas Wognum, Maciej Sypetkowski, Guillaume Rabusseau, Reihaneh Rabbany, Jian Tang, Christopher Morris, Mirco Ravanelli, Guy Wolf, Prudencio Tossou, Hadrien Mary, Therence Bois, Andrew W. Fitzgibbon, Blazej Banaszewski, Chad Martin, and Dominic Masters. Towards foundational models for molecular learning on large-scale multi-task datasets. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net, 2024. URL https://openreview.net/forum?id=Zc2aIcucwc.

Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, 1975. doi: 10.1145/361002.361007. URL https://doi.org/10.1145/361002.361007.

Manuel Blum, Robert W. Floyd, Vaughan R. Pratt, Ronald L. Rivest, and Robert Endre Tarjan. Time bounds for selection. *J. Comput. Syst. Sci.*, 7(4):448–461, 1973. doi: 10.1016/S0022-0000(73)80033-9. URL https://doi.org/10.1016/S0022-0000(73)80033-9.

Edgar Chávez, Gonzalo Navarro, Ricardo A. Baeza-Yates, and José L. Marroquín. Searching in metric spaces. *ACM Comput. Surv.*, 33(3):273–321, 2001. doi: 10.1145/502807.502808. URL https://doi.org/10.1145/502807.502808.

Mohamad Dolatshah, Ali Hadian, and Behrouz Minaei-Bidgoli. Ball*-tree: Efficient spatial indexing for constrained nearest-neighbor search in metric spaces. *CoRR*, abs/1511.00628, 2015. URL http://arxiv.org/abs/1511.00628.

Raphael A. Finkel and Jon Louis Bentley. Quad trees: A data structure for retrieval on composite keys. *Acta Informatica*, 4:1–9, 1974. doi: 10.1007/BF00288933. URL https://doi.org/10.1007/BF00288933.

Jerome H. Friedman, Jon Louis Bentley, and Raphael A. Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Trans. Math. Softw.*, 3(3):209–226, 1977. doi: 10.1145/355744.355745. URL https://doi.org/10.1145/355744.355745.

Kiana Hajebi, Yasin Abbasi-Yadkori, Hossein Shahbazi, and Hong Zhang. Fast approximate nearest-neighbor search with k-nearest neighbor graph. In Toby Walsh (ed.), *IJCAI 2011, Proceedings of the 22nd International Joint Conference on Artificial Intelligence, Barcelona, Catalonia, Spain, July 16-22, 2011*, pp. 1312–1317. IJCAI/AAAI, 2011. doi: 10.5591/978-1-57735-516-8/IJCAI11-222. URL https://doi.org/10.5591/978-1-57735-516-8/IJCAI11-222.

Hervé Jégou, Matthijs Douze, and Cordelia Schmid. Product quantization for nearest neighbor search. *IEEE Trans. Pattern Anal. Mach. Intell.*, 33(1):117–128, 2011. doi: 10.1109/TPAMI.2010.57. URL https://doi.org/10.1109/TPAMI.2010.57.

Kerstin Kläser, Blazej Banaszewski, Samuel Maddrell-Mander, Callum McLean, Luis Müller, Ali Parviz, Shenyang Huang, and Andrew W. Fitzgibbon. Minimol: A parameter-efficient foundation model for molecular learning. *CoRR*, abs/2404.14986, 2024. doi: 10.48550/ARXIV.2404.14986. URL https://doi.org/10.48550/arXiv.2404.14986.

Wen Li, Ying Zhang, Yifang Sun, Wei Wang, Mingjie Li, Wenjie Zhang, and Xuemin Lin. Approximate nearest neighbor search on high dimensional data - experiments, analyses, and improvement. *IEEE Trans. Knowl. Data Eng.*, 32(8):1475–1488, 2020. doi: 10.1109/TKDE.2019.2909204. URL https://doi.org/10.1109/TKDE.2019.2909204.

Qin Lv, William Josephson, Zhe Wang, Moses Charikar, and Kai Li. Multi-probe lsh: efficient indexing for high-dimensional similarity search. In *Proceedings of the 33rd international conference on Very large data bases*, pp. 950–961, 2007.

Yury A. Malkov and Dmitry A. Yashunin. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE Trans. Pattern Anal. Mach. Intell.*, 42(4): 824–836, 2020. doi: 10.1109/TPAMI.2018.2889473. URL https://doi.org/10.1109/TPAMI.2018.2889473.

Tomás Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. In Yoshua Bengio and Yann LeCun (eds.), *1st International Conference on Learning Representations, ICLR 2013, Scottsdale, Arizona, USA, May 2-4, 2013, Workshop Track Proceedings*, 2013. URL http://arxiv.org/abs/1301.3781.

Marius Muja and David G. Lowe. Scalable nearest neighbor algorithms for high dimensional data. *IEEE Trans. Pattern Anal. Mach. Intell.*, 36(11):2227–2240, 2014. doi: 10.1109/TPAMI.2014.2321376. URL https://doi.org/10.1109/TPAMI.2014.2321376.

Bilegsaikhan Naidan, Leonid Boytsov, and Eric Nyberg. Permutation search methods are efficient, yet faster search is possible. *Proc. VLDB Endow.*, 8(12):1618–1629, 2015. doi: 10.14778/2824032.2824059. URL http://www.vldb.org/pvldb/vol8/p1618-naidan.pdf.

scikit-learn developers. sklearn.neighbors.BallTree. scikit-learn 1.2 documentation, 2022. URL https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.BallTree.html. Accessed: April 2, 2025.

Thomas B. Sebastian and Benjamin B. Kimia. Metric-based shape retrieval in large databases. In *16th International Conference on Pattern Recognition, ICPR 2002, Quebec, Canada, August 11-15, 2002*, pp. 291–296. IEEE Computer Society, 2002. doi: 10.1109/ICPR.2002.1047852. URL https://doi.org/10.1109/ICPR.2002.1047852.

Markku Tamminen and Hanan Samet. Efficient octree conversion by connectivity labeling. In Hank Christiansen (ed.), *Proceedings of the 11th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH 1984, Minneapolis, Minnesota, USA, July 23-27, 1984*, pp. 43–51. ACM, 1984. doi: 10.1145/800031.808576. URL https://doi.org/10.1145/800031.808576.

Jing Wang, Jingdong Wang, Gang Zeng, Zhuowen Tu, Rui Gan, and Shipeng Li. Scalable k-nn graph construction for visual descriptors. In *2012 IEEE Conference on Computer Vision and Pattern Recognition*, pp. 1106–1113. IEEE, 2012.

Peter N. Yianilos. Data structures and algorithms for nearest neighbor search in general metric spaces. In Vijaya Ramachandran (ed.), *Proceedings of the Fourth Annual ACM/SIGACT-SIAM Symposium on Discrete Algorithms, 25-27 January 1993, Austin, Texas, USA*, pp. 311–321. ACM/SIAM, 1993. URL http://dl.acm.org/citation.cfm?id=313559.313789.

Appendix

A DATASETS

In this section, we provide further information about datasets used in this report and their statistic. Complete statistics about the datasets are described in Table 1.

Table 1: Dataset statistics.

Name	n $(\times 10^3)$	M	**Type**
Cifar	50	512	Image
Sun	79	512	Image
Enron	95	1,369	Text
Trevi	100	4,096	Image
Sift	994	128	Image
Gauss	2,000	512	Synthetic

Cifar is a labeled subset of **TinyImage** dataset, which consists of 60000 32 $\times$ color images in 10 classes, with each image represented by a 512-d GIST feature vector.

Sun397 contains about 0.08 million 512-d GIST features of images.

Enron origins from a collection of emails. yifang et. al. extract bi-grams and form feature vectors of 1369 dimensions.

Trevi consists of 0.4 million $\times$ 1024 bitmap(.bmp) images, each containing a 16 $\times$ 16 array of image patches. Each patch is sampled as 64 $\times$ 64 grayscale, with a canonical scale and orientation. Therefore, Trevi patch dataset consists of around 100,000 4096-d vectors.

Sift consists of 1 million 128-d SIFT vectors.

Gauss consists of 2,000,000 points generated by randomly choosing 1000 cluster centers with in space $[0, 10]^{512}$, and each cluster follows the a Gaussian distribution with deviation 1 on each dimension.

B DETAIL ALGORITHMS

B.1 K-D TREE

Algorithm 1 Construct

```
function CONSTRUCT(P,δ,M)
    // list of points P
    // current depth level δ
    // dimension of each point M
    if P is empty then
        return None
    end if
    d ← δ mod M // Compute discriminator
    pivot ← find median of P according to dimesion d
    left ← all points in P that are less than pivot at dimension d
    right ← all points in P that are greater than pivot at dimension d
    node ← initialize a tree node of pivot point
    node.left ← Construct(left,δ + 1, M)
    node.right ← Construct(right,δ + 1, M)
    return node
end function
```

Algorithm 2 Insert

```
function INSERT(r,x,δ)
    // current node r
    // an insert point x
    // current depth level δ
    if r is None then
        // Tree is empty
        r ← initialize a tree node of new node x
        return
    end if
    d ← δ mod M // Compute discriminator
    dx ← x[d] - r[d]
    if dx > 0 then
        if r.right is None then
            r.right ← initialize a tree node of new node x
        else
            Insert(r.right,x, δ + 1 )
        end if
    else
        if r.left is None then
            r.left ← initialize a tree node of new node x
        else
            Insert(r.left,x, δ + 1 )
        end if
    end if
end function
```

Algorithm 3 Delete

```
1: function DELETE(r,x,δ)
2:     // current node r
3:     // a point x
4:     // a point x
5:     // current depth level δ
6:     d ← δ mod M // Compute discriminator
7:     if x = r.coordinate then
8:         // Found the target node
9:         if r.right is None and r.left is None then
10:            return
11:        end if
12:        if r.right is not None then
13:            min ← find the δ-minimum node in the right branch, r.right
14:            r.coordinate = min.coordiate // replace root with j-min from right subtree
15:            r.right = Delete(r.right,min.coordiate,δ + 1)
16:        else
17:            max ← find the δ-maximum node in the left branch, r.left
18:            r.coordinate = max.coordiate // replace root with δ-max from left subtree
19:            r.left = Delete(r.left,max.coordiate,δ + 1)
20:        end if
21:    else
22:        if x[d] ≥ r.coordinate[d] then
23:            r.right = Delete(r.right,x,δ + 1)
24:        else
25:            r.left = Delete(r.left,x,δ + 1)
26:        end if
27:    end if
28: end function
```

Algorithm 4 Get K-NN

```
1: function GET K-NN(x, r, k, prQueue, δ ,M, f )
2:     // query point x
3:     // current node r
4:     // number of nodes return k
5:     // current priority queue (initialize with empty at the beginning) prQueue
6:     // current depth level δ
7:     // dimension of each point M
8:     // distance function f
9:     if r is None then
10:        return None
11:    end if
12:    d ← δ mod M // Compute discriminator
13:    distance = f(x, r.coordinate)
14:    if length(prQueue) < k then
15:        prQueue.enqueue((distance,r.coordinate)) // Insert new point
16:    else if distance < prQueue.peek()[0] then
17:        priQueue.dequeue() // Remove the farthest point in the queue
18:        prioQueue.enqueue((distance,r.coordinate)) // Insert new point that is closer to x
19:    end if
20:    if x < r.coordinate[d] then
21:        next = r.left
22:        other = r.right
23:    else
24:        next = r.right
25:        other = r.left
26:    end if
       K-NN(x,next,k, prQueue, δ + 1, M, f)
27:    if |x[d] − r.coordinate[d]| < prQueue.peek()[0] then
28:        K-NN(x,other,k, prQueue, δ + 1, M, f)
29:    end if
30:    return prQueue
31: end function
```

B.2 BALL* TREE

Algorithm 5 Construct

```
function CONSTRUCTTREE(points, depth)
    // points: List of n-dimensional points to organize in the tree
    // depth: Current recursion depth (used for dimension cycling)
    if points is empty then
        return None
    end if
    if |points| ≤ leaf_size then
        center ← mean(points)
        radius ← max_{p∈points} sqrt(d(center, p))
        return BallNode(center, radius, points=points)
    end if
    center ← mean(points)
    cov_matrix ← covariance(points)
    eigenvalues, eigenvectors ← eigendecomposition(cov_matrix)
    if max(eigenvalues) < threshold then // Degenerate case: low variance in all dimensions
        radius ← max_{p∈points} sqrt(d(center, p))
        return BallNode(center, radius, points=points)
    end if
    principal_axis ← eigenvector with largest eigenvalue
    projections ← [dot(p - center, principal_axis) for p in points]
    median_proj ← median(projections)
    left_points ← [p for p, proj in zip(points, projections) if proj ≤ median_proj]
    right_points ← [p for p, proj in zip(points, projections) if proj > median_proj]
    if left_points is empty or right_points is empty then
        radius ← max_{p∈points} sqrt(d(center, p))
        return BallNode(center, radius, points=points) // Force leaf when split fails
    end if
    radius ← max_{p∈points} sqrt(d(center, p))
    left_child ← CONSTRUCTTREE(left_points, depth+1)
    right_child ← CONSTRUCTTREE(right_points, depth+1)
    return BallNode(center, radius, left=left_child, right=right_child)
end function
```

Algorithm 6 Insert

```
function INSERT(node, point, depth)
    // node: Current node in the recursion (initially the root)
    // point: New point to be inserted into the tree
    // depth: Current recursion depth (used for tree reconstruction)
    if node is None then
        return new BallNode(center=point, radius=0, points=[point]) // Create new leaf
    end if
    if node is Leaf then // Node has points array
        if |node.points| < leaf_size then
            node.points.append(point) // Add point to non-full leaf
            node.center ← mean(node.points) // Update center
            node.radius ← max_{p∈node.points} √d(node.center, p) // Update radius
            return node
        else
            all_points ← node.points ∪ {point} // Leaf is full
            return ConstructTree(all_points, depth) // Split leaf into subtree
        end if
    end if
    dist ← √d(node.center, point) // Distance from center to new point
    if dist > node.radius then
        node.radius ← dist // Expand sphere if needed
    end if
    left_dist ← √d(node.left.center, point) // Distance to left child
    right_dist ← √d(node.right.center, point) // Distance to right child
    if left_dist ≤ right_dist then
        node.left ← INSERT(node.left, point, depth+1) // Recurse into closer child
    else
        node.right ← INSERT(node.right, point, depth+1)
    end if
    return node // Return updated node
end function
```

Algorithm 7 Delete

```
function DELETE(point)
    // point: The n-dimensional point to delete from the tree
    node, deleted ← DELETEHELPER(root, point, 0)
    if not deleted then
        throw "Point not found in the Ball*-tree"
    end if
end function
function DELETEHELPER(node, point, depth)
    // node: Current node in the recursion
    // point: The point to be deleted
    // depth: Current recursion depth (used for tree reconstruction)
    if node is None then
        return None, False
    end if
    if node is Leaf then // Node has points array
        if point exists in node.points then
            node.points ← node.points \ {point} // Remove point
            if node.points is empty then
                return None, True // Delete empty node
            end if
            node.center ← mean(node.points)
            node.radius ← max_{p∈node.points} sqrt(d(node.center, p))
            return node, True // Return updated node and success
        end if
        return node, False // Point not found in this leaf
    end if
    left_deleted ← False
    node.left, left_deleted ← DELETEHELPER(node.left, point, depth+1)
    if not left_deleted then
        node.right, left_deleted ← DELETEHELPER(node.right, point, depth+1)
    end if
    if left_deleted then
        all_points ← COLLECTPOINTS(node) // Gather all remaining points
        if all_points is not empty then
            return CONSTRUCTTREE(all_points, depth), True // Rebuild subtree
        else
            return None, True // No points remain
        end if
    end if
    return node, False // Point not found in this subtree
end function
```

Algorithm 8 Get K-NN

```
function GetKNN(point, k)
    // point: The query point for which to find neighbors
    // k: Number of nearest neighbors to retrieve
    pq ← EmptyMaxHeap() // Priority queue (-dist, tiebreaker, point)
    pq, _ ← GetKNNHelper(point, root, k, pq, 0, 1)
    knn ← [(−d, pt) for d, _, pt in pq]
    Sort knn by distance
    return [pt for _, pt in knn]
end function
function GetKNNHelper(point, node, k, pq, depth, tiebreaker)
    // point: The query point for nearest neighbor search
    // node: Current node being processed in the recursion
    // k: Number of nearest neighbors to find
    // pq: Max-heap priority queue of current best candidates
    // depth: Current recursion depth
    // tiebreaker: Value to resolve equal distances
    if node is None then
        return pq, tiebreaker
    end if
    center_dist ← √d(node.center, point)
    lower_bound ← max(0, center_dist − node.radius) // Minimum possible distance to any point in node
    if |pq| = k and lower_bound ≥ −pq[0][0] then
        return pq, tiebreaker // Prune: all points in this node are farther than current kth best
    end if
    if node is Leaf then
        for all pt in node.points do
            dist ← √d(pt, point)
            if |pq| < k then
                pq.Push((−dist, tiebreaker, pt)) // Priority queue not yet full
                tiebreaker ← tiebreaker + 1
            else if dist < −pq[0][0] then
                pq.PopAndPush((−dist, tiebreaker, pt)) // Replace current worst point
                tiebreaker ← tiebreaker + 1
            end if
        end for
        return pq, tiebreaker
    end if
    children ← EmptyList()
    if node.left is not None then
        left_dist ← √d(node.left.center, point)
        left_lb ← max(0, left_dist − node.left.radius)
        children.Append((left_lb, node.left))
    end if
    if node.right is not None then
        right_dist ← √d(node.right.center, point)
        right_lb ← max(0, right_dist − node.right.radius)
        children.Append((right_lb, node.right))
    end if
    Sort children by lower_bound // Process most promising nodes first for better pruning
    for all (lb, child) in children do
        pq, tiebreaker ← GetKNNHelper(point, child, k, pq, depth+1, tiebreaker)
    end for
    return pq, tiebreaker
end function
```

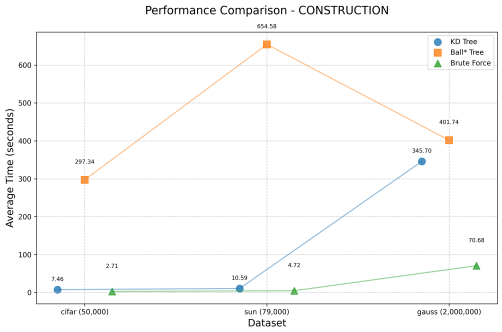
Algorithm 9 Lazy Delete in Ball*-Tree

```
function LazyDelete(point)
    // point: The n-dimensional point to mark as deleted
    point_tuple ← PointToTuple(point) // Convert to hashable tuple
    if point_tuple ∈ deleted_points then
        return true // Already marked as deleted
    end if
    found ← CheckPointExists(root, point)
    if not found then
        throw "Point not found in the Ball*-tree"
    end if
    deleted_points ← deleted_points ∪ {point_tuple}
    deleted_count ← deleted_count + 1
    total_points ← |CollectAllPoints(root)|
    if deleted_count > total_points × rebuild_threshold then
        Cleanup // Rebuild tree if too many deleted points
    end if
    return true
end function
```

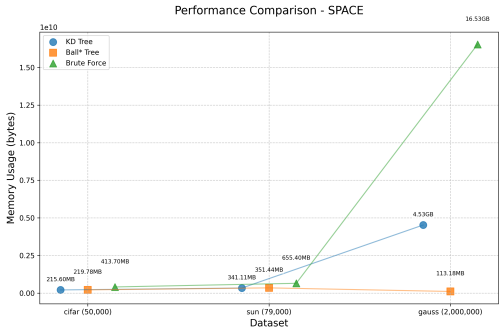
Algorithm 10 Lazy Delete in Ball*-Tree Helper Functions

```
function CHECKPOINTEXISTS(node, point)
    // node: Current tree node being checked
    // point: Point to search for in the tree
    if node = null then
        return false
    end if
    if node.points ≠ null then // Leaf node
        return ∃ pt ∈ node.points : pt = point
    end if
    left_dist ← ∞ if node.left = null else √d(point, node.left.center)
    right_dist ← ∞ if node.right = null else √d(point, node.right.center)
    if node.left ≠ null and left_dist ≤ node.left.radius then
        if CHECKPOINTEXISTS(node.left, point) then
            return true
        end if
    end if
    if node.right ≠ null and right_dist ≤ node.right.radius then
        if CHECKPOINTEXISTS(node.right, point) then
            return true
        end if
    end if
    return false // Point not found in this subtree
end function
function CLEANUP // Rebuilds the tree without the deleted points
    all_points ← COLLECTALLPOINTS(root) // Get non-deleted points
    root ← CONSTRUCTTREE(all_points, 0)
    deleted_points ← ∅
    deleted_count ← 0
end function
function COLLECTALLPOINTS(node) // node: Current node to collect points from
    if node = null then
        return ∅
    end if
    points ← ∅
    if node.points ≠ null then
        for all pt ∈ node.points do
            if POINTTOTUPLE(pt) ∉ deleted_points then
                points ← points ∪ {pt} // Keep only non-deleted points
            end if
        end for
    else
        points ← points ∪ COLLECTALLPOINTS(node.left)
        points ← points ∪ COLLECTALLPOINTS(node.right)
    end if
    return points
end function
```

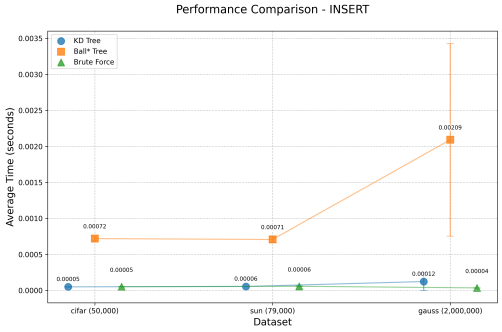
C DETAILED RESULTS



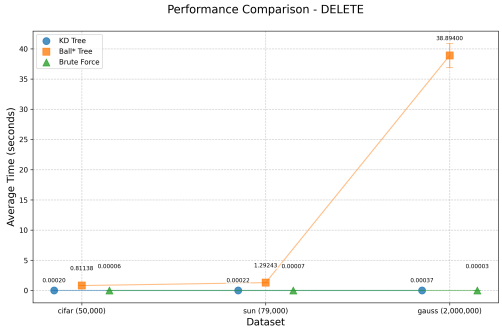
(a) Construction Time



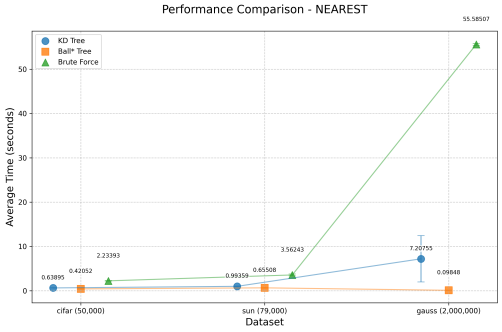
(b) Memory Usage



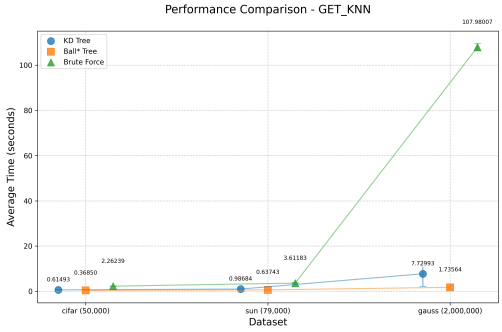
(c) Insert Operation Time



(d) Delete Operation Time

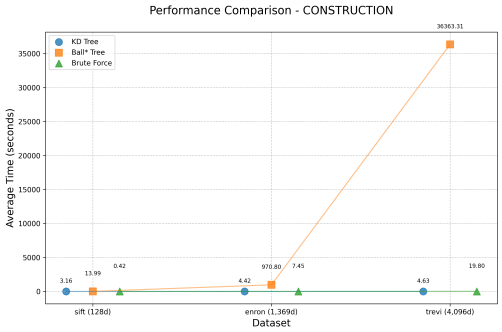


(e) Nearest Neighbor Query Time

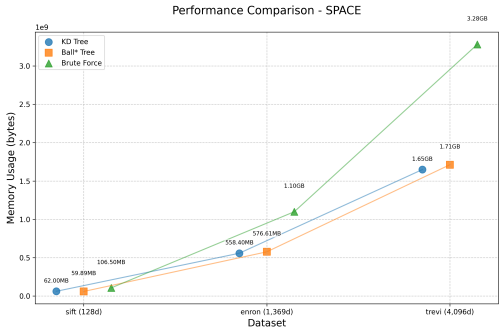


(f) k-NN Query Time

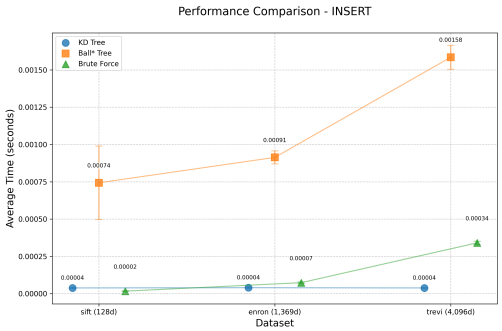
Figure 4: Performance comparison across different dataset sizes. Points corresponding to each data structure are horizontally spaced for clarity.



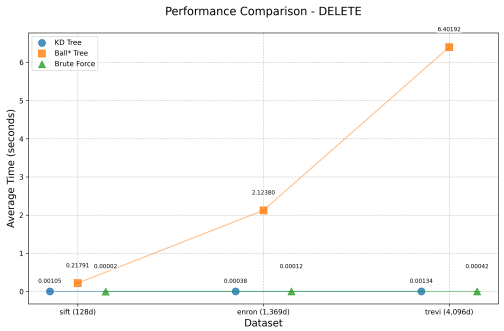
(a) Construction Time



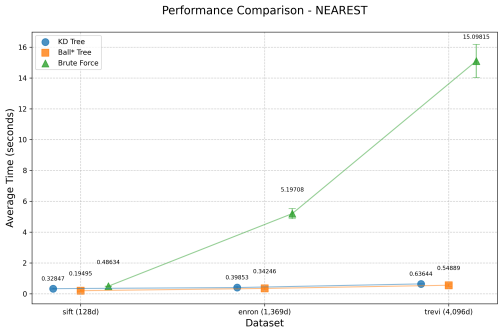
(b) Memory Usage



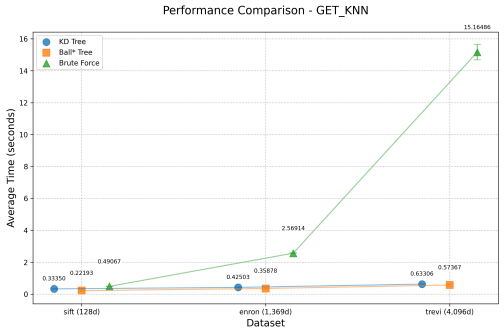
(c) Insert Operation Time



(d) Delete Operation Time



(e) Nearest Neighbor Query Time



(f) k-NN Query Time

Figure 5: Performance comparison across different dataset dimensionalities. Points corresponding to each data structure are horizontally spaced for clarity.