## A DiskANN Algorithm

**Algorithm 1** RobustPruning$(i, U, \alpha, R)$

**Input** Vertex $i$, candidate neighbor set $U$, pruning parameter $\alpha$, degree limit $R$(default $R$ is $n$ if not given)
**Result** Update $N_{out}(i)$, the set of out-neighbors of $i$
$U \leftarrow U \cup N_{out}(i)$
$N_{out}(i) \leftarrow \varnothing$
**while** $U \neq \varnothing$ and $|N_{out}(i)| < R$ **do**
  $v \leftarrow \text{argmin}_{v \in U} D(x_v, x_i)$
  $N_{out}(i) \leftarrow N_{out}(i) \cup v$
  $U \leftarrow U \backslash v$
  $U \leftarrow \{v' \in U : D(x_v, x_{v'}) \cdot \alpha > D(x_i, x_{v'})\}$
**end while**

**Algorithm 2** GreedySearch$(s, q, L)$

**Input** Graph $G = (V, E)$, seed $s$, query point $q$, queue length limit $L$
**Output** visited vertex list $U$
$A \leftarrow \{s\}$
$U \leftarrow \varnothing$
**while** $A \backslash U \neq \varnothing$ **do**
  $v \leftarrow \text{argmin}_{v \in A \backslash U} D(x_v, q)$
  $A \leftarrow A \cup N_{out}(v)$
  $U \leftarrow U \cup v$
  **if** $|A| > L$ **then**
    $A \leftarrow$ top $L$ closest vertices to $q$ in $A$
  **end if**
**end while**
sort $U$ in increasing distance from $q$
**return** $U$

**Algorithm 3** DiskANN indexing algorithm (with fast preprocessing)

**Input** Point set $P = \{x_1 ... x_n\}$, degree limit $R$, queue length $L$
**Output** A proximity graph $G = (V, E)$ where $V = \{1..n\}$ are associated with point sets $P$.
$G \leftarrow$ randomly sample a $R$-regular graph on vertex set $V = \{1..n\}$
$s \leftarrow$ vertex for the point closest to the centroid of $P$
**for** $k = 1$ **to** $2$ **do**
  $\sigma \leftarrow$ a random permutation of $[1...n]$
  **for** $i = 1$ **to** $n$ **do**
    $U \leftarrow GreedySearch(s, x_{\sigma(i)}, L)$
    $RobustPruning(\sigma(i), U, \alpha, R)$
    **for** vertex $j$ in $N_{out}(\sigma(i))$ **do**
      $N_{out}(j) \leftarrow N_{out}(j) \cup \sigma(i)$
      **if** $|N_{out}(j)| > R$ **then**
        $RobustPruning(j, N_{out}(j), \alpha, R)$
      **end if**
    **end for**
  **end for**
**end for**

**Algorithm 4** DiskANN indexing algorithm (with slow preprocessing)

**Input** Vertex set $P = \{x_1 ... x_n\}$, parameters: degree limit $R$
**Output** A proximity graph $G = (V, E)$ where $V = \{1..n\}$ are associated with point sets $P$.
$s \leftarrow$ vertex for the point closest to the centroid of $P$
**for** $i = 1$ **to** $n$ **do**
  $N_{out}(i) \leftarrow RobustPruning(i, V, \alpha, R)$
**end for**

In this section we give an overview of the DiskANN procedures. For the full description the reader is referred to the original paper [19].

The data structure is based on a directed graph $G$ over the set $P$, i.e., the set of vertices $V$ of $G$ are associated with the set of points $P$. After the graph is constructed, to answer a given query $q$, the algorithm performs search starting from some vertex $s$. In what follows we describe the search and insertion procedure in more detail.

The search procedure, $GreedySearch(s, q, L)$, has the following parameters: the start vertex $s$, the query point $q$, and the queue size $L$. It performs a best-first-search using a queue of with a bounded length $L$, until the $L$ vertices $v$ with the smallest value of $D(v, q)$ seen so far are all scanned. Upon completion, it returns a list of vertices in an increasing distance from $q$ where the first vertex (or the first $k$ vertices) are answers for the query. Note that as long as the graph is connected, the procedure runs for at least $L$ steps. The total running time of the procedure is bounded by the number of steps times the out-degree bound of the graph $G$.

The construction of the graph $G = (V, E)$ is done by a repeated invocation of a procedure called $RobustPruning$. For any vertex $v$, a set of vertices $U$ (specified later), and parameters $\alpha > 1$ and $R$, $RobustPruning(v, U, \alpha, R)$ proceeds as follows. First, the set $U$ is sorted in the increasing order of the distance to $v$. The algorithm traverses this sequence in order. After encountering a new vertex $u$, the algorithm deletes all other vertices $w$ from $U$ such that $D(u, w) \cdot \alpha < D(v, w)$. Finally, the node $v$ is connected to all vertices in $U$ that have not been pruned.

The starting point of the DiskANN data structure construction algorithm[4] is the following simple procedure: for each vertex $v$, execute $RobustPruning(v, U, \alpha, R)$ with $U = V$ and $R = n$. That is, robust pruning is applied to *all* vertices in the graph. We refer to this procedure as *slow preprocessing*, as it can be seen that a naive implementation of this method takes time $O(n^3)$. Although the construction time is slow, we show that this construction method provably constructs a graph whose degree depends only logarithmically on the aspect ratio of the graph (assuming constant doubling dimension), and guarantees that the greedy search procedure has polylogarithmic running time. We note that this result is inspired by an observation in [19] about convergence of greedy search in a logarithmic number of steps, though to obtain our result we also need to bound the degree of the search graph and analyze the approximation ratio.

Since the slow-preprocessing-algorithm is too slow in practice, the authors of [19] propose a faster heuristic method to construct the graph $G$, which we call *fast preprocessing* method. At the beginning, the graph $G$ is initialized to be a random $R$-regular graph. Then the construction of the graph $G = (V, E)$ is done incrementally. The construction algorithms make two passes of the point set in random order. For each vertex $v$ met, the algorithm computes a set of vertices $U = GreedySearch(s, x_v, L)$ (for some starting vertex $s$) and then calls the pruning procedure on $U$, not $V$. That is, it executes $RobustPruning(v, U, \alpha, R)$. After pruning is performed, the insertion procedure adds both edges $(v, u)$ and $(u, v)$ for all vertices $u \in U$ output by the prunning procedure. Finally, if the degree of any of $u \in U$ exceeds a threshold $R$, then the set of neighbors of $u$ is pruned via $RobustPruning(u, N_{out}(u), \alpha, R)$ as well. This construction method is implemented and evaluated in the paper.

## B Dependence on aspect ratio $\Delta$

Consider the following 1-dimensional data set with $n = 2k$ points located at $\{x_i\}_{i=1}^n$, where

$$x_i = \begin{cases} \alpha^i & \text{for } 1 \leqslant i \leqslant k \\ 2\alpha^k + \alpha^k\beta - \alpha^{2k+1-i} & \text{for } k < i \leqslant n \end{cases}$$

and $\beta = \max(\frac{1}{\alpha-1}, \alpha - 1)$. This is a symmetric line starting from 0 to $(2 + \beta)\alpha^k$. Each point's distance toward the closer endpoint is $\alpha$ times larger than that of the previous point.

**Lemma B.1.** *The graph $G = (V, E)$ built on the above instance using the slow preprocessing version of DiskANN satisfies the following properties:*

*(1) For any $i \in [k + 1, n]$, $(i, k) \in E$ and $(i, j) \notin E$ for any $j < k$*

*(2) For any $j < i \leqslant k$, $(i, j) \in E$ if and only if $j = i - 1$*

*Since the $x_i$'s are symmetric, the same properties also hold in the other direction.*

*Proof.* (1): For any $i \in [k + 1, n]$, we can check that no vertex $j$ such that $k < j < i$ can delete $k$ from $i$'s neighborhood, because $\frac{x_j - x_k}{x_i - x_k} > \frac{\alpha^k\beta}{\alpha^k\beta + \alpha^k} \geqslant \frac{1}{\alpha}$. Thus, we have $(i, k) \in E$. Similarly, we

[4] See the discussion in Section 2.2 of [19].

have that $k$ will delete any vertex $j < k$ from $i$'s neighborhood because $\frac{x_k - x_j}{x_i - x_j} \leqslant \frac{\alpha^k}{\alpha^k + \alpha^k \beta} \leqslant \frac{1}{\alpha}$. These two inequalities use that $\beta = \max(\frac{1}{\alpha-1}, \alpha - 1)$

(2): For any $i \in [1, k]$, $x_{i-1}$ is the closest point on $x_i$'s left, so $(i, i-1) \in E$. Then, for any $j < i-1$, $j$ will be deleted from $i$'s neighbor by $i - 1$ because $\frac{x_{i-1} - x_j}{x_i - x_j} < \frac{\alpha^{i-1}}{\alpha^i} = \frac{1}{\alpha}$. □

*Proof of Theorem 3.5.* Based on the graph properties in Lemma B.1, let us determine the length of the shortest path from a starting point $s$ (selected arbitrarily by the DiskANN algorithm) to a constant approximate nearest neighbor of a given query $q$. We select our query $q$ to be either 0 or $2\alpha^k + \alpha^k\beta$, i.e., one of the two endpoints of the data set, whichever is farther from $x_s$. To find an $O(1)$-approximate nearest neighbor of the query $q$ within $l$ steps of GreedySearch, there should be at least one path with less than $l$ hops from $s$ to $q$'s approximate nearest neighbor. WLOG, let's assume $q = 0$ and $s > k$. By Property (1) of Lemma B.1, among $\{1...k\}$, the vertex $k$ is the only neighbor of any vertex on the right of $k$. By Property (2) of Lemma B.1, for any vertex $i \in [1, k]$, its only neighbor on its left is $i - 1$. Therefore, it takes at least $l \geqslant \Omega(n)$ and $l \geqslant \Omega(\log \Delta)$ steps for slow preprocessing DiskANN with GreedySearch to reach any $O(1)$- approximate nearest neighbor in this constructed instance. □

# C More experimental results

## C.1 Hard instance for KD-Tree based nearest neighbor search algorithm

Some nearest neighbor search algorithms use KD-tree to find the entry point for greedy search. In this case, we design a hard instance where KD-tree cannot get good entry point close to the nearest neighbor. See Figure 7. We draw our constructed instance for $n = 10^6$. It is easy to mimic this instance for other data sizes.

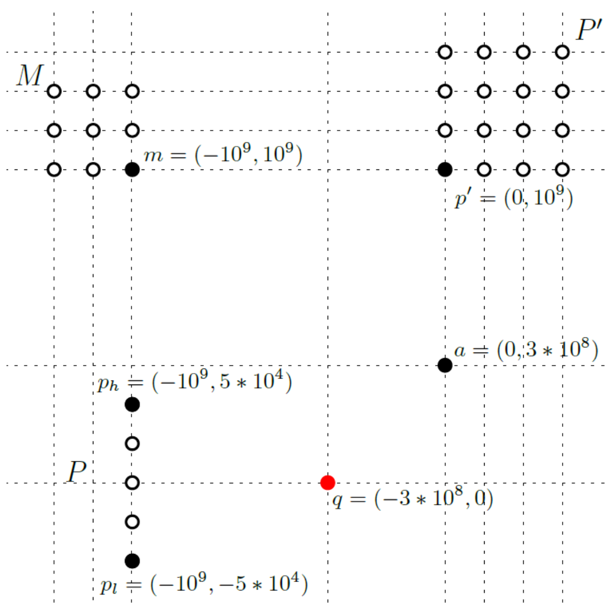


Figure 7: Our constructed hard instance for SPTAG on the scale $n = 10^6$. The instance lives in a two-dimensional Euclidean space, and therefore has a constant doubling dimension. Black dots (solid or hollow) are points in the database, the red dot is the query point. **Grids are only used to show the layout structure.** See section C.1 for detailed description.

**Description of instance in Figure 7** Our instance has 6 dimensions, where the first two dimensions of the instance are similar to Figure 2 but with different choices of parameters. Again, the vertex set $V$ consists of three sets of points $M, P, P'$ (with size $0.1n, 0.1n, 0.8n$ respectively) and a single answer point $a$: $M$ is a $\sqrt{|M|} \times \sqrt{|M|}$ square grid with unit length 1 and with bottom right corner $m = (-10^9, 10^9)$. $P$ is a vertical chain of points consisting of unit intervals from $p_l = (-10^9, -0.05n)$ to $p_h = (-10^9, 0.05n)$. $P'$ is a $\sqrt{|P'|} \times \sqrt{|P'|}$ square grid with unit length 1 whose bottom left corner is $p' = (0, 10^9)$. The answer point is $a = (0, 3 * 10^8)$. The query point is $q = (-3 * 10^8, 0)$. For a vertex $v \in M \cup P' \cup \{a\}$, each of $v$'s other 4 coordinates are sampled from a uniform distribution supported on $[5 * 10^7, 6 * 10^7]$. For a vertex $v \in P$, its other 4 coordinates are sampled from a uniform distribution supported on $[10^7, 2 * 10^7]$. And for point $q$, its other 4 coordinates are all 0. Note that though such choices of parameters are tailored to the implementation of KD-trees used in SPTAG and EFANNA in [35], some general ideas are reusable for constructing hard instances for other implementations.

**Intuition for instance in Figure 7** The main reason why we construct a new example here is to handle the use of KD-tree to search for a good starting point. Implementation of KD-tree provided in [35] always randomly picks one of the top 5 dimensions with the largest variance as the splitting dimension, and divides the vertices into two halves at their mean. In Figure 7, because of the separation on the other 4 dimensions, our construction can make sure that KD-tree quickly gets to a subtree with vertices only in $P$. Then KD-tree will only horizontally split the chain from $p_h$ to $p_l$ or split via the other 4 dimensions. In the vertical axis, the coordinates for vertices in $P$ and $a$ are quite close, so KD-tree will assign them a low distance estimation based on only a horizontal split, resulting in KD-tree scanning all vertices on the chain before scanning other vertices outside of the set $P$. As long as we make the KD-tree select a vertex in $P$ as the starting point, we can ensure (as in Figure 2) that $GreedySearch$ will scan all vertices in $P$ before going to $M$ or $P'$.

### C.2 More experiments on other popular nearest neighbor search algorithms

We further test the other 7 popular nearest neighbor search algorithms studied in the survey [35]. We use the same setting as in Section 4. We run each algorithm for 20 different data sizes $n \in \{10^5, 2 \cdot 10^5, \ldots, 2 \cdot 10^6\}$ using hard instances in Figure 4, Figure 2, or Figure 7 (introduced in Appendix C.1). We plot the Recall@5 rate for answering the query with queue length $L$ equal to $pn$ where the percentage $p$ is enumerated from the set $1\%, 2\%, \ldots, 12\%, 15\%, 18\%, 20\%, 30\%, 40\%, 50\%$.

**NGT [18]** We use NGT's implementation from the authors' GitHub repository [17]. We run NGT on the hard instance in Figure 2, using all default parameters as stated in GitHub's readme, except that we use the command "-i g" to generate only the graph index, because of our focus on graph-based nearest neighbor search algorithms. We use command "-p 1" to set the number of threads to 1. We run this experiment 10 times and report the average recall.

**SSG [13]** We use SSG implementation from the authors GitHub repository [12]. We run SSG on the hard instance in Figure 4, using parameters $K = 200, L = 200, iter = 12, S = 10, R = 100$ (for building KNN graph) $L = 100, R = 50, Angle = 60$ (constructing SSG). The parameters chosen here are copied from the author's selected parameters for data set SIFT1M [21], whose data size is close to ours. We run this experiment 10 times using different random seeds and report the average recall.

**KGraph** We use KGraph implementation due to [35], from the GitHub repository [36]. We run KGraph on the hard instance in Figure 4 using parameter $K = 100, L = 130, iter = 12, S = 20, R = 50$. Parameters here are copied from the authors' selected parameters used for their synthetic data set named "$n$_1000000", whose size is close to ours. We run this experiment 5 times using different random seeds and report the average recall.

**DPG [25]** We use DPG implementation due to [35], from the GitHub repository [36]. We run DPG on the hard instance in Figure 4 using parameter $K = 100, L = 100, iter = 12, S = 20, R = 300$. Parameters here are copied from the authors' selected parameters for their synthetic dataset named "$n$_1000000", whose size is close to ours. We run this experiment 10 times using different random seeds and report the average recall.

**NSW [28]** We use NSW's implementation due to [35], from the GitHub repository [36]. We run NSW on the hard instance in Figure 4 (with a different vertex permutation) using parameter $max\_m0 = 100, ef\_construction = 400$. Parameters here are copied from the author's selected parameters for their synthetic dataset named "$n$_1000000", whose data size is close to ours. We run this experiment 5 times using different random seeds and report the average recall.

**SPTATG-KDT [7]** We use SPTATG-KDT implementation due to [35], from the GitHub repository [36]. We run SPTATG-KDT on the hard instance in Figure 7 using parameters $KDT\_number =$

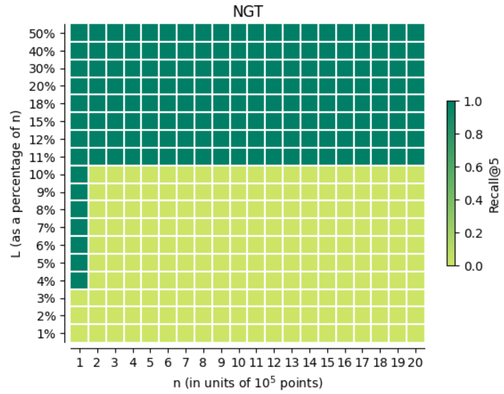


Figure 8: Results for running NGT algorithm on the family of instances in Figure 2. The horizontal axis represents the data size $n$, in multiples of $10^5$ points. The vertical axis represents the size of the search queue length $L$ in terms of the percentage of the data size. each pixel represents a query result, with its $Recall@5 \in [0,1]$ mapping to the spectrum on the right. We run NGT algorithm 10 times and report the average recall rate.

$1, TPT\_number = 16, TPT\_leaf\_size = 1500, scale = 2, CEF = 1500$. Parameters here are copied from the authors' selected parameters for their synthetic dataset named "$n$_1000000", whose size is close to ours. We run this experiment 5 times and report the average recall.

**EFANNA [11]** We use EFANNA's implementation due to [35], from the GitHub repository [36]. We run EFANNA on the hard instance in Figure 7 using parameter $nTrees = 4, mLevel = 8, K = 80, L = 140, iter = 7, S = 25, R = 150$. Parameters here are copied from the authors' selected parameters for their synthetic dataset named "$n$_1000000", whose size is close to ours. We run this experiment 10 times and report the average recall.

Experimental results for these 7 algorithms are plotted in Figure 8, Figure 9, and Figure 10. We can see that all algorithms achieve suboptimal Recall@5 rates until the queue length $L$ is greater than $10\%$ of the data size.

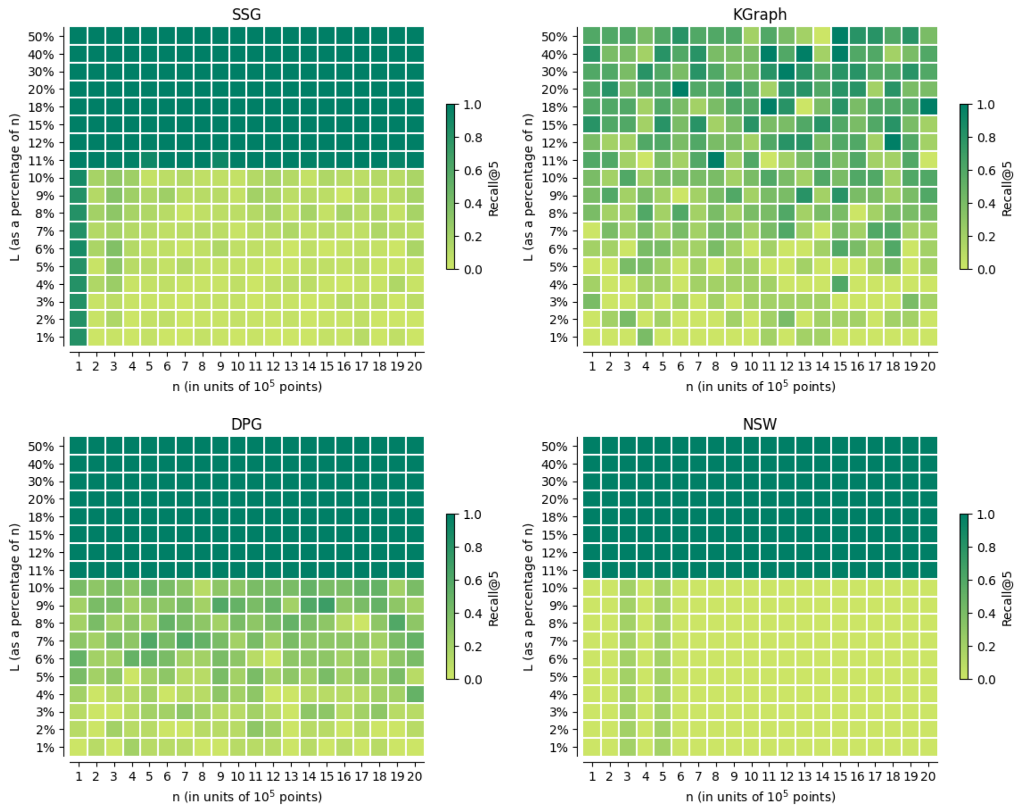


Figure 9: Results for running SSG, KGraph, DPG, NSW algorithms on the family of instances in Figure 4. The horizontal axis represents the data size $n$, in multiples of $10^5$ points. The vertical axis represents the size of the search queue length $L$ in terms of the percentage of the data size. Each pixel represents a query result, with its $Recall@5 \in [0, 1]$ mapping to the spectrum on the right. We run each algorithm 10 (or 5) times and report the average recall rate.

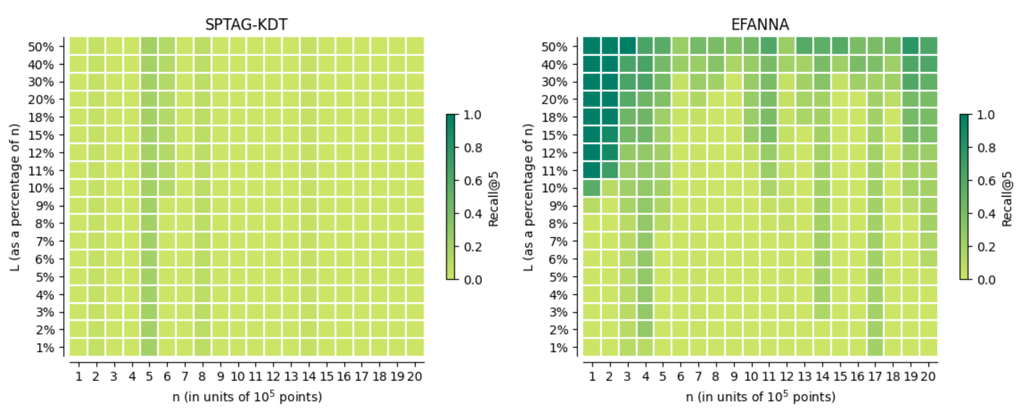


Figure 10: Results for running SPTAG-KDT and EFANNA algorithms on the family of instances in Figure 7. The horizontal axis represents the data size $n$, in multiples of $10^5$ points. The vertical axis represents the size of the search queue length $L$ in terms of the percentage of the data size. Each pixel represents a query result, with its $Recall@5 \in [0, 1]$ mapping to the spectrum on the right. We run each algorithm 10 (or 5) times and report the average recall rate.