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. Hence, one application of this is to leverage ML models to learn representations of all products on e-commerce markets. When a user searches for a product, we can search for other related products on our platform according to their learned representation and recommend them to the user. A naive approach to this problem is to compute the distance 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. This has led to 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 further 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 most well-known geometric data structures and study their time and complexity by performing extensive experiments on large, high-dimensional datasets.

*Equal contribution

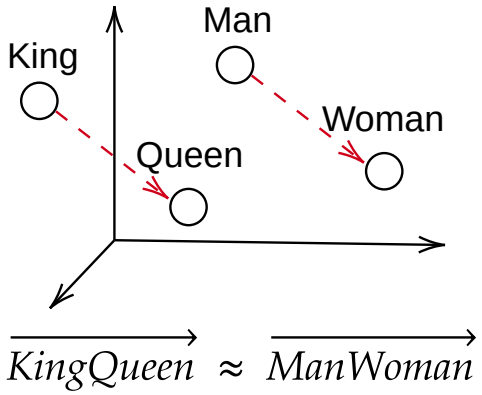

Figure 1: An example of representations of English words. Advancements in ML allow learning representations as high-dimension vectors (embeddings) that preserve semantic information for each word. In this example, the distance and direction of vector $\overrightarrow{KingQueen}$ is relatively the same 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 $x_i \in \mathbb{R}^M$ is the M-dimensional vector of point i. Let $n = |\mathcal{P}|$ be the number of points in $\mathcal{P}$. We consider the task of finding the Nearest Neighbour (NN) and the top K Nearest Neighbours (K-NN), where each task is defined as follows.

Definition 1 (Nearest Neighbour)**.** *Given a query point x_q, $q \in [1, n]$, the nearest neighbour 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 Neighbour)**.** *Given a query point x_q, $q \in [1, n]$, the K nearest neighbour 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 neighbour 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-base 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. However, 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 vector x_i of node i. This improvement made K-D tree work well on moderate dimensional data but still suffers from the curse of dimensionality when working with high dimensional data. K-D tree has shown to soon 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 KD-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" (Malkov & Yashunin, 2020). To overcome this, an approximate nearest neighbour 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 neighbour 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 neighbour techniques, however, highly depends on the quality of hashing functions. Another nearest neighbour 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 neighbour search methods are not guaranteed to find the correct solution. Hence, in this report, we focus on exact search methods, K-D tree and Ball* tree in particular, which is also well-aligned with the scope of our course.

3 METHODOLOGY

In this report, we focus on evaluating the performance of K-D tree (Section 3.1) and Ball* tree (Section 3.2) on five main operations: tree construction, insert, delete, get nearest neighbour 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 these selected geometric data structures.

3.1 K-D TREE

In general, K-D tree is a binary search tree where each node in the tree represents a K-dimensional point x_i from $\mathcal{P}$. In K-D tree, given a parent node, child nodes are split into left and right branches according to the value at an individual dimension defined by the discriminator. All nodes at the same 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 level is defined by the depth level modulus of the dimension of each point. The root node at a depth level of 0 partitions nodes into two sub-branches 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.

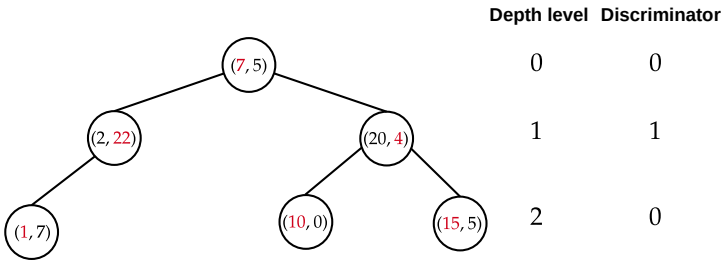

Figure 2: An example of K-D tree for 2 dimensional points (2-d tree)

Insert. Insert on K-D tree is similar to a binary tree with the time complexity of $O(logn)$. Algorithm 1 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 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 with from the subtree where it is found. Similar to insertion, deletion operation can be accomplished in $O(logn)$.

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 2. 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 $O(nlogn)$.

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 the K-D tree from the root node to a leaf node. However, to keep track of the top K nearest neighbour 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 procedure 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 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 neighbour operation by simply calling get K-NN with $K = 1$.

3.2 BALL* TREE

The 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 partitioning, Ball* tree partitions space with hyperspheres that can be oriented in any direction, allowing it to better adapt to the underlying data distribution. The tree forms 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. Figure 3 illustrates this nested ball hierarchy. This structure is particularly effective for high-dimensional data where axis-aligned partitioning becomes inefficient, as the ball-shaped regions can more compactly represent clusters of points regardless of their orientation in space.

Insert. The Ball* tree insertion algorithm has time complexity $O(\log n)$ in the average case. Algorithm 5 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

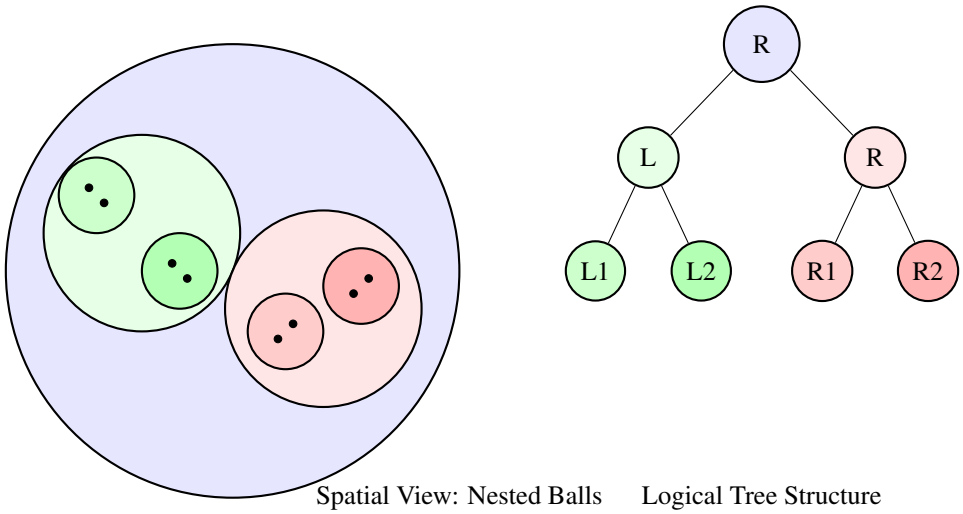

Figure 3: Ball* tree representation showing both spatial organization of points and the corresponding tree structure. The root ball contains all points and is recursively partitioned into smaller balls until leaf nodes are reached.

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.

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 6. Unlike K-D tree which splits along coordinate axes, Ball* tree uses Principal Component Analysis (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 (typically 10-100, 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 $O(n \log n)$.

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], Guass(Li et al., 2020), Enron and Trevi[4]. Further information about datasets used in this report is provided in Appendix A.

To evaluate time complexity, we tested each operation of K-D tree and Ball* tree using a consistent methodology. We selected 200 distinct query points x_i and performed each operation 3 times per point, 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 implemented K-D tree and Ball* tree in Python (version 3.11) and utilized Numpy (version 1.26.4) to optimize matrix operations. The computational environment was 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

5.1 SCALE NUMBER OF DATA POITNS

5.2 SCALE DIMENSION

6 DISCUSSION

ACKNOWLEDGMENTS

This research project uses the ICLR 2021 Conference submission template for formal formatting.

[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`

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.

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. A complete statistics about the datasets are described tin 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 Insert

Input: current node r,A point x, current depth level δ
if r is None **then**
    // Tree is empty
    $r \leftarrow$ initialize a tree node of new node x
    return
end if
$d \leftarrow \delta$ mod M // Compute discriminator
dx $\leftarrow x[d]$ - $r[d]$
if dx > 0 **then**
    if r.right is None **then**
        r.right $\leftarrow$ initialize a tree node of new node x
    else
        Insert(r.right,x, $\delta + 1$)
    end if
else
    if r.left is None **then**
        r.left $\leftarrow$ initialize a tree node of new node x
    else
        Insert(r.left,x, $\delta + 1$)
    end if
end if

Algorithm 2 Construct

Input: List of points $\mathcal{P}$, current depth level δ, dimension of each point M
if $\mathcal{P}$ is empty **then**
    return None
end if
$d \leftarrow \delta$ mod M // Compute discriminator
pivot $\leftarrow$ find median of $\mathcal{P}$ according to dimesion d
left $\leftarrow$ all points in $\mathcal{P}$ that are less than pivot at dimension d
right $\leftarrow$ all points in $\mathcal{P}$ that are greater than pivot at dimension d
node $\leftarrow$ initialize a tree node of pivot point
node.left $\leftarrow$ Construct(left,$\delta + 1$, M)
node.right $\leftarrow$ Construct(right,$\delta + 1$, M)
return node

Algorithm 3 Delete

Input: current node r,A point x, current depth level δ
$d \leftarrow \delta$ mod M // Compute discriminator
if $x = r$.coordinate **then**
    // Found the target node
    if r.right is None and r.left is None **then**
        return
    end if
    if r.right is not None **then**
        min $\leftarrow$ find the δ-minimum node in the right branch, r.right
        r.coordinate = min.coordiate // replace root with j-min from right subtree
        r.right = Delete(r.right,min.coordiate,$\delta + 1$)
    else
        max $\leftarrow$ find the δ-maximum node in the left branch, r.left
        r.coordinate = max.coordiate // replace root with δ-max from left subtree
        r.left = Delete(r.left,max.coordiate,$\delta + 1$)
    end if
else
    if $x[d] \geq r$.coordinate[d] **then**
        r.right = Delete(r.right,x,$\delta + 1$)
    else
        r.left = Delete(r.left,x,$\delta + 1$)
    end if
end if

Algorithm 4 Get K-NN

Input: Query point x, current node r, number of nodes return k, current priority queue (initialize with empty at the beginning) $prQueue$, current depth level δ , dimension of each point M, distance function f
if r is None **then**
    return None
end if
$d \leftarrow \delta$ mod M // Compute discriminator
distance = $f(x, r.\text{coordinate})$
if length($prQueue$) $< k$ **then**
    $prQueue$.enqueue((distance,r.coordinate)) // Insert new point
else if distance $< prQueue$.peek()[0] **then**
    $priQueue$.dequeue() // Remove the farthest point in the queue
    $prioQueue$.enqueue((distance,r.coordinate)) // Insert new point that is closer to x
end if
if $x < r$.coordinate[d] **then**
    next = r.left
    other = r.right
else
    next = r.right
    other = r.left
end if
    K-NN(x,next,k, $prQueue$, $\delta + 1$, M, f)
if $|x[d] - r.\text{coordinate}[d]| < prQueue$.peek()[0] **then**
    K-NN(x,other,k, $prQueue$, $\delta + 1$, M, f)
end if
return $prQueue$

B.2 BALL* TREE

Algorithm 5 Insert

procedure INSERT(node, point, depth)
  if node is None **then**
    return new BallNode(center=point, radius=0, points=[point])
  end if
  if node is Leaf **then** // Node has points array
    if $|$node.points$|$ $<$ leaf_size **then**
      node.points.append(point)
      node.center $\leftarrow$ mean(node.points) // Update center
      node.radius $\leftarrow$ $\max_{p \in \text{node.points}} \sqrt{d(\text{node.center}, p)}$ // Update radius
      return node
    else
      all_points $\leftarrow$ node.points $\cup$ {point}
      return ConstructTree(all_points, depth) // Rebuild subtree
    end if
  end if
  dist $\leftarrow$ $\sqrt{d(\text{node.center}, \text{point})}$ // Distance from center to new point
  if dist $>$ node.radius **then**
    node.radius $\leftarrow$ dist // Expand sphere if needed
  end if
  left_dist $\leftarrow$ $\sqrt{d(\text{node.left.center}, \text{point})}$ // Distance to left child
  right_dist $\leftarrow$ $\sqrt{d(\text{node.right.center}, \text{point})}$ // Distance to right child
  if left_dist $\leq$ right_dist **then**
    node.left $\leftarrow$ INSERT(node.left, point, depth+1)
  else
    node.right $\leftarrow$ INSERT(node.right, point, depth+1)
  end if
  return node
end procedure

Algorithm 6 Construct

```
procedure ConstructTree(points, depth)
    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
        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)
    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 procedure
```

Algorithm 7 Delete

```
procedure DELETE(point)
    node, deleted ← DELETEHELPER(root, point, 0)
    if not deleted then
        throw "Point not found in the Ball*-tree"
    end if
end procedure
procedure DELETEHELPER(node, point, depth)
    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
            end if
            node.center ← mean(node.points)
            node.radius ← max_{p∈node.points} sqrt(d(node.center, p))
            return node, True
        end if
        return node, False
    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
        else
            return None, True
        end if
    end if
    return node, False
end procedure
```

Algorithm 8 Get K-NN

```
procedure GETKNN(point, k)
    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 procedure
procedure GETKNNHELPER(point, node, k, pq, depth, tiebreaker)
    if node is None then
        return pq, tiebreaker
    end if
    center_dist ← √d(node.center, point)
    lower_bound ← max(0, center_dist − node.radius)
    if |pq| = k and lower_bound ≥ −pq[0][0] then
        return pq, tiebreaker // Prune this branch
    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))
                tiebreaker ← tiebreaker + 1
            else if dist < −pq[0][0] then
                pq.PopAndPush((−dist, tiebreaker, pt))
                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
    for all (lb, child) in children do
        pq, tiebreaker ← GETKNNHELPER(point, child, k, pq, depth+1, tiebreaker)
    end for
    return pq, tiebreaker
end procedure
```