

Benchmarking and Engineering Data Structures for Spherical Range Queries

Thomas Bläsius  

Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany

Jean-Pierre von der Heydt  

Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany

Tobias Kempf  

Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany

Dennis Kobert  

Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany

Nikolai Maas  

Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany

Abstract

Spherical range queries are a fundamental primitive for working with spatial data. Many spatial data structures have been developed to answer these queries, but choosing the optimal one for a specific application is a difficult task. This is because theoretical worst-case bounds are often overly pessimistic, and existing average-case analyses are rather restricted and hard to compare.

We address this problem with two main contributions. First, we present a comprehensive evaluation of state-of-the-art spatial indices across a diverse set of benchmarks. This includes a new benchmark based on graph embeddings alongside multiple real-world datasets from the literature. Our benchmark covers instances scaling up to 10 M points and ranging between 2 and 960 dimensions.

Second, we introduce the Sorted-Projection Radius KD-tree (SPRK-tree), a high-performance KD-tree variant. The SPRK-tree combines aggressive subtree pruning via radius reduction, sorted projection-based leaf nodes, and careful implementation optimizations. It consistently achieves the fastest query times in almost all benchmarks, and ranks second in the few remaining cases.

2012 ACM Subject Classification Theory of computation → Computational geometry; Theory of computation → Data structures design and analysis

Keywords and phrases Spherical Range Queries, Fixed-Radius Nearest Neighbor Search, Spatial Indexing, KD-tree, Benchmarking, Graph Embedding, SPRK-Tree

Digital Object Identifier 10.4230/LIPIcs.ESA.2026.22

Related Version *Full Version:* <https://arxiv.org/pdf/2607.07367>

Supplementary Material *Software (Source Code):* <https://github.com/wembed-pdf/sprk> [34]

archived at `swb:1:dir:293dee235b22ac27188619ae7c1b50797096641e`

Software (Source Code): <https://github.com/wembed-pdf/rembed> [33]

archived at `swb:1:dir:48b2e0e7921e86d574d2f7d019c422dddf8186dd`

Dataset: <https://doi.org/10.5281/zenodo.21243483>

Funding *Jean-Pierre von der Heydt:* This work was supported by funding from the pilot program Core-Informatics of the Helmholtz Association (HGF).

1 Introduction

Spherical range queries are a fundamental primitive for working with spatial data. Given a set of n points in d -dimensional Euclidean space, the objective is to retrieve all points that lie within the specified radius of a query point. In low-dimensional spaces, spatial indices



© Thomas Bläsius, Jean-Pierre von der Heydt, Tobias Kempf, Dennis Kobert, and Nikolai Maas; licensed under Creative Commons License CC-BY 4.0

34th Annual European Symposium on Algorithms (ESA 2026).

Editors: Philip Bille, Seth Pettie, and Sabine Storandt; Article No. 22; pp. 22:1–22:17

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

significantly accelerate these queries by pruning large portions of the search space. However, in higher dimensions a brute-force approach that checks the distance to every point is often the only viable option due to the curse of dimensionality [27].

One application of spatial indices is the computation of graph embeddings [7, 8, 31, 48], which play an important role in machine learning tasks such as community detection, link prediction, and node classification [18, 23, 36, 40, 47]. These methods map the vertices of a graph into a continuous space such that spatial distance reflects structural similarity [37]. This is often achieved by force-directed optimization, which applies attracting forces to adjacent vertices that are too distant from each other and repelling forces to non-adjacent vertices that are too close. The attracting forces can be computed in linear time $O(m)$ by iterating over all m edges. However, computing repelling forces requires identifying all non-adjacent vertices within a specific distance of each other. A naive brute-force implementation would check all $O(n^2)$ pairs of vertices, which is particularly prohibitive for large sparse graphs. Alternatively, the close vertex pairs can be identified by performing a spherical range query for each vertex, making the performance of the embedding framework heavily dependent on the efficiency of the underlying spatial index.

Beyond graph embedding, range queries also play a crucial role in numerous other domains such as particle-based simulations, database systems, and computer graphics [21, 26, 32]. For a comprehensive overview of applications, see the book by Samet [43]. To support these applications, numerous spatial indices have been developed; see Table 1 for a list. The theoretical worst-case bounds also shown in Table 1 give the impression that these spatial indices do not yield significant improvements over the brute-force approach. However, this is highly misleading and comes from the fact that the worst-case is often too pessimistic. In practice, the different spatial indices are often successful in pruning large chunks of the search space. In the following, we give a brief overview of the different spatial indices and broadly categorize them by how they organize and prune the search space.

The first category of spatial indices is based on partitioning the geometric space. Uniform grids partition space into fixed-size cells and insert the points into them. Queries are answered by a lookup of the corresponding cell and inspecting it and adjacent cells. These yield especially fast lookups in low dimensions when the points are evenly distributed. However, they lack adaptability and in datasets with heterogeneous density, dense cells often become a bottleneck. Orthtrees (also known as Quadtrees in 2D and Octrees in 3D) [42] address this by recursively dividing the space only where necessary. While more flexible than grids, they share the same limitation, where each split generates 2^d children, which is infeasible in higher dimensions.

Unlike space partitioning methods, data partitioning methods divide the point set itself. KD-trees [3, 19] split the point set along axis-aligned hyperplanes, to construct a balanced tree. To answer a query, the tree is traversed recursively, pruning subtrees that cannot contain query results. R-trees [24] group points into bounding rectangles, VP-trees [10, 49] partition by distance to a vantage point, and Balltrees [38] use bounding hyperspheres.

The SNN structure of Chen and Güttel [12] avoids hierarchical traversal entirely. It uses the singular value decomposition (SVD) to identify the direction of maximum variance in the dataset. Points are projected onto this axis and sorted. The query algorithm uses a binary search to identify a contiguous range of candidate points that is then scanned exhaustively.

This shows that there are plenty of spatial indices to choose from. As the theoretical worst-case bounds do not provide strong guidance, choosing the right data structure for the application at hand, e.g., graph embedding, can be a daunting task. While there are some average-case results that give guarantees in certain situations [4], they are rather

■ **Table 1** Theoretical complexity of spatial data structures. *Variables:* n = number of points; d = number of dimensions; k = output size; g = grid resolution (number of cells per dimension).

Structure	Construction	Space	Query (Worst-case)
Brute-Force	$O(1)$	$O(nd)$	$O(nd)$
Uniform Grid ¹	$O(nd + g^d)$	$O(nd + g^d)$	$O(nd + 2^d)$
Orthtree [42]	$O(n \cdot 2^d)$	$O(n \cdot 2^d)$	$O(n \cdot 2^d)$
KD-tree [3, 19]	$O(dn \log n)$	$O(nd)$	$O(d \cdot n^{1-1/d} + dk)$
R-tree [24]	$O(dn \log n)$	$O(nd)$	$O(nd)$
VP-tree [49, 10]	$O(dn \log n)$	$O(nd)$	$O(nd)$
Balltree [38]	$O(dn \log n)$	$O(nd)$	$O(nd)$
SNN	$O(dn \log n + nd^2)$	$O(nd)$	$O(nd)$

restricted and sometimes difficult to compare. Moreover, the performance depends on the data distribution of the specific application as well as implementation details. Consequently, an empirical evaluation is essential to choose the best data structure. The only empirical study in this direction we are aware of is by Lawson, Gropp, and Lofstead [35], whose comparison is tailored for the HPC-context, focusing on orthogonal range queries in 3-dimensional space in a distributed setting. Additionally, some libraries, e.g., the KD-tree library Kiddo [13], include benchmarks. However, to the best of our knowledge, there has been no extensive scientific comparison of spatial indices on a diverse set of inputs with varying dimensions yet.

1.1 Our Contribution

We have two main contributions. First, we address the need for an empirical comparison by providing a comprehensive evaluation of different state-of-the-art spatial indices. Secondly, we provide a new KD-tree variant that we call *Sorted-Projection Radius KD-tree* (SPRK-tree; pronounced “spark”), which outperforms the competition on almost all benchmarks. In the following, we discuss these two contributions in more detail; starting with the SPRK-tree.

SPRK-Tree. At its core, the SPRK-tree is a KD-tree variant that recursively partitions the data points until the number of points falls below a certain predefined bucket size for the leaves; see Section 2.1. In the leaves, we use a variant of SNN [12], i.e., we sort the remaining points with respect to a projection line; see Section 2.3. For the query, we utilize pruning techniques, which reduce the query radius incrementally while traversing the hierarchy [1, 19, 44]. This allows more aggressive pruning of subtrees, while still ensuring that no points in the original query ball are omitted; see Section 2.2. For high-dimensional data sets, we initially rotate the data based on a singular value decomposition of sampled points, such that the first splits done by the KD-tree separate along directions with high spread; see Section 2.4. Our Rust implementation features SIMD vectorization and cache-aware memory layout (see Section 2.5), and provides bindings for C++ and Python.

Experimental Evaluation. Our evaluation uses a large and diverse set of benchmarks, including a newly created benchmark set. Our new set comes from the graph embedding application discussed above. It features over 225 embedding instances with up to 1 M points

¹ Grid: for the stated query time, the query radius must be known at construction time.

in dimensions $\{2, 3, \dots, 16\}$. Beyond that, we also consider points distributed uniformly at random and benchmarks from the literature, including data sets from clustering, point of interest queries, and a high-dimensional real-world data set with point sets of up to 960 dimensions. Besides our SPRK-tree, we evaluate 10 other implementations as well as a brute-force approach. The evaluated data structures are diverse (Quadtree, uniform grid, Balltree, VP-tree, R-tree, KD-tree, SNN) and include state-of-the-art implementations of KD-trees in multiple programming languages. Our findings can be summarized as follows.

- For the embedding benchmark, KD-trees are the method of choice. While the KD-tree libraries Kiddo [13] and Neighbourhood [25] already perform well, our SPRK-tree consistently outperforms them. For sufficiently high dimensions, brute-force eventually becomes the best option, yet our SPRK-tree still outperforms brute-force by an order of magnitude for 16 dimensions.
- For randomly distributed points, tree-based methods perform well for low dimensions but degrade for higher dimensions. This is contrasted by SNN and brute-force, which perform much worse on lower dimensions but outperform the tree-based methods for dimension 32. The exception to this is our SPRK-tree. While being the best tree-based method for low dimensions, it is also (slightly) faster than SNN and brute-force for dimension 32, outperforming the other tree-based methods by an order of magnitude.
- While we developed the SPRK-tree with the embedding application in mind, it still outperforms the competitors on most benchmarks from other applications. The only exception is the benchmark of fixed-radius search in (very) high dimensions, where the SPRK-tree is usually second after SNN or brute-force, but still stays competitive.

Beyond the extensive comparison with other data structures, we provide an empirical analysis of how our design and optimization choices contribute to the performance of the SPRK-tree.

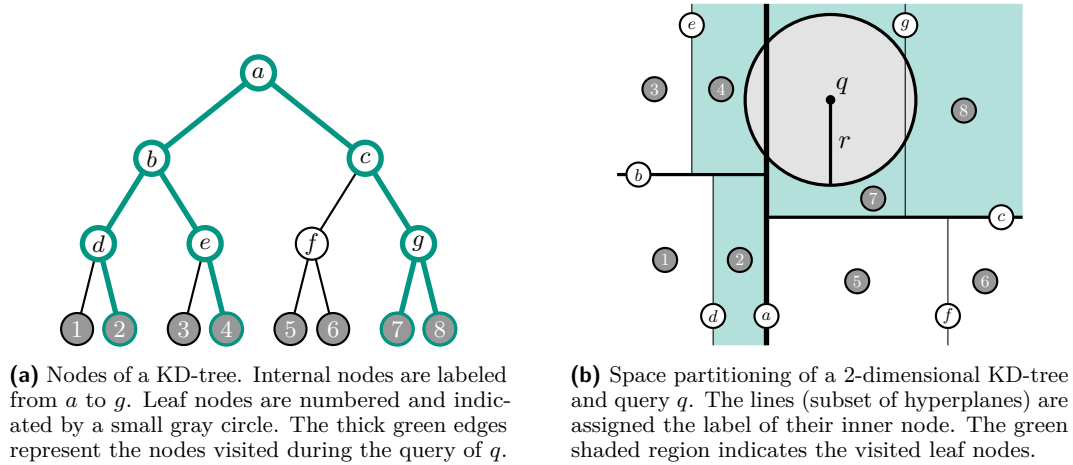
2 The SPRK-tree

In this section we introduce the SPRK-tree, which improves upon the standard KD-tree through three primary mechanisms: aggressive pruning of subtrees using radius reduction, the use of SNN-inspired sorted leaf buckets, and an SVD-based rotation to improve performance on high-dimensional data. We begin with a brief description of the standard KD-tree, followed by a detailed explanation of our improvements. We conclude by describing implementation optimizations that further boost the performance of the SPRK-tree.

2.1 KD-tree

The *KD-tree* [3, 19] is a spatial index that organizes a point set $P \subseteq \mathbb{R}^d$ for range queries. It chooses an axis-aligned hyperplane to split the whole space into two half-spaces such that both half-spaces contain the same number of points. This procedure is applied recursively, yielding a binary tree, where each *internal node* corresponds to a hyperplane and its two children correspond to the half-spaces. When the number of points falls below a predefined bucket size, the recursion stops and creates a *leaf node*. Each node is associated with a *region*, namely the intersection of all half-spaces on the path from the *root* to the node. With this, the leaves form a partition of the whole space $\mathbb{R}^d$; for an example see Figure 1.

A degree of freedom to fill in is the choice of the axis-aligned hyperplane in each step. We achieve this by cycling through the dimensions in round-robin order and placing the hyperplane at the median coordinate of the current points along that dimension. This halves the point set at every split and produces a tree of logarithmic depth.



■ **Figure 1** Illustration of a 2-dimensional KD-tree and the nodes visited by a query q .

A *spherical range query* with query point $q \in \mathbb{R}^d$ and radius r asks for all points in P within distance r of q . The KD-tree resolves this by recursively traversing the tree and pruning subtrees that cannot contain query results. Specifically, at an internal node, a child subtree is pruned if the query sphere does not intersect the child's half-space. Upon reaching a leaf, every point in the leaf is checked against the query radius by an exhaustive search.

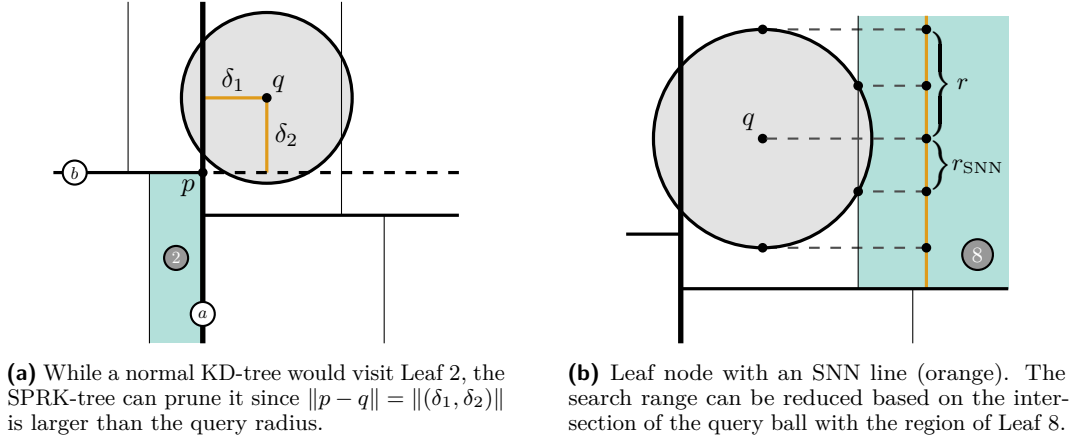
2.2 Radius Reduction

Note that the default pruning of a KD-tree does visit unnecessary regions, e.g., Region 2 in Figure 1 despite the fact that Region 2 does not intersect the query ball. This can be prevented by a more informed pruning procedure [1, 19, 44]. For this, consider the example in Figure 2a (which is the same as in Figure 1), and let p be the point of Region 2 closest to the query point q . Let δ_1 and δ_2 be the distance between p and q in dimensions 1 and 2, respectively. If we had computed δ_1 and δ_2 , we could have pruned early by observing that $\|(\delta_1, \delta_2)\| > r$. To compute δ_1 and δ_2 , note that they are also the distances of q to the hyperplanes a and b , respectively, which both separate q from Region 2. This is not a coincidence: for any given region, we can compute the corresponding δ_1 and δ_2 by going up the tree; for each dimension taking the maximum distance between q and a hyperplane (of that dimension) separating q from the region.

Similar to [1, 44], we do not recompute all δ values for each node of the tree individually but keep track of a vector $\Delta = (\delta_1, \dots, \delta_d)$ while traversing the tree. This allows us to update $\|\Delta\|$ using a constant number of arithmetic operations for the distance check. We start with $\delta_i = 0$ for all $i \in [d]$. Now let u be a node in the tree with corresponding hyperplane H_u . Consider traversing from u to a child v such that H_u separates the query point q from the region corresponding to v . Then we update δ_i to the distance from the query point q to the hyperplane H_u , where i is the dimension of H_u .

2.3 SNN Buckets

In a regular KD-tree, the points in a leaf node are filtered by a linear scan. The SPRK-tree replaces this with an SNN data structure [12] for each leaf. The SNN structure is constructed by projecting all the points of the leaf onto an axis-aligned line and sorting along its dimension; see Region 8 in Figure 2b. Instead of now scanning all points in the leaf, we can prune some



■ **Figure 2** Illustration of the search space pruning in the SPRK-tree.

points based on the projection. We can do this by projecting the query point q to the SNN line and then finding all points within distance r in both directions from q on the line. This can be done with a binary search followed by a linear scan along the order. Note that the resulting set of points has to be filtered for points actually in the query ball.

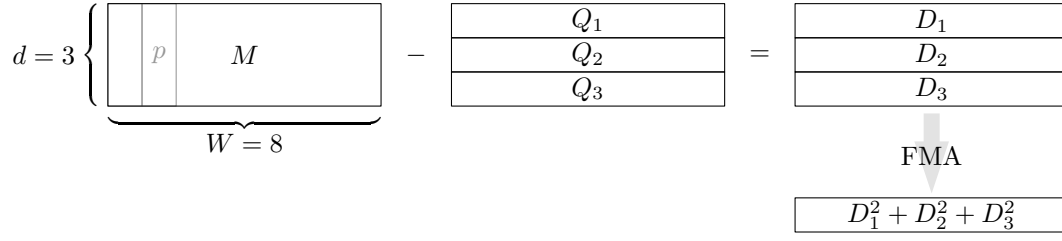
Observe in Figure 2b that the intersection of the query ball with Region 8 gives rise to an improvement over this. In the direction of the SNN line, this intersection spans only a smaller range of size $2 \cdot r_{\text{SNN}}$ than the range of $2 \cdot r$ queried above. To compute this smaller radius r_{SNN} , we can utilize the distance vector Δ from the previous section. To make this precise, let the dimension of the SNN line be, without loss of generality, the first dimension. Then r_{SNN} is the maximum value such that $(r_{\text{SNN}}, \delta_2, \dots, \delta_d)$ has length at most r .

Furthermore, to avoid the binary search, the SPRK-tree computes a lookup table for the sorted coordinates of the leaf nodes by discretizing the coordinate range into fixed-width bins.

2.4 Singular Value Decomposition

In many scenarios, high-dimensional data often resides within a lower-dimensional subspace. The SPRK-tree exploits this property by rotating the data so that the d principal components with the largest variance align in descending order with the coordinate axes. Because the tree splits dimensions in a round-robin sequence, this rotation ensures that the earliest partitions occur along the directions of maximum spread, enabling more efficient pruning near the root. The rotation is computed via the SVD of a sampled subset of up to 10k points of P , which ensures that the SVD computation does not dominate the overall construction cost.

However, this transformation introduces a slight overhead during the query phase, as every subsequent query requires an initial rotation into the new coordinate space. Consequently, the SPRK-tree only applies this rotation for datasets with more than 16 dimensions. In these high-dimensional cases, the tree depth is often smaller than d , meaning a standard traversal would never cycle through all dimensions. The SVD rotation guarantees that the limited number of splits are used to separate the data along the most informative dimensions.



■ **Figure 3** Illustration of the squared distance computation for $W = 8$ points in $d = 3$ dimensions, requiring $2d$ SIMD instructions.

2.5 Performance Engineering

Beyond algorithmic improvements, we engineered our implementation to utilize the full capabilities of modern hardware. In the following, we describe the most important optimizations.

Distance Computations. During a query, the spatial index spends a large part of the time computing distances between the query point and candidate points in the leaf nodes. A naive SIMD (Single Instruction, Multiple Data [17]) implementation might attempt to accelerate this by computing the differences across the d dimensions of a single point simultaneously. However, this can be wasteful since the dimensionality d varies per dataset and does not necessarily align with the fixed width of hardware SIMD registers.

To resolve this, we vectorize across points rather than dimensions. The leaf buckets already store points in a contiguous array, and we group them into blocks of fixed width W . We represent each block as a matrix $M \in \mathbb{R}^{d \times W}$ in row-major order. By choosing $W = 8$, the matrix rows fit exactly into one 256-bit SIMD register using 32-bit floating-point arithmetic.

Let q be the query point. To compute the squared distance to all W points simultaneously, we broadcast its i -th coordinate q_i into a SIMD register Q_i , creating W copies of that value. Second, we calculate the coordinate-wise differences $D_i = Q_i - M_i$ using one SIMD subtraction per dimension, where M_i is the i -th row of the matrix. We then square and sum these differences element-wise into an accumulator vector acc using one fused multiply-add (FMA) instruction per dimension, effectively computing $\text{acc} = \sum_{i=1}^d D_i^2$ element-wise. This requires $2d$ SIMD instructions (d subtractions and d FMAs) to evaluate the distance to W vectors; see Figure 3.

For higher dimensions, we reduce the instruction count further by rewriting the squared distance $\|p - q\|^2$ between a candidate point p and the query point q as $(p^\top p + q^\top q) - 2p^\top q$ and precomputing the squared norms $p^\top p$, as suggested by Chen and Güttel [12]. By storing the halved squared norms, we also avoid a multiplication by 2 during the accumulation phase:

$$\|p - q\|^2 = (p^\top p + q^\top q) - 2p^\top q \leq r^2 \iff \left(\frac{p^\top p}{2} + \frac{q^\top q}{2} \right) - p^\top q \leq \frac{r^2}{2}.$$

Computing the left side of the inequality for all W points simultaneously requires only d FMA instructions. However, it requires loading the precomputed values $\frac{p^\top p}{2}$ from memory. Due to this overhead, it only becomes faster in practice for $d \geq 6$.

To break data dependencies and maximize instruction throughput, we split the computation across 2 independent accumulators. We initialize them to the SIMD vectors containing $\frac{p^\top p}{2}$ and $\frac{q^\top q}{2}$ respectively, and alternate the subtraction of the coordinate-wise products of $p^\top q$ using the previously allocated SIMD registers M_i and Q_i . For $d > 30$, utilizing 4 accumulation registers provides additional performance gains by further hiding instruction

latency; see the detailed analysis in Section 3.3. Note that the dot-product reformulation potentially introduces numerical instabilities. We therefore add a small fixed $\varepsilon = 10^{-4}$ to the query radius in our implementation. Across all our test cases, this returned every point found by a naive reference implementation.

After computing all candidate distances, the indices corresponding to points fulfilling $\|p - q\| \leq r$ must be compacted into the result buffer. On CPUs with AVX-512F [28, 29] support, we use the `vcompressps` [30] instruction to gather the indices in a single step.

Memory Layout. In addition to vectorized arithmetic, we optimize for cache-efficiency. Inner nodes are extremely lightweight, storing only the 32-bit floating-point split value of the dividing hyperplane. In contrast, leaf nodes are heavy, containing the SNN bucket with metadata and point range indices. Therefore, we keep leaves and inner nodes in separate arrays, and use for the inner nodes an implicit tree layout analogous to binary heaps.

Beyond the memory layout, we also optimize memory access patterns. During a query, a traditional KD-tree implementation frequently switches between traversing the tree and processing distance computations in the leaf nodes. This causes cache misses, as the CPU evicts the tree structure from the cache to load point coordinates. To resolve this, we decouple the query into a two-phase algorithm. The first phase traverses the tree in depth-first search order to compute the candidate ranges for the leaves. It identifies all ranges that intersect the query ball and collects them into an intermediate buffer. The second phase then computes the actual distances. Because the coordinate data is stored contiguously, iterating over the collected ranges results in a mostly linear access pattern (with skips between ranges), thereby improving the memory prefetcher performance.²

3 Experiments

In this section, we present our experiments comparing the SPRK-tree against state-of-the-art spatial data structures. Section 3.1 outlines the experimental setup, introduces the baseline data structures, and details our new graph embedding benchmark. Following this, we compare query performance across a diverse set of benchmarks in Section 3.2, showing that the SPRK-tree consistently achieves the fastest query times across most evaluated datasets. Finally, we identify which optimizations contribute most to its efficiency in Section 3.3.

3.1 Experiment Setup

All benchmarks are performed single-threaded on a dual-socket Intel Xeon Gold 6144 CPU @ 3.50 GHz with 8 cores per socket, 32 KiB L1 and 1 MiB L2 cache per core, and 24.75 MiB L3 cache per socket, supporting AVX-512. The system has 192 GB DDR4 RAM and runs Ubuntu 24.04 LTS. Rust code was compiled with `rustc` version 1.94.0 in release mode with `RUSTFLAGS = -Ctarget-cpu=native`. C++ code was compiled with `gcc` version 14.2 using `-O3` and `-march=native`. For the Python libraries, CPython v3.12.3 was used.

After a 3-second warm-up, each benchmark runs for at least 20 seconds and at least 10 repetitions. We additionally verified that the measured times are consistent with hardware performance counters (`cpu_cycles`, `cpu_ref_cycles`, and `instructions`).

² As our access pattern is already very efficient, more complex inner node layouts yielded no further benefits in our experiments. For example, we observed no improvement with a van Emde Boas layout [20, 41].

Evaluated Data Structures. We compare the SPRK-tree against a broad selection of state-of-the-art spatial index implementations. The evaluated implementations span the scikit-learn Balltree [39], a VP-tree implemented in Rust [45], the Boost R-tree [9], and SNN [12]. Since KD-trees proved to be the strongest category overall, we include KD-tree implementations across multiple languages: the Rust crates Kiddo [13], nabo [15] and Neighbourhood [25], the C++ libraries nanoflann [5] and CGAL [44], and the scikit-learn KD-tree [39].

In addition to our SPRK-tree, we provide a brute-force baseline and our own Orthtree and uniform grid implementations. Notably, our grid implementation requires the query radius to be specified at construction time, which may be prohibitive for some applications. For datasets with varying query radius, we build the grid using the average query radius.

Graph Embedding Benchmark. As introduced in Section 1, we provide a new benchmark set based on graph embeddings. We use geometric inhomogeneous random graphs (GIRGs) [11] generated using the method in [6]. GIRGs are a widely used model for real-world networks with heterogeneous degree distributions and community structure. For generation, we set the average degree to 15, the power-law exponent to 2.5, the dimension to 4, and the temperature to 0 (corresponding to $\alpha = \infty$). We vary the number of vertices n such that $n \in \{10^4, 10^{4.5}, 10^5, 10^{5.5}, 10^6\}$ and use 3 different seeds for each parameter configuration.

To obtain a data set from these graphs, we compute embeddings using the WEmbed algorithm [8]. The algorithm iteratively adjusts vertex positions starting from a random initial state. In each iteration, the algorithm performs n spherical range queries, finding the set of vertices that are within a certain distance for each vertex. The query radius depends on the vertex degree, with high-degree vertices having a larger query radius. To obtain data sets with varying dimension, we set the embedding dimension d from 2 to 16. This results in $5 \cdot 3 \cdot 15 = 225$ different embeddings across all graph sizes.

The embedding algorithm updates the vertex positions for t iterations until convergence (averaging $t \approx 530$), yielding t different position and query sets per embedding. For each run, we choose the positions from the latest iteration divisible by 100 to include in our benchmark, as later iterations feature less random and more interesting vertex positions and are usually more expensive. To normalize the number of queries, we choose at most 10k queries per embedding, sorting the vertices by degree and then choosing every k -th vertex for $k = n/10^4$.

3.2 Performance Comparison

In this section, we compare the query performance of the evaluated data structures. We begin with the graph embedding benchmark, which motivated the design of the SPRK-tree. To evaluate the SPRK-tree’s applicability beyond graph embedding and obtain a more robust comparison, we consider four additional categories: uniform point distributions, high-dimensional real-world data, clustering workloads, and Point of Interest (POI) queries.

Graph Embedding. Figure 4 illustrates query times across embedding dimensions and vertex counts for a subset of data structures, including the best performing implementation for each algorithm. The SPRK-tree achieves the best query time in every tested configuration, across all vertex counts and dimensions. For $n = 10\text{ k}$ and $d \geq 12$, all structures except the SPRK-tree are outperformed by brute-force, an expected consequence of the curse of dimensionality. Here, the SPRK-tree likely prevails due to better-optimized constant factors. SNN generally exhibits similar performance to brute-force for high dimensions. We observe that SNN performs worse in dimensions not divisible by 4, which is likely due to the BLAS [46] implementation using a 4×4 kernel for the `sgemv` subroutine. Furthermore, at 1 M vertices

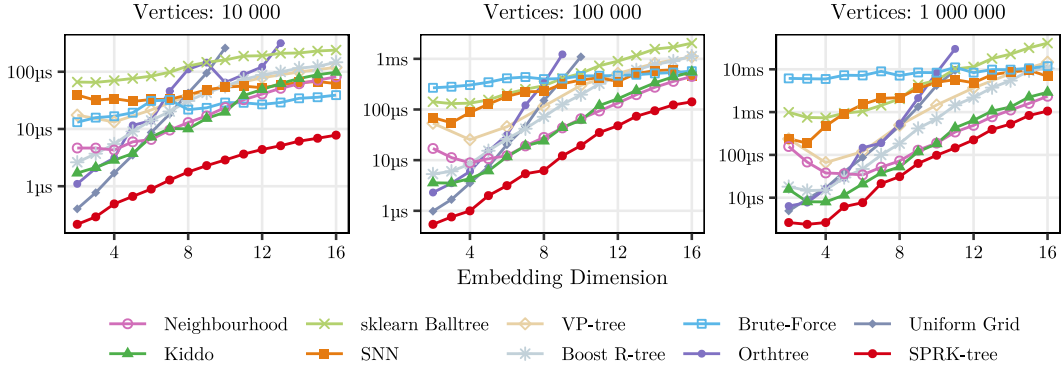


Figure 4 Average time per query for our graph embedding benchmark, with varying number of vertices and embedding dimensions. The plots show a subset of structures for legibility.

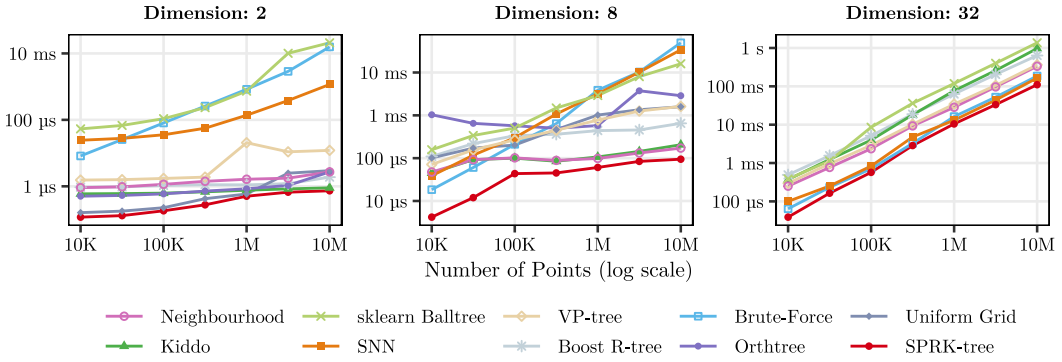


Figure 5 Query time for uniformly distributed points in a unit hypercube. The radius is chosen such that the expected number of neighbors is 15.

several structures (Kiddo, Boost R-tree, Neighbourhood, and scikit-learn Balltree) are slower in 2 dimensions than in 4. This might be an artifact of the embedding benchmark, where lower dimensions often have a high density of points just outside the query radius, thereby slowing structures that are unable to efficiently prune these points.

Table 2 lists the comprehensive results for all evaluated structures. Most competitors not shown in Figure 4 are clearly outperformed by the remaining structures. In higher dimensions the grid and Orthtree ran out of memory (oom).

Uniform Distributions. Next, we evaluate a benchmark of uniformly distributed points within a unit hypercube. We vary the number of points from 10 k to 10 M and the dimension from 2 to 32. For each point set, we perform 10 k queries with the query radius set such that the expected number of returned points is 15. Figure 5 shows the results, restricted to the best-performing implementations for readability. The SPRK-tree is the fastest structure in every tested configuration. Its advantage is most pronounced at low point counts and low dimensions, where it outperforms the next-best competitor by up to an order of magnitude. This gap narrows at high dimensions with many points, maintaining a factor of roughly $1.6\times$ over brute-force at dimension 32 with 10 M points. Most tree-based methods scale similarly to the SPRK-tree, but for low point counts the SPRK-tree is significantly faster, likely due to better cache performance.

■ **Table 2** Query time in μs for a single spherical range query in an embedded graph for varying number of vertices and embedding dimensions. The **fastest** and **second-fastest** performing methods for each combination are highlighted. The first section contains KD-tree implementations, the second section contains other methods and the third section contains our own implementations.

num vertices	10 000			100 000			1 000 000		
dimensions	2	8	16	2	8	16	2	8	16
sklearn KD-tree	67.7	110.4	316.8	144.9	265.9	2038.0	1009.5	1171.7	14 708.7
CGAL KD-tree	4.3	50.5	253.5	8.6	93.8	1248.1	29.1	153.7	7734.9
nanoflann	8.6	15.7	121.2	27.5	41.6	869.4	183.6	189.0	8439.3
nabo	5.6	18.9	135.3	15.0	34.9	668.7	50.4	65.6	2866.3
Neighbourhood	4.6	13.0	81.3	17.0	28.1	<u>448.7</u>	156.9	72.3	<u>2339.3</u>
Kiddo	1.7	<u>10.1</u>	98.0	3.6	<u>23.8</u>	549.6	15.7	<u>52.5</u>	2900.8
sklearn Balltree	66.1	126.0	239.0	142.1	294.9	2041.0	990.5	2036.5	40 821.5
SNN	39.2	39.6	62.2	69.2	227.3	451.5	238.8	2174.0	6875.8
VP-tree	17.5	38.0	119.3	53.5	116.0	1096.5	236.0	493.9	14 639.7
Boost R-tree	2.6	30.9	146.6	5.3	71.7	1129.7	18.1	183.8	10 543.4
Brute-Force	13.2	21.8	<u>39.0</u>	268.0	398.8	570.8	6133.0	7111.6	11 808.4
Orthtree	0.6	55.1	1175.6	1.2	165.7	2162.7	3.2	269.9	oom
Uniform Grid	0.2	21.3	oom	0.5	76.0	oom	2.5	239.1	oom
SPRK-tree	0.2	1.8	7.8	0.5	6.2	141.7	<u>2.6</u>	31.1	1056.5

The relative ranking of the competitors shifts with dimensionality. In low dimensions (2 and 8), Kiddo and the Boost R-Tree are close to the SPRK-tree, while SNN is significantly slower. For larger dimensions, the more sophisticated tree-based structures tend to be increasingly outperformed by brute-force, and SNN converges toward brute-force performance.

High-Dimensional Real-World Datasets. We further evaluate data structures on high-dimensional real-world datasets from the benchmark collection of Aumüller, Bernhardsson and Faithfull [2]. This collection features datasets with up to 10 M points and up to 960 dimensions; see Table 3 (first section). Some datasets exhibit very high dimension-to-point ratios; the small SIFT dataset, for instance, has 128 dimensions but only 25 k points.

Based on our previous benchmark results, one would expect tree-based methods to struggle heavily in this regime. While other tree-based implementations indeed fall behind by an order of magnitude or more, the SPRK-tree remains competitive, converging toward brute-force and SNN performance levels. SNN is the fastest method on four of the six nearest neighbor datasets, while the SPRK-tree consistently places second. We attribute the SPRK-tree’s strong performance mostly to the initial SVD rotation and optimized distance computation. The SVD is also applied by SNN, explaining the similar performance.

Clustering. We also evaluate the clustering benchmarks introduced in the SNN paper [12]. These benchmarks use DBSCAN [16], a density-based clustering algorithm whose core operation is a fixed-radius range query for every point in the dataset. We isolate the all-to-all query component to measure spatial index performance independently of the clustering logic (note that [12] reports the end-to-end runtime). The datasets are small, containing 200 to 4500 points, but span dimensions from 4 to 256.

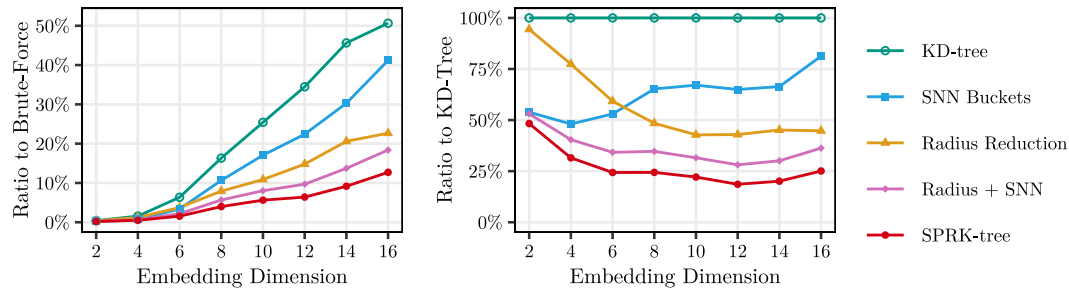
■ **Table 3** Real-world query benchmarks. Running time is in ms (nearest neighbor) or ns (clustering, POI) averaged over all queries and radii. The **fastest** and second-fastest methods are highlighted. In some cases, the Orthtree runs out of memory (oom).

Data Set	n	d	Brute-Force	Orthtree	Kiddo	Balltree	SNN	SPRK
Real-World high-dimensional datasets (in ms) [2]								
SIFT1M	100 k	128	5.8	oom	15.3	19.9	2.1	<u>2.3</u>
SIFT10K	25 k	128	0.9	oom	2.2	4.1	0.3	<u>0.4</u>
GIST	1 M	960	433.8	oom	2591.8	1415.4	183.2	<u>255.4</u>
GloVe100	1 M	100	55.4	oom	49.5	295.7	<u>29.7</u>	4.4
Deep1B	10 M	96	308.0	oom	2831.2	2700.9	453.7	<u>312.2</u>
F-MNIST	60 k	784	21.2	oom	41.0	60.8	5.9	<u>6.0</u>
Clustering (in ns) [14]								
Banknote	1.5 k	4	1750.6	<u>294.8</u>	672.4	57 833.6	21 593.0	99.8
Wine	0.2 k	13	<u>1131.4</u>	204 595.4	1922.6	54 822.0	18 907.0	143.0
Dermatology	0.4 k	34	<u>3023.2</u>	oom	10 815.4	65 718.8	23 220.4	1136.4
Ecoli	0.3 k	7	884.4	1018.4	<u>816.6</u>	54 875.8	18 502.8	78.6
Point of Interest Search (in ns) [22]								
ATM	12 k	2	10 189.0	<u>242.0</u>	269.0	53 160.2	20 069.5	86.0
Bakery	33 k	2	27 424.0	<u>454.2</u>	648.8	69 502.0	23 148.8	112.0
Parking	769 k	2	637 569.5	<u>1990.5</u>	3199.8	605 426.2	66 871.8	619.5
Bus Stop	761 k	2	628 722.5	<u>1481.2</u>	2094.8	512 312.8	51 150.8	444.8
Restaurant	103 k	2	84 881.0	<u>598.0</u>	745.0	109 636.5	27 474.8	167.5

As shown in the second section of Table 3, the SPRK-tree achieves the best query time on all clustering datasets, often by a wide margin. Brute-force and Kiddo alternate as the second-best method depending on the dimensionality: Kiddo places second on the lower-dimensional datasets (Banknote, Ecoli), while brute-force takes second on the higher-dimensional ones (Dermatology, Wine). SNN performs poorly across all clustering datasets, likely due to constant overheads that dominate at these small point counts.

POI Queries. To evaluate performance on geographic data, we benchmark fixed-radius POI queries, e.g., finding all bus stops within 500 m of a given location. These datasets are two-dimensional but exhibit highly heterogeneous density, with dense clusters in cities and sparse coverage in rural areas. We use data from the OpenStreetMap project for Germany [22], with five categories: ATMs (≈ 12 k), bakeries (≈ 33 k), parking facilities (≈ 769 k), bus stops (≈ 761 k), and restaurants (≈ 103 k). Each query set is constructed by using POIs of one category as dataset and POIs of a different category as query points. Queries are run at radii of 500 m, 1000 m, 2000 m, and 5000 m, and times are averaged across all radii and queries.

As shown in the POI section of Table 3, the SPRK-tree achieves the best query time across all five datasets. SNN is 2 to 3 orders of magnitude slower in this setting. The reason is likely that in 2 dimensions, the pruning of tree-based methods is more effective than projecting onto a single axis. Among tree-based methods, the Orthtree is the strongest competitor, yet the SPRK-tree outperforms it by a factor of $2.8\times$ on the smallest dataset (ATM, 12 k points) and up to $3.3\times$ on larger datasets (Bus Stop, 761 k points). We attribute this to spherical pruning, which reduces the distance checks by roughly 50 % in 2 dimensions.



■ **Figure 6** Effect of radius reduction and SNN buckets for a 100 k point dataset. The y-axis shows the ratio of distance checks performed compared to brute-force (left) and a KD-tree (right).

3.3 Breakdown of Performance Gains

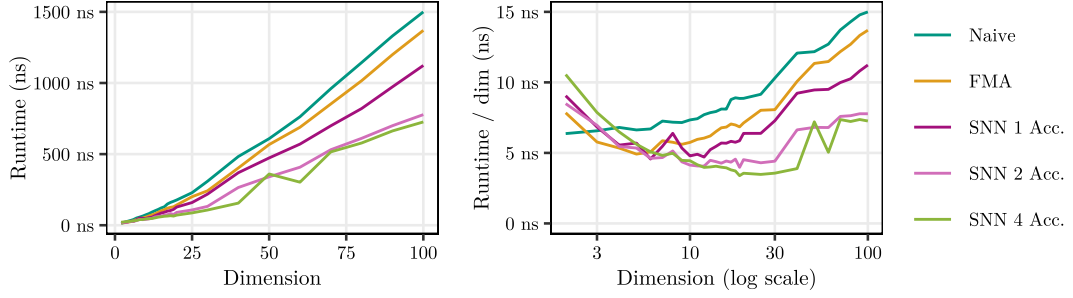
In this section, we analyze the individual contributions of the different optimizations (both algorithmic and low-level) to the overall performance of the SPRK-tree. We first measure the reduction in distance checks achieved by radius reduction and SNN buckets, followed by evaluating the SIMD and memory layout optimizations.

Algorithmic Optimizations. Recall that the SPRK-tree is a KD-tree with radius reduction and SNN in the leaves (Section 2). In Figure 6, we present an ablation study for these optimizations on the 100 k embedding dataset. We measure the number of distance computations performed by the data structure, which is independent of low-level optimizations.

The baseline is our KD-tree implementation without radius reduction or SNN (**KD-tree**) as described in Section 2.1. Using only radius reduction (**Radius Reduction**) cuts the number of distance checks significantly, achieving up to 50 % fewer checks in higher dimensions. On the other hand, SNN buckets in isolation (**SNN Buckets**) are most effective in lower dimensions, where they achieve a similar reduction of 50 %. The full benefit is realized by combining both techniques (**Radius + SNN**) and also using the reduced radius to further narrow the SNN candidate range (representing the full **SPRK-Tree**). This brings the total number of distance checks down to about 25 % of a regular KD-tree for $d \geq 6$.

Distance Computation. To evaluate the low-level optimizations of the distance computation introduced in Section 2.5, we conduct a microbenchmark. We measure the computation of 128 distances to a single query point across varying dimensions. We compare the direct distance computation without optimizations (**naive**), the approach using fused multiply-add instructions (**FMA**), the inner product reformulation (**SNN Acc**), and the variants using 2 and 4 accumulation registers (**SNN 2 Acc** and **SNN 4 Acc**). The results are shown in Figure 7.

The fused multiply-add approach provides moderate improvement over the naive baseline, reducing runtime by roughly 10 % on average. The inner product reformulation initially performs worse, but becomes faster than **FMA** at 6 dimensions. In low dimensions, loading the precomputed norms from memory likely outweighs the benefits of reduced arithmetic operations. The variants with multiple accumulators provide additional improvements in higher dimensions. For the dimensions targeted by our main benchmarks ($d \leq 16$), 2 accumulators offer a good trade-off, with a speedup of up to 35 % over one accumulator. This is likely due to fewer data dependencies, allowing the CPU to better hide instruction latency.



■ **Figure 7** Effect of fused multiply-add operations and different numbers of accumulation registers. Runtime is measured for computing 128 distances to a single query point. Right: performance normalized by dimension with a logarithmic x-axis.

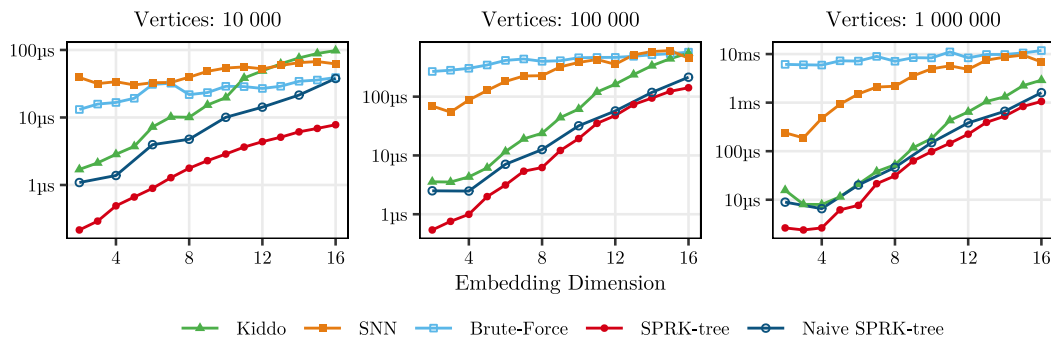
Based on these insights, the SPRK-tree uses a dimension-dependent case distinction. For $d < 6$ the SPRK-tree uses the FMA-based distance computation. For dimensions up to $d = 32$, it switches to the inner product method with 2 accumulators. Beyond that, it uses 4 accumulators. We note that the optimal number of accumulators is architecture-dependent, as it interacts with register pressure and the compiler’s auto-vectorization logic.

Cumulative Optimization Effects. In Figure 8, we compare the full SPRK-tree against a naive variant that retains all algorithmic optimizations but disables the low-level optimizations from Section 2.5, including the memory layout and access pattern optimizations. The gains are most pronounced for low point counts, where for example even writing results to the output buffer constitutes a significant fraction of the total query time. Notably, the naive SPRK-tree already performs well for small point counts and high dimensions. This demonstrates the effectiveness of the algorithmic optimizations alone, with the low-level engineering further improving the performance by up to an order of magnitude at small point counts.

4 Conclusion

While developed primarily for repelling-force computation in graph embeddings, the SPRK-tree proves to be an efficient, general-purpose spatial index across a wide range of applications. Even in extreme scenarios, such as high-dimensional real-world datasets, the SPRK-tree remains competitive with the best-performing methods. This is achieved with a combination of algorithmic optimizations, such as the spherical pruning and SNN buckets, and low-level engineering that maximizes hardware utilization. Integrating the SPRK-tree should allow to substantially improve the performance of applications across various domains.

Another common strategy for high-dimensional regimes is to resort to approximate queries. Our initial attempts in this direction were rather unpromising, due to a significant drop in embedding quality for only small performance gain. However, additional work is warranted to fully evaluate the potential of this direction.



■ **Figure 8** SPRK-tree query time with and without the optimizations from Section 2.5. The naive variant retains all algorithmic optimizations but disables low-level performance optimizations.

References

- 1 S. Arya and D.M. Mount. Algorithms for fast vector quantization. In *Data Compression Conference (DCC)*, pages 381–390, 1993. doi:10.1109/DCC.1993.253111.
- 2 Martin Aumüller, Erik Bernhardsson, and Alexander John Faithfull. ANN-benchmarks: A benchmarking tool for approximate nearest neighbor algorithms. *Information Systems*, 87, 2020. doi:10.1016/j.is.2019.02.006.
- 3 Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975. doi:10.1145/361002.361007.
- 4 Jon Louis Bentley and Jerome H. Friedman. Data structures for range searching. *ACM Computing Surveys*, 11(4):397–409, 1979. doi:10.1145/356789.356797.
- 5 Jose Luis Blanco and Pranjali Kumar Rai. nanoflann: a C++ header-only fork of FLANN, a library for nearest neighbor (NN) with kd-trees. <https://github.com/jlblancoc/nanoflann>, 2014.
- 6 Thomas Bläsius, Tobias Friedrich, Maximilian Katzmann, Ulrich Meyer, Manuel Penschuck, and Christopher Weyand. Efficiently generating geometric inhomogeneous and hyperbolic random graphs. *Network Science*, 10(4):361–380, 2022. doi:10.1017/nws.2022.32.
- 7 Thomas Bläsius, Tobias Friedrich, Anton Krohmer, and Sören Laue. Efficient embedding of scale-free graphs in the hyperbolic plane. *IEEE Transactions on Networking (TON)*, 26(2):920–933, 2018. doi:10.1109/TNET.2018.2810186.
- 8 Thomas Bläsius, Jean-Pierre von der Heydt, Maximilian Katzmann, and Nikolai Maas. Weighted embeddings for low-dimensional graph representation. In *AAAI Conference on Artificial Intelligence (AAAI)*, pages 15587–15595, 2025. doi:10.1609/aaai.v39i15.33711.
- 9 Boost Community. Boost C++ libraries, 2026. URL: <https://github.com/boostorg/boost>.
- 10 Sergey Brin. Near neighbor search in large metric spaces. In *Very Large Data Bases Conference (VLDB)*, pages 574–584, 1995. doi:10.5555/645921.673006.
- 11 Karl Bringmann, Ralph Keusch, and Johannes Lengler. Geometric inhomogeneous random graphs. *Theor. Comput. Sci.*, 760:35–54, 2019. doi:10.1016/J.TCS.2018.08.014.
- 12 Xinye Chen and Stefan Güttel. Fast and exact fixed-radius neighbor search based on sorting. *PeerJ Comput. Sci.*, 10:e1929, 2024. doi:10.7717/PEERJ-CS.1929.
- 13 Scott Donnelly. Kiddo. URL: <https://github.com/sdd/kiddo>.
- 14 Dheeru Dua and Casey Graff. UCI machine learning repository, 2017. URL: <https://archive.ics.uci.edu>.

- 15 J. Elseberg, S. Magnenat, R. Siegwart, and A. Nüchter. Comparison of nearest-neighbor-search strategies and implementations for efficient shape registration. *Journal of Software Engineering for Robotics (JOSER)*, 3(1):2–12, 2012. doi:10.6092/JOSER_2012_03_01_P2.
- 16 Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD)*, pages 226–231, 1996. URL: <http://www.aaai.org/Library/KDD/1996/kdd96-037.php>.
- 17 Michael J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, 21(9):948–960, 1972. doi:10.1109/TC.1972.5009071.
- 18 Santo Fortunato. Community detection in graphs. *Physics Reports*, 486(3):75–174, 2010. doi:10.1016/j.physrep.2009.11.002.
- 19 Jerome H. Friedman, Jon Louis Bentley, and Raphael A. Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Transactions on Mathematical Software (TOMS)*, 3(3):209–226, 1977. doi:10.1145/355744.355745.
- 20 Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. In *Symposium on Foundations of Computer Science (FOCS)*, pages 285–298. IEEE Computer Society, 1999. doi:10.1109/SFFCS.1999.814600.
- 21 Volker Gaede and Oliver Günther. Multidimensional access methods. *ACM Comput. Surv.*, 30(2):170–231, 1998. doi:10.1145/280277.280279.
- 22 Geofabrik Gmbh. OSM Data Germany. URL: <https://download.geofabrik.de/>.
- 23 Aditya Grover and Jure Leskovec. node2vec: Scalable feature learning for networks. In *ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD)*, pages 855–864, 2016. doi:10.1145/2939672.2939754.
- 24 Antonin Guttman. R-trees: A dynamic index structure for spatial searching. In *ACM SIGMOD Conference (SIGMOD)*, pages 47–57, 1984. doi:10.1145/602259.602266.
- 25 HOminus. Neighbourhood. URL: <https://github.com/HOminus/neighbourhood>.
- 26 Markus Ihmsen, Jens Orthmann, Barbara Solenthaler, Andreas Kolb, and Matthias Teschner. SPH fluids in computer graphics. In *Eurographics State of the Art Reports*, pages 21–42. Eurographics Association, 2014. doi:10.2312/EGST.20141034.
- 27 Piotr Indyk and Rajeev Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. In *Symposium on the Theory of Computing (STOC)*, pages 604–613, 1998. doi:10.1145/276698.276876.
- 28 Intel. Intel® Architecture Instruction Set Extensions Programming Reference.
- 29 Intel. Intel® AVX-512 Instructions. URL: <https://www.intel.com/content/www/us/en/developer/articles/technical/intel-avx-512-instructions.html>.
- 30 Intel. Intel® Intrinsics Guide. URL: https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html#text=vcompress&ig_expand=1332.
- 31 Mathieu Jacomy, Tommaso Venturini, Sebastien Heymann, and Mathieu Bastian. Forceatlas2, a continuous graph layout algorithm for handy network visualization designed for the gephi software. *PLOS ONE*, 9(6):1–12, June 2014. doi:10.1371/journal.pone.0098679.
- 32 Henrik Wann Jensen. Global illumination using photon maps. In *Eurographics Symposium on Rendering (EGSR)*, pages 21–30, 1996. doi:10.1007/978-3-7091-7484-5_3.
- 33 Tobias Kempf and Dennis Kobert. wembed-pdf/rembed. Software, swhId: swh:1:dir:48b2e0e7921e86d574d2f7d019c422dddf8186dd (visited on 2026-08-17). URL: <https://github.com/wembed-pdf/rembed>, doi:10.4230/artifacts.27687.
- 34 Tobias Kempf and Dennis Kobert. wembed-pdf/sprk. Software, swhId: swh:1:dir:293dee235b22ac27188619ae7c1b50797096641e (visited on 2026-08-17). URL: <https://github.com/wembed-pdf/sprk>, doi:10.4230/artifacts.27686.
- 35 Margaret Lawson, William Gropp, and Jay F. Lofstead. Exploring spatial indexing for accelerated feature retrieval in hpc. In *IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*, pages 605–614, 2022. doi:10.1109/CCGrid54584.2022.00070.

- 36 David Liben-Nowell and Jon M. Kleinberg. The link-prediction problem for social networks. *Journal of the Association for Information Science and Technology (JASIST)*, 58(7):1019–1031, 2007. doi:10.1002/asi.20591.
- 37 Ilya Makarov, Dmitrii Kiselev, Nikita Nikitinsky, and Lovro Subelj. Survey on graph embeddings and their applications to machine learning problems on graphs. *PeerJ Computer Science*, 7:e357, 2021. doi:10.7717/peerj-cs.357.
- 38 Stephen M. Omohundro. Five balltree construction algorithms. Technical report, International Computer Science Institute, 1989. URL: https://steveomohundro.com/wp-content/uploads/2009/03/omohundro89_five_balltree_construction_algorithms.pdf.
- 39 F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011. doi:10.5555/1953048.2078195.
- 40 Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. Deepwalk: online learning of social representations. In *ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD)*, pages 701–710, 2014. doi:10.1145/2623330.2623732.
- 41 Harald Prokop. Cache-oblivious algorithms. Master’s thesis, Massachusetts Institute of Technology, 1999.
- 42 Hanan Samet. An Overview of Quadtrees, Octrees, and Related Hierarchical Data Structures. In Rae A. Earnshaw, editor, *Theoretical Foundations of Computer Graphics and CAD*, pages 51–68. Springer Berlin Heidelberg, 1988. doi:10.1007/978-3-642-83539-1_2.
- 43 Hanan Samet. *Foundations of multidimensional and metric data structures*. Morgan Kaufmann series in data management systems. Academic Press, 2006.
- 44 Hans Tangelder and Andreas Fabri. dD spatial searching. In *CGAL User and Reference Manual*. CGAL Editorial Board, 6.1.1 edition, 2026. URL: <https://doc.cgal.org/6.1.1/Manual/packages.html#PkgSpatialSearchingD>.
- 45 Tavian Barnes. Acap vp-tree. URL: <https://github.com/tavianator/acap>.
- 46 OpenBLAS Development Team. OpenMathLib/OpenBLAS. URL: <https://github.com/OpenMathLib/OpenBLAS>.
- 47 Anton Tsitsulin, Davide Mottin, Panagiotis Karras, and Emmanuel Müller. VERSE: versatile graph embeddings from similarity measures. In *Proceedings of the 2018 World Wide Web Conference (WWW)*, pages 539–548, 2018. doi:10.1145/3178876.3186120.
- 48 Laurens van der Maaten. Accelerating t-sne using tree-based algorithms. *Journal of Machine Learning Research (JMLR)*, 15(1):3221–3245, 2014. doi:10.5555/2627435.2697068.
- 49 Peter N. Yianilos. Data structures and algorithms for nearest neighbor search in general metric spaces. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 311–321, 1993. URL: <http://dl.acm.org/citation.cfm?id=313559.313789>.