

EMPIRICAL COMPLEXITY OF COMPARATOR-BASED NEAREST NEIGHBOR DESCENT

JACOB D. BARON R.W.R. DARLING

National Security Agency, Fort George G. Meade, MD 20755-6844, USA

ABSTRACT. A Java parallel streams implementation of the K -nearest neighbor descent algorithm is presented using a natural statistical termination criterion. Input data consist of a set S of n objects of type V , and a `Function<V, Comparator<V>>`, which enables any $x \in S$ to decide which of $y, z \in S \setminus \{x\}$ is more similar to x . Experiments with the Kullback-Leibler divergence `Comparator` support the prediction that the number of rounds of K -nearest neighbor updates need not exceed twice the diameter of the undirected version of a random regular out-degree K digraph on n vertices. Overall complexity was $O(nK^2 \log_K(n))$ in the class of examples studied. When objects are sampled uniformly from a d -dimensional simplex, accuracy of the K -nearest neighbor approximation is high up to $d = 20$, but declines in higher dimensions, as theory would predict.

Keywords: similarity search, nearest neighbor, ranking system, triplet comparison, comparator, random graph, proximity graph, expander graph

MSC class: Primary: 90C35; Secondary: 06A07

CONTENTS

1. Introduction	1
2. K -nearest neighbors based on <code>Comparators</code>	3
3. K -NN descent algorithm with a statistical stopping rule	4
4. A Java implementation of K -NN descent	5
5. Conclusions and future work	7
References	8

1. INTRODUCTION

1.1. Context. Baron and Darling [3] provided a theoretical analysis of the **K -nearest neighbor descent** (K -NN Descent) algorithm for K -nearest neighbor approximation proposed and implemented by Dong, Charikar, and Li [7].

This sequel reports on a generic Java parallel streams implementation of K -NN Descent, which was written to support a forthcoming implementation of the partitioned nearest neighbors local depth algorithm [4]. While testing this implementation, we acquired statistical data which shed light on the performance of K -NN Descent, under a new termination criterion. This brief report does not attempt comparison of K -NN Descent with other algorithms, as

TABLE 1. Comparison among four K -NN Descent implementations.

Authors	Language	Asymmetric?	Parallelization?	Termination
Dong et al [7]	C++	no	OpenMP & map-reduce	δ -proportion update
McInnes [14]	Python	no	none	no possible update
Kluser et al [12]	C	no	none	no possible update
This paper	Java	yes	fork-join pool	statistical criterion

is reported in [7]. Nor shall we outline the different approaches to K -nearest neighbor approximation, briefly surveyed by Aumüller, Bernhardsson and, Faithfull [1]. We mention the recent competition [13] to surpass the industry leader FAISS [11], in the case of a billion dense vectors in dimension 96 to 256, under the ℓ_2 norm.

1.2. Previous implementations. A high level summary of previous K -NN Descent implementations is shown in Table 1, along with our own in the final row.

1.2.1. Original implementation for metrics. A sophisticated OpenMP and map-reduce implementation of K -NN Descent is described by Dong et al [7]. These authors employ optimizations applicable to symmetric similarity functions, and employ two stopping criteria, both of which are different to ours. The authors describe in detail the application to five well-studied data sets of sizes between 28,755 and 857,820 using several symmetric similarity measures, and compare K -NN Descent with Recursive Lanczos Bisection and Locality Sensitive Hashing.

1.2.2. Python implementation. McInnes created the widely-used `pynndescent` Python version [14], which is used in UMAP [15]. It assumes that similarity is obtained from a metric, of which 22 examples are available in the code. This is one of 19 single-threaded approximate K -NN Python algorithms among benchmarks at [5].

1.2.3. Single-threaded C Implementation. Kluser et al [12] describe a runtime-optimized C implementation for the ℓ_2 -distance metric, and report performance improvements over the two versions above. Their approach increases locality by improving the otherwise irregular memory access pattern.

1.3. Novelty of our implementation. Here are some more details, beyond Table 1, of what distinguishes our implementation from the others.

- (a) **Non-metric:** Input data consist of a set S of n objects of type V , and a `Function<V, Comparator<V>>`, which enables any $x \in S$ to decide which of $y, z \in S \setminus \{x\}$ is more similar to x . This “triplet comparison” has more general application than similarity induced by a symmetric distance function, as we discuss in [4], but disallows optimizations based on symmetric numerical similarity functions, used in the works cited above.
- (b) **Stopping criterion:** Termination depends on a statistical criterion, presented in Section 3.3, applied to a sampled quantity called the friend clustering rate. By contrast, other implementations continue until no further updates are possible, except for a variant by [7] which stops when no more than a proportion δ of points allow updates.

- (c) **Parallelism:** By casting the algorithm into a functional programming framework, we enable the Java Virtual Machine to distribute tasks among threads via a fork join pool, invisible to the programmer. Speedup due to parallelism is reported below.

2. K -NEAREST NEIGHBORS BASED ON COMPARATORS

2.1. Setting. Input data consist of a set S of n objects of type V , and what we call a **ranking system** [3], which attaches to each $x \in S$ a total order¹ \prec_x on $S \setminus \{x\}$; here $y \prec_x z$ is interpreted to mean that y is more similar to x than z is. In data science, this is called *triplet comparison*. The computer science equivalent is a **Function** $\langle V, \text{Comparator}\langle V \rangle \rangle$. Here we map each $x \in S$ to a specific **Comparator** [10], whose **compare** method depends on x . Formally **compare**($x; y, z$) < 0 means $y \prec_x z$, and **compare**($x; y, z$) > 0 means $z \prec_x y$, for distinct $x, y, z \in S$.

Such a family of **Comparators** gives an orientation² of the line graph³ $\mathcal{L}(K_n)$, called the *ranking digraph*. The relation $y \prec_x z$ is interpreted as an arc $xy \rightarrow xz$. See Figure 1 for an example. No comparison between xy and zw is provided if $\{x, y, z, w\}$ is a set of size four.

Most authors study the special case of a metric ρ on S , where $y \prec_x z$ means $\rho(x, y) < \rho(x, z)$. Thus a metric (without ties) gives a total order (by distance) on points of the line graph $\mathcal{L}(K_n)$.

Theorem: (Baron and Darling [3, Lemma 5.5]) *The ranking digraph is acyclic if and only if the Comparators arise from some metric.*

2.2. Example of non-metrizable Comparator. For points x, y, z in the interior of the d -dimensional simplex (here $d \geq 3$), choose this **Comparator**: y is closer to x than z is when

$$D(x||y) < D(x||z),$$

where $D(x||y) := \sum_1^d x_i \log(x_i/y_i)$ is Kullback-Leibler (KL) divergence. KL divergence is asymmetric in the pair (x, y) , and does not satisfy the triangle inequality. This example was chosen to dispel the notion that a metric is needed. An example of a ranking digraph from six randomly generated points is shown in Figure 1. The existence of a 3-cycle shows that these **Comparators** are not metrizable. Further examples and properties of ranking digraphs are described in [4].

2.3. K -NN graph. The K -NN graph is the **directed graph** with an arc from each $x \in S$ to the K elements of $S \setminus \{x\}$ most similar to x . Naive computation would take n calls to a **Comparator**-based sorting operation on $n - 1$ objects, which would be $O(n^2 \log n)$ work – or in the special case of a metric, $\binom{n}{2}$ distance evaluations followed by an $O(n^2 \log n)$ sort. An ideal outcome would be to approximate the K -NN graph using $O(n \log_K n)$ calls to a **Comparator**-based sort of $O(K^2)$ objects.

Before we describe how K -NN Descent works, we mention that [4] lists half a dozen data science algorithms besides K -NN Descent that accept triplet comparisons as input, including another nearest neighbor search algorithm [9].

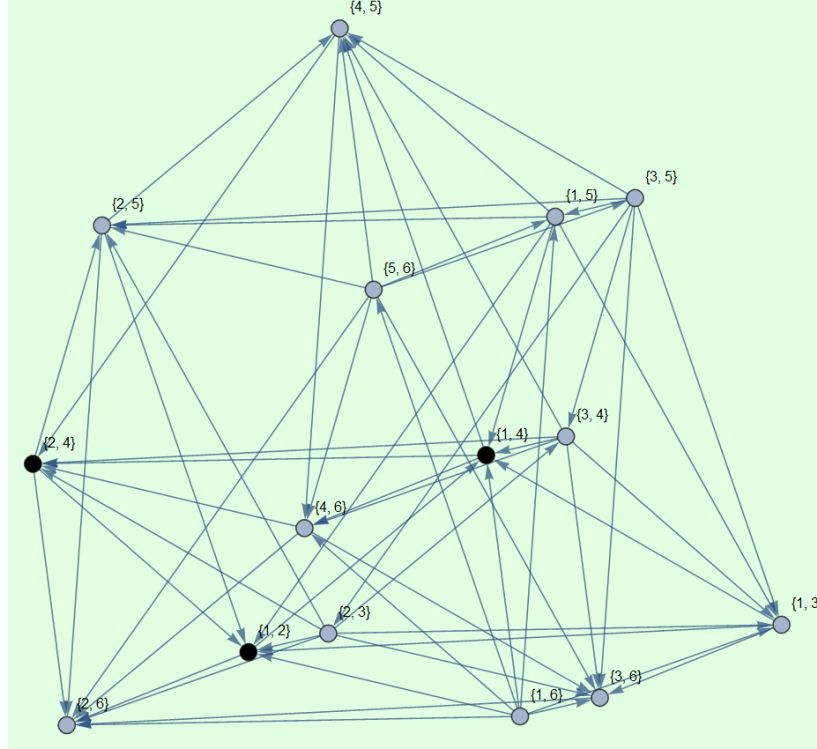
¹By definition, this relation is anti-symmetric and transitive for all x .

²An orientation of a graph G is a digraph obtained by replacing each edge $\{a, b\}$ by one of the arcs $a \rightarrow b$ or $b \rightarrow a$. The textbook [2] explains graph-theoretic terms.

³Undirected edges of the complete graph K_n on S form the points of the line graph $\mathcal{L}(K_n)$, in which $xy \sim xz$ for distinct x, y, z . Here xy is an abbreviation for the edge $\{x, y\}$ of K_n .

FIGURE 1. *Ranking digraph on $\binom{6}{2}$ vertices, forming the unordered pairs from a set $\{x_i\}_{1 \leq i \leq 6}$, with each x_i drawn uniformly at random from a 14-simplex. Vertex label $\{1, 6\}$ (bottom right) refers to the unordered pair $\{x_1, x_6\}$. Its neighbors are $\{1, 2\}, \{1, 3\}, \{1, 4\}, \{1, 5\}$ together with $\{2, 6\}, \{3, 6\}, \{4, 6\}, \{5, 6\}$. Orientation of an arc such as $\{1, 2\} \rightarrow \{1, 4\}$ means that Kullback-Leibler divergences satisfy $D(x_1 \| x_2) < D(x_1 \| x_4)$. Proof by counterexample that this ranking system is not metrizable: vertices marked in black form a cycle*

$$(\{1, 2\} \rightarrow \{1, 4\}, \{1, 4\} \rightarrow \{2, 4\}, \{2, 4\} \rightarrow \{1, 2\}).$$



3. K -NN DESCENT ALGORITHM WITH A STATISTICAL STOPPING RULE

3.1. Friend-of-a-Friend Principle. Dong, Charikar, and Li [7] base K -NN Descent on the *Friend-of-a-Friend Principle*, which states that *a friend of a friend may be suitable as a friend*. The algorithm consists of a sequence of rounds. At each round, we have a digraph on S , with regular out-degree K . Such a graph is called a **K -out** graph [8, Ch. 16]. If $x \rightarrow y$, call y a **friend** of x , and x a **co-friend** of y . If there is a ranking system on S , one K -out graph may be replaced by another K -out graph using a procedure we call **friend set update**.

Friend set update is like a cocktail party, attended by all members of S .

Meet: Each x “meets” all friends of friends, and friends of co-friends, which become new acquaintances.

Update: The new friend set of x consists of the closest K out of all new acquaintances and former friends.

TABLE 2. Ingredients of the Java functional implementation of NND. For example, the Java object **rankingSystem** is an instance of the Java type `Function<V, Comparator<V>>`.

Java Type	Instance Object
<code>List<V></code>	points
<code>Function<V, Comparator<V>></code>	rankingSystem
<code>Map<V, Set<V>></code>	friends, coFriends
<code>Function<V, Set<V>></code>	proposeNewFriendSet

3.2. Initialization of K -NN Descent.

- Each x selects K elements of $S \setminus \{x\}$ uniformly at random as its initial friend set. This digraph is called *random K -out* (Frieze & Karonski [8, Ch. 16]).
- Random K -out is an expander graph, whose undirected version has diameter $\leq \lceil \log_{K-1} n \rceil$ with high probability [3, Appendix].
- **Plausible heuristic:** $2\lceil \log_{K-1} n \rceil$ rounds of cocktail parties (friend updates) should suffice for “everyone to get to know each other”. Twice the diameter allows a “message” to travel from any vertex to its antipode, and back.

3.3. Sampling Method for Termination for K -NN Descent.

- Sample uniformly a point $x \in S$, and two friends $y, z \in S$ of x .
- The *friend clustering rate* is the sample relative frequency that y is a friend or co-friend of z .
- The friend clustering rate is close to zero at the outset, and increases during the algorithm towards a plateau.
- Stop at the first round at which the friend clustering rate does not increase, compared to the previous round.

4. A JAVA IMPLEMENTATION OF K -NN DESCENT

4.1. Java parallel streams. The tools of modern Java [10], including generic types, pure functions, and parallel streams, enable a concise and performant distributed implementation of NND. Four main Java types are listed in the left column of Table 2 in monospaced font, and instances of these types used in NND are listed in the right column in boldface font.

Elements of S have a generic type V , supplied by the invoking class (e.g. strings, vectors, trajectories). A ranking system is a **Function** from V to a **Comparator** of objects of type V . Given three elements x, y, z of S , the assertion that $y \prec_x z$ is equivalent in Java to:

$$\text{rankingSystem.apply}(x).compare(y, z) < 0.$$

The **Comparator** for object x is visible as an `x.getComparator()` method of the class V .

4.2. Distribution of tasks to processors. Initialize the **friends** Map so that the value associated with the key x is a `Set<V>` object containing K elements of $S \setminus \{x\}$ selected uniformly at random. The **coFriends** Map is derived from the **friends** Map.

Given x , the function **proposeNewFriendSet** (bottom of Table 2) compares all the cofriends, friends of friends, and friends of cofriends, with the current friend set of x , and proposes the best K of all these as a new friend set. It is crucial that *the friends of x are not updated at the time the function is called*. The **friends** and **coFriends** Maps remain

TABLE 3. Scaling observed in parallel streaming NND (dual Intel X5660, 12 cores total, JRE 11). Processing times decrease in successive rounds, because duplicate candidates are proposed by different friends. Time to execute a single round of NND scaled linearly with n . When K was doubled, time to execute a round increased by a factor less than four. The FCC column shows the final value of the friend clustering coefficient, which appears to decrease with n . The proportion of the true K nearest neighbors found, among a uniform sample of six points, was typically 95% or better; see Table 4. Results are consistent with the heuristic that $2\lceil\log_K n\rceil$ rounds of NND suffice.

n	K	rounds	$2\lceil\log_K n\rceil$	1st round	last round	FCC
2×10^4	16	5	8	1.9 sec	0.35 sec	0.271
	32	6	6	2.6 sec	0.9 sec	0.264
2×10^5	16	7	10	9.1 sec	5.3 sec	0.210
	32	5	8	26 sec	13 sec	0.231
2×10^6	16	8	12	88 sec	56 sec	0.205
	32	7	10	295 sec	153 sec	0.210
	64	6	8	1059 sec	401 sec	0.215

effectively immutable while all these proposals are constructed in a parallel stream. This is part of the contract of `java.util.stream`, and makes it possible to execute a round of NND in a single line of code, by invoking the `collect()` method of `Stream` [10]:

```
points.parallelStream().collect(Collectors.toMap( $x \rightarrow x, x \rightarrow \text{proposeNewFriendSet.apply}(x)$ )).
```

The `Map<V, Set<V>>` produced by this command becomes the **friends Map** for the next round, and **coFriends** is updated accordingly.

The value type of the **friends Map** is `NavigableSet<V>`, with respect to the `Comparator`. Initial friends are ordered on insertion. During a friend set update at x , each new candidate is compared to the last member of the `NavigableSet` at x , and replaces it if appropriate.

The Java Virtual Machine allocates **proposeNewFriendSet** tasks among the processors and threads at run time. For example, experiments on a 12-core workstation with JRE 11 gave an eightfold speedup⁴, per round, compared to the same code where a single stream was used instead of a parallel stream. With n points and p processors, the speedup should be monotonically increasing in $\frac{n}{p}$ for fixed K , assuming that the limitation is thread contention for access to the **friends** and **coFriends** maps.

4.3. How many rounds of NND are needed? How many rounds of friend updates do we expect before the termination criterion of Section 3.3, is satisfied? Baron and Darling [3], and other citations therein, justify $\lceil\log_K n\rceil$ as an estimate for the diameter of the undirected graph on S whose edges are the pairs $\{x, y\}$, where y is an initial friend of x . In our experiments, the number of rounds never exceeded, but was close to, $2\lceil\log_K n\rceil$; see Table 3.

4.4. Scaling of execution time with n and K : The type V that we chose for our timing experiments was a point on the interior of the 9-dimensional standard simplex in \mathbf{R}^{10} , representing a probability measure on a set of size ten. The ranking system was defined by taking

⁴Here $n = 2 \times 10^6$, $K = 16$. The speedup was much less for smaller n .

TABLE 4. Here $n = 2 \times 10^5$ points were sampled from the $(d-1)$ -dimensional simplex in \mathbf{R}^d , for four different values of d , and K -NN descent was performed for $K = 16$ and $K = 64$. All runs finished within $2\lceil \log_K n \rceil$ (8 or 6) rounds. The FCC row shows the final value of the friend clustering coefficient. The accuracy row shows proportion of the true K nearest neighbors found, among a sample of six points. Note the gradual decline of accuracy with dimension, especially when the dimension $d-1$ exceeds the number K of neighbors.

K	feature	$d = 10$	$d = 20$	$d = 40$	$d = 60$
64	FCC	0.24	0.13	0.08	0.06
	accuracy	1.0	1.0	0.90	0.84
16	FCC	0.21	0.13	0.09	0.08
	accuracy	0.95	0.52	0.43	0.36

$y \prec_x z$ whenever

$$D(x\|y) < D(x\|z)$$

where $D(x\|y)$ denotes Kullback-Leibler divergence of y from x . The points themselves were sampled from a 10-dimensional Dirichlet distribution. Results are shown in Table 3 and discussed in the caption. The practical implications are:

- (1) A single call to **proposeNewFriendSet** costs $O(K^2 \log K)$ work on average⁵. Each round needs n calls to **proposeNewFriendSet**.
- (2) Run time for a single round of NND scales linearly with n , for fixed K .
- (3) The number of rounds of NND is not observed to exceed $2\lceil \log_K n \rceil$, suggesting an overall $O((n \log n)K^2)$ run time, on cancelling two $\log K$ factors.
- (4) For points on a 9-dimensional simplex, the accuracy (or recall) is 95% or better using our chosen stopping criterion, where accuracy means proportion of the true K nearest neighbors found, among a uniform sample of six⁶ points.
- (5) On p processors, parallel streams yield a speedup slightly less than p , presumably because of thread contention. We observed at best an 8 times speedup on 12 cores.

4.5. Effect of dimension. The experiments in Table 3 were performed on points from a 9-dimensional simplex, taking $K = 16, 32, 64$. We also performed experiments where the points were drawn from $(d-1)$ -dimensional simplices, for $d = 10, 20, 40, 60$, comparing the cases $K = 16$ and $K = 64$. Table 4 shows a decline both in the friend clustering coefficient, and in the accuracy of the K -NN approximation as dimension increases, for fixed K . Similar results are reported by Dong et al [7, Section 4.5], who interpret them as a consequence of the fact that, when sampling many points at random in high dimensions, the nearest neighbor and farthest neighbor of any point are at roughly the same distance [6].

5. CONCLUSIONS AND FUTURE WORK

In the benign setting of probability measures sampled uniformly at random from a simplex in Euclidean space, with a comparator based on Kullback-Leibler divergence, performance of K -nearest neighbor descent conforms to the predictions based loosely on expander graphs.

⁵On average $K + 2K^2$ items or fewer are proposed for insertion into a sorted set of size K .

⁶We did not choose a larger sample size than six, because the random initialization already causes random variation in the outcome when NND is applied repeatedly to the same data set.

In particular, our statistical stopping criterion is satisfied within $2\lceil\log_K n\rceil$ rounds on a set of n points, giving a run time proportional to

$$(1) \quad K^2 n \log n$$

in contrast to the $O(n^{1.14})$ run time (for fixed K) reported by Dong et al [7]. The accuracy of NND in (intrinsic) dimension up to 20 is entirely satisfactory in examples studied, and does not require that the similarity measure be symmetric or derived from a metric.

Performance of NND can be much worse in other settings, such as a collection of long multi-character strings under a metric based on the longest common substring [3, Section 4.4]. More theory and new experiments will be needed to delineate the contexts in which K -nearest neighbor descent works well.

Acknowledgment: The authors thank James Maissen for his comments and suggestions on the manuscript.

REFERENCES

- [1] M. Aumüller, E. Bernhardsson, A. Faithfull. ANN-Benchmarks: a benchmarking tool for approximate nearest neighbor algorithms. In: *Similarity Search and Applications 2017*. Lecture Notes in Computer Science, vol 10609. Springer, 2017
- [2] J. Bang-Jensen; G. Gutin. *Digraphs: Theory, Algorithms, and Applications*. Springer-Verlag London, 2009.
- [3] Jacob D. Baron; R. W. R. Darling. K -nearest neighbor approximation via the friend-of-a-friend principle. arXiv:1908.07645, 2019
- [4] Jacob D. Baron; R. W. R. Darling; J. Layton Davis; R. Pettit. Partitioned K -nearest neighbor local depth for scalable comparison-based learning. arXiv:2108.08864 [cs.DS], 2021
- [5] Erik Bernhardsson. Benchmarks of approximate nearest neighbor libraries in Python. github.com/erikbern/ann-benchmarks [Accessed December 27, 2021].
- [6] K. Beyer, J. Goldstein, R. Ramakrishnan, U. Shaft. When Is “Nearest Neighbor” Meaningful?. *ICDT 1999*. Lecture Notes in Computer Science, vol 1540. Springer, Berlin, Heidelberg. 1999
- [7] Wei Dong, Moses Charikar, Kai Li. Efficient k -nearest neighbor graph construction for generic similarity measures. *Proceedings of the 20th International Conference on World Wide Web*, 577–586, 2011
- [8] Alan Frieze & Michal Karonski. *Introduction to Random Graphs*. Cambridge U.P, 2016.
- [9] Siavash Haghighi, Debarghya Ghoshdastidar, Ulrike von Luxburg. Comparison based nearest neighbor search. arXiv: 1704.01460, 2017
- [10] Java™Platform, Standard Edition 17 API Specification, (2021). `Comparator< T >`, `Stream< T >`. <https://docs.oracle.com/en/java/javase/17/docs/api> [Accessed December 27, 2021].
- [11] Jeff Johnson, Matthijs Douze, Hervé Jégou. Billion-scale similarity search with GPUs. arXiv:1702.08734, 2017.
- [12] Dan Kluser, Jonas Bokstaller, Samuel Rutz, Tobias Buner. Fast single-core k -nearest neighbor graph computation. arXiv:2112.06630 [cs.LG], 2021
- [13] NeurIPS’21. Billion-scale approximate nearest neighbor search challenge. big-ann-benchmarks.com, 2021 [Accessed December 27, 2021].
- [14] Leland McInnes. `pynndescent`: A Python nearest neighbor descent for approximate nearest neighbors. github.com/lmcinnes/pynndescent, 2018 [Accessed December 27, 2021].
- [15] Leland McInnes; John Healy; James Melville. UMAP: uniform manifold approximation and projection for dimension reduction. arXiv:1802.03426, 2018