# Scalable $k$-Means Clustering for Large $k$ via Seeded Approximate Nearest-Neighbor Search

**Jack Spalding-Jamieson** [1] **Eliot Wong Robson** [*2] **Da Wei Zheng** [*2]

## Abstract

For very large values of $k$, we consider methods for fast $k$-means clustering of massive datasets with $10^7 \sim 10^9$ points in high-dimensions ($d \geq 100$). All current practical methods for this problem have runtimes at least $\Omega(k^2)$. We find that initialization routines are not a bottleneck for this case. Instead, it is critical to improve the speed of Lloyd's local-search algorithm, particularly the step that reassigns points to their closest center. Attempting to improve this step naturally leads us to leverage approximate nearest-neighbor search methods, although this alone is not enough to be practical. Instead, we propose a family of problems we call **Seeded Approximate Nearest-Neighbor Search**, for which we propose **Seeded Search-Graph** methods as a solution.

## 1. Introduction

$k$-means clustering is a classical problem in unsupervised learning and computational geometry, with numerous applications in machine learning and data mining. It has been thoroughly studied over the years, and it holds significant practical importance. See the recent survey of Ikotun et al. (2023) for a detailed discussion of this problem, its variants, and applications.

The problem considers as input a finite set of points $P \subset \mathbb{R}^d$ in $d$-dimensional space, and a parameter $k$. The goal is to choose a set of $k$ centers $C \subset \mathbb{R}^d$, $|C| = k$, minimizing the function $\sum_{p \in P} \min_{c \in C} \|p - c\|^2$. In other words, we wish to find $k$ centers $C$ such that the sum of the squared distances from each input point $p \in P$ to its closest center is minimized. Note that that the centers we choose may not be points of the original point set $P$. There are many alternative versions of this problem that have also been extensively studied (An & Svensson, 2017).

$k$-means clustering has long been known to be NP-hard, even for $k = 2$ (Aloise et al., 2009), but a number of polynomial time approaches are known to obtain good solutions, both in theory and in practice. We will discuss several of them in Appendix A.1. The most significant is a local search algorithm commonly attributed to Lloyd (1982) which is extremely popular in practice. The basic version of the algorithm is as follows:

1. **Initialize** A set of $k$ centers $C$ by uniform sampling from $P$.
2. **Assign** each point $P$ to its closest center. If no points change their assignment, the algorithm terminates.
3. **Recompute** each center $C_i$ by taking the mean of points assigned to it. Return to step 2.

We will henceforth refer to this method as **Lloyd's algorithm**. The second and third steps constitute a local search on the problem, and are referred to as **Lloyd iterations**.

Importantly, note that a single iteration of step 2 always computes $\Theta(|P| \cdot k)$ pairwise distances, regardless of the dataset. For use-cases of $k$-means clustering that require very large values of $k$, the runtime of this algorithm has a rather impractical dependence on $k$. To our knowledge, almost every other known competitive approach to $k$-means clustering over large high-dimensional datasets, both in practice and in theory, requires at least $\Omega(k^2)$ time overall. In particular, horizontal scaling and specialized hardware (such as GPUs) have been the only successful approaches used to mitigate this dependence.

### 1.1. Our Contribution

In this work, we study methods to better-mitigate the dependence on $k$ in standard $k$-means that do not require specialized hardware or multiple machines. We focus on the particularly challenging case of large datasets ($|P| \geq 10^7$)and moderate-to-high dimensionality ($d \geq 100$). As specific

[*]Equal contribution [1]Independent [2]Department of Computer Science; University of Illinois; 201 N. Goodwin Avenue; Urbana, IL, 61801, USA. Correspondence to: Jack Spalding-Jamieson <jacksj@uwaterloo.ca>, Eliot Wong Robson <erobson2@illinois.edu>, Da Wei Zheng <dwzheng2@illinois.edu>.

*Proceedings of the 1st Workshop on Vector Databases at International Conference on Machine Learning*, 2025. 

motivation, we will also briefly discuss one possible application of this particular case in Section 2.1.1. Additionally, in a first experiment, we will show that the most promising path towards designing a solution for this high-dimensional large-$k$ case is to study improvements to Lloyd's algorithm.

For this challenging case, we will present one modified form of Lloyd's method that is quite practical, even when $k$ is almost on the same order of magnitude as $|P|$. Moreover, our method requires no specialized hardware, requiring only a reasonably fast CPU to out-perform GPU implementations of the best known methods at this scale. At a high-level, our method will leverage techniques devised for (in-memory) **approximate nearest-neighbor search** (ANNS).

However, as we will see, the most direct method for applying ANNS techniques does not result in practical algorithms. Instead, we propose a more appropriate family of problems to study, which we call **seeded approximate nearest-neighbor search** (SANNS) where we have initial guesses (called seeds) for candidate nearest neighbors. SANNS can be seen as a learning-augmented form of ANNS. We present a framework of solutions to SANNS that we call **seeded search-graphs**. In particular, we present one particularly practical solution to SANNS using this framework. After tailoring our practical seeded search-graph approach to $k$-means, we show that our solution is highly effective for scalable high-dimensional $k$-means clustering with large $k$. We call our full solution **SHEESH** (Seeded searcH-grapHs for $k$-mEans cluStEring). [1]

### 1.2. Outline

In Section 2 and Appendix A, we discuss existing works on $k$-means clustering, approximate nearest-neighbor search, and related works. We will run several experiments in our paper, so we present shared details of our experimental setup in Appendix B. In Section 3, we experiment with initialization methods for large $k$ (Section 3.1) and a straightforward approach to accelerating Lloyd's algorithm (Section 3.2), and conclude that these methods are not sufficient to surpass simple forms of hardware-acceleration in terms of practicality. In Section 4, we first present the seeded approximate nearest-neighbor search problem (SANNS), as well as a semi-offline variant, and present seeded search-graphs as a method for solving SANNS. In addition, we discuss a highly practical seeded search-graph method specialized for $k$-means clustering, which consistently beats the hardware-accelerated implementations of Lloyd's algorithm. In Appendix E, we discuss some implementation details of this practical approach. In Appendix C, we present the full set of results. In Appendix D, we discuss another seeded search-graph method with some theoretical bounds. Lastly, in Section 5, we discuss some avenues for future work.

[1] Our code is available at https://github.com/jacketsj/mopbucket. It includes instructions for dataset retrieval and experimental reproduction.

## 2. Background

In this section, we discus some background on approximate nearest-neighbor search. We have deferred discussion of existing works on $k$-means clustering, as well as existing works applying approximate nearest-neighbor search methods to clustering, to Appendix A.

### 2.1. Background on ANNS Methods

In the **approximate nearest-neighbor search** (ANNS) problem, the goal is to design a data structure that takes as input a set $P$ of points and a pairwise distance/similarity function on the points, and efficiently outputs the $k'$ (approximate) nearest-neighbors to a query point $q$ in $P$ (we use $k'$ to differentiate from the $k$ in $k$-means). For a set of $d$ dimensional points $P \subset \mathbb{R}^d$, it is standard to use one of Euclidean, cosine, or inner-product functions as the distance/similarity function. All of these are essentially equivalent for high-dimensional ANNS (Bachrach et al., 2014). This problem is also often called **vector search** or **vector similarity search**. There are strong lower bounds for exact nearest-neighbor search data structures (Borodin et al., 1999), as well as lower bounds in approximate settings (Liu, 2004).

For our purposes, practical applications of approximate nearest-neighbor search can be divided into two groups:

- Those permitting **in-memory** techniques (i.e., the entire dataset $P$ can be stored in RAM). We will refer to this as in-memory ANNS.
- Those requiring **out-of-core** techniques (i.e., the dataset is too large to store in RAM, and is instead stored on disk or over a network). We will refer to this as out-of-core ANNS.

The distinguishing difference between these two groups is often the size of the data sets considered. Techniques for in-memory ANNS are usually only applied to million-scale datasets, while techniques for out-of-core ANNS are frequently applied to billion-scale datasets. There has been significant divergence between in-memory and out-of-core techniques.

For simplicity, we can categorize some of the most relevant techniques as follows:

- **Quantization methods**, such as product quantization (Matsui et al., 2018) and vector quantization (Liu et al., 2024), are (in a simplified sense) similar to dimension-reduction. Such methods are usually used in tandem with another technique, either as a method

to reduce memory usage, to reduce runtime, or both. Although they are an important tool for ANNS in other cases, we will not need to discuss them in detail for the purposes of this work.

- **Space-partitioning and clustering-based methods** constitute a very broad category of methods for ANNS. Theoretically-studied methods in this category include locality-sensitive hashing (Jafari et al., 2021) and RP-trees (Dasgupta & Freund, 2008). Several popular practical heuristic approaches include IVF (Sivic & Zisserman, 2003), IVFADC (Jegou et al., 2010; Jégou et al., 2011), SPANN Chen et al. (2021), and ScaNN/SOAR (Guo et al., 2020; Sun et al., 2024). In particular, all of these popular heuristic approaches apply some form of $k$-means clustering. In practice, such methods usually answer queries in two parts: First, a number of candidate clusters/partitions are identified (sometimes recursively). Next, they are searched, usually via another technique.
- **Graph-search methods**, such as HNSW (Malkov & Yashunin, 2018), NSG (Fu et al., 2019), and NSSG (Fu et al., 2022). Wang et al. (2021) give a survey of many such techniques. These methods answer queries using a **beam search** on a (sparse) directed graph defined over the dataset (described formally in Algorithm 1). In particular, many of the most popular methods for defining such a graph involve variations of a nearest-neighbor graph.

Almost all successful modern techniques for both in-memory and out-of-core ANNS leverage some sort of quantization, although they appear to be more critical for out-of-core ANNS (where they serve the purpose of memory-usage reduction, in addition to speed). However, the other two categories are more clearly separated. As a general rule, space-partitioning and clustering-based methods are used for out-of-core ANNS, while graph-search methods are used for in-memory ANNS. One reason for this rule is that graph-search methods are extremely efficient in terms of number of operations, but exhibit very poor locality, which is important in out-of-core contexts.

*We will only apply in-memory ANNS algorithms in this work.* However, out-of-core ANNS is still important to discuss for another reason: It serves as some direct motivation for accelerating $k$-means clustering, since many out-of-core ANNS algorithms rely on $k$-means clustering for extremely large datasets.

#### 2.1.1. Out-of-core ANNS

For out-of-core ANNS, space partitioning or clustering-based methods are practically essential, since they are the most effective tool for reducing memory usage. Some of the most recent successful methods for out-of-core ANNS also use a hybrid approach that additionally incorporates a graph (see Jayaram Subramanya et al. (2019), as well as some of the 2024 submissions to Big ANN Benchmarks (Simhadri et al., 2024)).

**Out-of-Core ANNS Uses $k$-Means** As stated before, we will not be running any out-of-core ANNS algorithms in this work, but they do motivate improvements to $k$-means clustering with large $k$. In particular, variants of $k$-means clustering are used in the vast majority of popular methods for space partitioning and clustering-based approaches. One reason for this is that $k$-means clustering over a dataset actually obtains *two* things: A clustering of the dataset itself, *and* a straightforward method for assigning new points (i.e, query points) to clusters. However, the most straightforward application of $k$-means would require many clusters, and the query-time assignment routine would also require $k$ distance-comparisons per query, *in addition* to performing a search within the chosen cluster(s). One way to mitigate this issue is to choose a value of $k$ balancing the average cluster size and the total cluster count (i.e., $k^2 \approx |P|$). Existing work has either had to make this balancing tradeoff (Jegou et al., 2010; Jégou et al., 2011; Bachrach et al., 2014; Baranchuk et al., 2018; Johnson et al., 2021), or apply workarounds like hierarchical clustering (e.g., $k$-means trees) (Guo et al., 2020; Chen et al., 2021; Sun et al., 2024). This presents us with a clear motivation for improving $k$-means clustering:

*Mitigate the dependence on $k$ in methods for $k$-means clustering.*

If we could do so in a way that would also allow for efficient assignment of query points, this would pave the way for out-of-core algorithms that do not have to apply such workarounds or tradeoffs. In particular, the methods we will present in Section 3.2 and Section 4 will do exactly this, by using variants of *in-memory ANNS* on the cluster centers. In this sense, one could start with an out-of-core ANNS instance (i.e., a massive dataset), and leverage our techniques to reduce to a special variant of in-memory ANNS (i.e., a much smaller dataset).

#### 2.1.2. In-memory ANNS

In contrast to out-of-core ANNS, *almost* all of the competitive techniques for in-memory ANNS primarily use graph-search methods. To the best of our knowledge, the only notable exception to the dominance of graph-based techniques for in-memory ANNS is ScaNN/SOAR (Guo et al., 2020; Sun et al., 2024), which (at a high-level) uses a $k$-means tree and some clever quantization. Douze et al. (2024) note that the reference implementation for both works is thoroughly optimized (and moreover, that the engineering optimizations

**Algorithm 1** Beam Search

**Input:** $P \subset \mathbb{R}^d$, search-graph $G = (P, E)$, $p^* \in P$, $q \in \mathbb{R}^d$, $p^* \in P$, $b \in \mathbb{Z}_{\geq 1}$
Initialize sets $C, N = \{p^*\}$ (candidates, nearest).
Mark $p^*$ as visited.
**repeat**
  Extract the element $c$ from $C$ nearest to $q$.
  **if** $|N| = b$ and $d(c, q) > d(n, q)$ for all $n \in N$ **then**
    **break**
  **end if**
  **for** each (outgoing) neighbor $v$ of $c$ in $G$ **do**
    **if** $v$ is not marked as visited **then**
      Mark $v$ as visited
      **if** $|N| < b$ or $d(v, q) < d(n, q)$ for some $n \in N$ **then**
        Add $v$ to $C$ and $N$
        If $|N| > b$, remove the furthest element in $N$.
        If $|C| > b$, remove the furthest element in $C$.
      **end if**
    **end if**
    Mark $v$ as visited.
  **end for**
**until** $C$ is empty
**Output:** $N$, the $b$ points in $P$ close to $q$

are not discussed in the paper) so it is possible this performance is more a result of careful engineering rather than characteristic to the algorithm. The repository by Aumüller et al. (2020) maintains an up-to-date benchmark of various in-memory ANNS implementations. For an evaluation of graph-based algorithms for approximate nearest-neighbor search, and discussions of parallelization techniques, see ParlayANN (Manohar et al., 2024).

**HNSW** One important in-memory method we will highlight now is **Hierarchical Navigable Small Worlds** (HNSW) (Malkov & Yashunin, 2018), a graph-search method for in-memory ANNS that has seen considerable industry adoption among vector search databases and libraries (e.g. Qdrant (2024), Milvus (2024), Weaviate (2024), USearch (Vardanian, 2023), and many more). For a detailed discussion on vector similarity search databases, see the recent survey by Pan et al. (2024).

HNSW is an **incremental** graph-search method, meaning it allows for both queries and insertions. At a high-level, most incremental graph-search methods (including HNSW) maintain a sparse subgraph of an approximate $k'$-nearest-neighbor graph over the dataset $P$ for some value of $k'$. Queries are performed with this structure using beam search over the graph. Beam search (see Algorithm 1) is sometimes called greedy search or best-first search in this context.

For HNSW in particular, these searches have an initial starting point which is an approximate nearest-neighbor from a random subsample of the data. This subsample can be performed successively, and a sparse graph can be maintained at each level (that is, a search-graph is maintained for each level, whose initial search points are determined within the level above), allowing for the starting point of a search to be determined recursively.

The routine for insertions is actually quite similar: To insert a point $p$ into an HNSW data structure containing a set of points $P$, the first step is to find the (approximate) $k'$-nearest-neighbors $S$ of $p$ in $P$. Then, some local updates are made to the graph to incorporate $p$, in a special (relatively fast) routine. For our purposes, we need not discuss the details of this routine, and we point curious readers to the original paper (Malkov & Yashunin, 2018). It should be noted that Manohar et al. (2024) made the observation that a careful sequence of bulk insertions are often more efficient when using parallelism.

HNSW has three key parameters, which are typically tuned based on the dataset and desired results:

- `ef_build` is the value $k$ to use for queries at insertion-time (a build-time parameter).
- `M` controls the sparsity of the final graph — it is the maximum number of outgoing edges at each vertex (a build-time parameter).
- `ef_search` is the value $k$ to use for queries at query-time (a query-time parameter).

Larger values of `ef_build` require longer build times, but usually offer better query time/accuracy tradeoffs.`M` is intended to be representative of the "intrinsic dimensionality" of the dataset, in a sense that is commonly applied to manifold learning techniques (see e.g., (Belkin & Niyogi, 2001)). Finally `ef_search` allows for the tuning of the tradeoff between query time and accuracy. Malkov & Yashunin (2018) present several other parameters and suggest methods for choosing them based on these three.

## 3. Initialization and Black-Box Reassignment

In this section, we first run some experiments for alternative initialization methods with a fairly large value of $k$. We conclude that the typical initialization methods applied to small values of $k$ are not very effective for large values of $k$, and that the important step to obtaining good solutions is Lloyd's algorithm. Afterwards, we will present a naive formulation of Lloyd's algorithm using a black-box ANNS data structure. We will then test this formulation over a suite of ANNS data structures.

To aid in the reading of our plots, in each legend, entries are sorted by their best score before the timeout. This is true of *every plot in the paper*.

### 3.1. Initialization Techniques for Large $k$

Classic formulations of Lloyd's algorithm typically use uniform sampling to choose the initial centroids (see Appendix A.1 for further discussion). A key observation of typical applications of $k$-means (i.e., when $k$ is small) is that the initialization method can be quite important (Arthur & Vassilvitskii, 2006; Bahmani et al., 2012). However, our testing indicates that the same does not hold for larger values of $k$. Even for the relatively small value of $k = 10\,000$, we observe that all initialization methods appear to converge to a near-identically-scored solution quite quickly on all of our tested datasets: See Figure 1, where we plotted the score over Lloyd iterations after various initialization methods. We used the cuML (Raschka et al., 2020) implementations of `k-means++` and `k-means||`, as well as the SciKit-Learn (Pedregosa et al., 2011) `k-means++` implementation in the case of one dataset due to VRAM constraints. Note that there are some subtleties with some implementations of these methods (Grunau et al., 2023).

### 3.2. Black-Box ANNS for Reassignment

We suggest a natural modification of Lloyd iterations using in-memory approximate nearest-neighbor search:

**Build**: Compute an ANNS data structure over the centers.

**Reassign**: Use the data structure to compute the approximate nearest center for each point in the dataset, and assign the point to the corresponding cluster.

**Recompute**: Recompute the centers as the centroids for the contents of each cluster.

The recompute step remains unchanged from Lloyd's algorithm. In the low-dimensional setting, an exact nearest-neighbor search data structure (such as a k-d tree) can be used in this framework to give an *exact* speedup to Lloyd's algorithm. For the high-dimensional setting we consider, such methods are inefficient, and we instead apply ANNS data structures.

We ran experiments on a large suite of popular in-memory ANNS algorithms, mostly by leveraging the implementations in the FAISS library (Douze et al., 2024). For baselines, we used three different implementations of Lloyd's algorithm: The (CPU) SciKit-Learn implementation (Pedregosa et al., 2011), a simple (GPU) implementation of our own using PyTorch (Paszke et al., 2019), and the (GPU) cuML implementation (Raschka et al., 2020). We give a detailed overview of our experimental setup in Appendix B, and a more detailed overview of the suite in Appendix C, alongside more detailed plots.

Some of the algorithms we tested performed quite well (e.g., hnsw [b] in Figure 2). Additionally it turns out that one very simple change to the reassignment step can also lead to marginally better results for all such methods: When performing a reassignment of a point, ensure that the new center is closer than the previously assigned center. After performing this change, we did obtain some slightly improved results (e.g., hnsw [a] vs hnsw [b] in Figure 2). but our previous observations remain true.

For a full list of algorithms we tested, see Appendix C. The main observations are as follows:

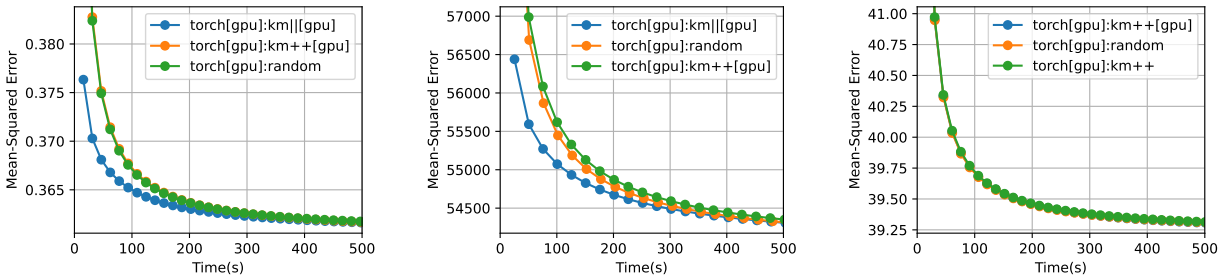
- For sufficiently large values of $k$, most methods based on clustering and/or quantization generally exhibited comparable performance to SciKit-Learn's implementation (Pedregosa et al., 2011) of Lloyd's algorithm. Generally these methods performed iterations much more rapidly than Lloyd's algorithm (as expected), but improved in score at a much slower rate over time.
- In contrast, the search-graph methods we tested were generally quite effective. Moreover, they appeared to obtain even better comparative performance compared to Lloyd's algorithm as $k$ increased. In a handful of cases, with large values of $k$, HNSW in particular was marginally better than the *GPU* implementations of Lloyd's algorithm (specifically, the cuML implementation (Raschka et al., 2020), and a basic implementation of ours leveraging PyTorch (Paszke et al., 2019)).

While these results are extremely promising, this black-box approach is (in most cases) not more practical than simply running Lloyd's algorithm on a GPU, even if it is comparable in some cases. We aim to arrive at a more practical method, so we will discuss our more carefully-designed methods in the next section.

## 4. Bulk Seeded Approximate Nearest-Neighbor Search via Search-Graphs

Although approximate nearest-neighbor search seems to fit in nicely into the framework of Lloyd's algorithm, naively using it does not result in good practical performance. We believe that this is because it does not take advantage of all available information.

Instead, we argue that the correct problem to solve is one we will call **Seeded Approximate Nearest-Neighbor Search** (SANNS), as well as a semi-offline variation of the problem we will call **Bulk Seeded Approximate Nearest-Neighbor Search** (BSANNS). Similarly to ANNS, in SANNS, the goal is to design a data structure that takes as input a set $P$




(a) Text2Image10M (b) SIFT20M (c) DPR5M

Figure 1: Comparisons of different initialization methods for $k$-means with $k = 10\,000$.

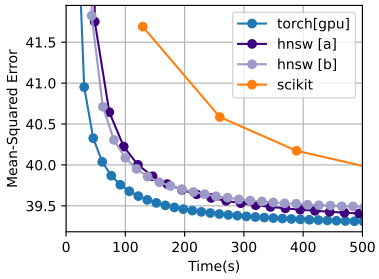



Figure 2: Comparison of HNSW as a black-box method for $k$-means clustering vs Lloyd's algorithm on the DPR5M dataset, with $k = 10\,000$. Initialization is uniformly random.

of points (say $P \subset \mathbb{R}^d$ for simplicity), and a pairwise distance function (say Euclidean distance for simplicity). Then, the data structure must answer queries consisting of a query point $q$ as well as identifiers for some small set of points in $P$. This small set of points in $P$ is called the set of **seed points**. This is intended to be a *learning-augmented* form of ANNS: It is *not* guaranteed that the seed points are good approximate nearest-neighbors of $q$. That is, algorithms for this problem should work regardless of whether or not the seed points provide useful information (in the language of learning-augmented algorithms, they should be **robust**) However, such algorithms should provide better results (in the tradeoff between time and result accuracy) if the seed points happen to be decent approximate neighbors of $q$ (in the language of learning-augmented algorithms, they should be **consistent**). See (Mitzenmacher & Vassilvitskii, 2021) for an overview of learning-augmented algorithms. In the batched version of this problem (BSANNS), the only difference is that the queries are given in large batches (say, of size $|P|$). The batched version of this problem is related to the so-called "approximate all-$k'$-nearest-neighbor search" problem, which is an offline version of ANNS (see e.g. (Ma & Li, 2019) for an algorithm in the low-dimensional exact version of this problem). In the context of $k$-means clustering, the "semi-offline" variation with batches of our dataset is more useful than a fully offline version because it allows us to minimize RAM usage.

Additionally, in Appendix D, we observe that a previously-studied search-graph algorithm with theoretical guarantees for ANNS naturally extends to SANNS as well, with guarantees for robustness and consistency (the expected types of guarantees for learning-augmented algorithms), based on analysis by Indyk & Xu (2023). Unfortunately, this algorithm is impractical for our use-case of $k$-means clustering with large $k$, since the build routine would take $\Omega(k^3)$ time in this context.

### 4.1. Seeded Search-Graphs

Recall that search-graph methods use the greedy/beam search routine in Algorithm 1. In particular, they use a prescribed initial point, whose choice depends on the particular search-graph method. We can additionally modify this routine to use *multiple* initial points, so long as the additional points are not too numerous. This leads us to a candidate method for SANNS: *In Algorithm 1, use the provided seed points as additional initial points in the greedy search routine.* We note that, for HNSW in particular, these additional initial points should only used during the search at the bottom-most layer, since such points may not even exist at higher layers.

**Bulk queries for additional seed points** To approach the BSANNS problem while leveraging our techniques for SANNS, we propose a heuristic: Specifically, we introduce *additional* seed points for bulk queries. By grouping together correlated queries from a bulk query, we can then perform queries for each group in a careful fashion. Specifically, while iterating through each group, we obtain the top results for each query, and then use them as additional seeds for the next query. We also propose a simple method for choosing an iteration order for the datapoints within each group: Randomly project all datapoints into a 1-dimensional

space, and sort them.

The idea here is simple: Correlated query groups can be considered a very "rough" clustering of the data (not necessarily a $k$-means clustering), and so they are more likely to be assigned to the same final centroid. Moreover, within each group, the ordering is expected to project each group of $n$ points into $\mathbb{R}$ with $O(\log n)$ distortion, as shown by Johnson & Lindenstrauss (1984). With HNSW, the correlation technique we used was simply to group everything by its default "initial point" given by the recursive structure of HNSW. One could also consider grouping everything by its best seed point, but we would only expect this to be effective in cases where the size of a bulk query is much larger than the number of datapoints (so that the groups are of nontrivial size).

### 4.2. Seeded Search-Graphs for $k$-Means

We now discuss how to apply seeded search-graphs (and more broadly, algorithms for SANNS/BSANNS) to Lloyd's algorithm, and some further specialized improvements. In particular, we will implement our method using HNSW as a basis, with several layered improvements.

**Using Seed Points** The first method we employed was to to use the previous iteration's assignments as seed points for seeded search-graphs. There is reason to believe this is a good heuristic: The centers slow their movements over the course of many Lloyd iterations (see Table 1), so the best approximate assignment is increasingly likely to be the previous assignment as the number of iterations increases.

| It. | Avg. dist. | It. | Avg. dist. | It. | Avg. dist. |
|---|---|---|---|---|---|
| 1 | 136.012 | 14 | 4.070 | 27 | 1.889 |
| 2 | 31.827 | 15 | 3.780 | 28 | 1.842 |
| 3 | 19.410 | 16 | 3.529 | 29 | 1.766 |
| 4 | 14.190 | 17 | 3.302 | 30 | 1.682 |
| 5 | 11.243 | 18 | 3.096 | 31 | 1.591 |
| 6 | 9.357 | 19 | 2.899 | 32 | 1.519 |
| 7 | 8.044 | 20 | 2.766 | 33 | 1.456 |
| 8 | 7.082 | 21 | 2.587 | 34 | 1.375 |
| 9 | 6.282 | 22 | 2.432 | 35 | 1.295 |
| 10 | 5.701 | 23 | 2.290 | 36 | 1.239 |
| 11 | 5.160 | 24 | 2.182 | 37 | 1.237 |
| 12 | 4.732 | 25 | 2.084 | 38 | 1.172 |
| 13 | 4.408 | 26 | 1.969 | | |

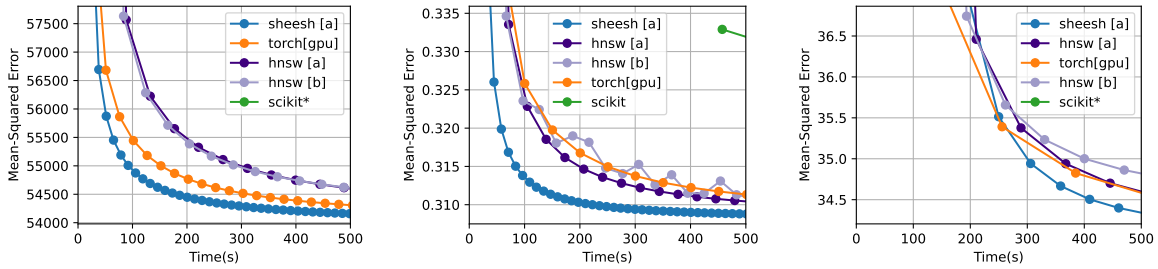
Table 1: The average distance centroids move during each standard Lloyd iteration while clustering SIFT1M ($10^6$ points) with $k = 5000$ clusters, demonstrating that the movement of the centroids slows over time. The average distance from a datapoint in SIFT1M to the respective closest of these centroids ranges from approximately $43\,481$ to $46\,524$ over 38 iterations, showing that these movements are small.

It turns out this often already provides a small improvement (this comparison is shown in Appendix C, alongside many others). However, we can obtain even better performance as follows: In each iteration, instead of recording just the best centroid assignment, we can actually record several of the top assignments. We use the top 10, although this parameter has not been tuned. Then, instead of using a single seed point during the next iteration, we can them all as seed points, by initializing the sets $C, N$ in Algorithm 1 with these seed points ($N$ obtaining only the nearest $b$ of the seed points). This leads to an even further improved algorithm (this comparison is also shown in Appendix C).

**Algorithm 2 SHEESH**: Accelerated Lloyd Iteration with BSANNS via Seeded Search-Graphs

**Input:** $P \subset \mathbb{R}^d$, centers $C \subset \mathbb{R}^d$, $|C| = k$, previous multi-assignments $S : P \to 2^C$, previous search-graph data structure $D$
**Build:** Build a new search-graph data structure $D'$ by using $D$.
**Reassign:**
**for** each chunk $U$ of $O(k)$ points in $P$ (in parallel) **do**
  Group the points of $U$ into roughly-correlated groups. Randomly project each group into $\mathbb{R}^1$, and sort the projected group.
  **for** each group $G$ of $U$ **do**
    **for** each point $p$ of $G$, in the sorted order **do**
      Let $q$ be the previous point.
      Use $S'(q) \cup S(p)$ as seeds.
      With all these seeds, compute the seeded approximate $\sim 10$ nearest centers of $p$ using $D'$, and save the results as $S'(p)$.
    **end for**
  **end for**
**end for**
**Recompute:** Compute the new centers $C'$ as centroids.
**Output:** New centers $C'$, new multi-assignments $S'$, new search-graph data structure $D'$

**Continuous Rebuilds** As noted by each of the works using the so-called "inverse assignment" method discussed in Appendix A.2, it is often inefficient to build a data structure over the centers from scratch at each iteration. The inverse assignment method is one way of handling this issue, although, as noted, it does not scale to large datasets. Instead, we suggest the following approach: On all iterations except the first, leverage the search-graph of the previous iteration to construct the new search-graph, rather than starting with an empty graph. Since most centers do not significantly change between later iterations on average (see Table 1), this graph serves as a good coarse approximation of the desired search-graph, and the quality of the approximation improves over time as the centers simultaneously converge.




(a) SIFT20M, $k = 10\,000$ (b) Text2Image10M, $k = 50\,000$ (c) DPR5M, $k = 100\,000$

Figure 3: Comparison of our approach with GPU acceleration, as well as the black-box HNSW approach on the SIFT20M, Text2Image10M, and DPR5M datasets respectively, for the listed values of $k$. Initialization is uniformly random.

For HNSW in particular, we note that we fix the subsampling of the centers at higher levels of the data structure — allowing the same simultaneous convergence to occur on the higher levels of the search-graph.

**Min Iteration Count** Intuitively, search-graphs seeded with result from a previous Lloyd iteration may get stuck more easily in the exact same local optimum across iterations, especially if the graph itself is less prone to change over time. To combat this, we suggest a heuristic: Rather than always terminating Algorithm 1 early once a local optimum is reached, we also require that a specified minimum number of iterations have been performed. In particular, for HNSW, this bound is only applied at the lowest level of the hierarchy (the search-graph over the full dataset), where the seeds are also applied.

The combination of these two additional improvements, along with the bulk methodology discussed Section 4.1, is an algorithm we call **SHEESH** (Seeded searcH-grapHs for $k$-mEans cluStEring). It is labeled sheesh [a] in Figure 3. Recall that, in all of our plots' legends, entries are in descending order by their best score. In addition, any plot entries marked with a "*" timed out by taking $> 500$ seconds to finish even a single Lloyd iteration. In particular, our algorithm with all features enabled (sheesh [a]), achieved the best score in all experiments. We have run numerous experiments in addition to the ones shown in Figure 3, which we discuss more thoroughly in Appendix C. Our full algorithm for $k$-means leveraging seeded search-graphs is outlined in Algorithm 2.

## 5. Conclusion

We have presented a methodology for accelerating $k$-means with large values of $k$. In particular, we leveraged methods previously applied to ANNS, and improved them for SANNS and the specific application of $k$-means clustering, culminating in Algorithm 2. We have demonstrated that our method is quite performant for $k$-means clustering some large high-dimensional image and text embedding datasets. We believe there are two main interesting directions of further research: Evaluating our methodology on various applications of $k$-means clustering, and Studying methods to further accelerate our methods. For the latter reason in particular, we believe the results in Section 3.2 are quite interesting, since they do not agree with the performance of the same techniques for standard in-memory ANNS. This may indicate that there are more interesting specialized methods that can be applied to SANNS in the context of $k$-means. One avenue for exploration could be to try adapting similar techniques to what is used for so-called "out-of-distribution" ANNS, such as Chen et al. (2024).

Another avenue for accelerating our methods could be to explore approaches based on specialized hardware. We focused on devising CPU-based algorithms, and our final methods rely on search-graphs. There is some work exploring GPU-based algorithms for ANNS on search-graphs (Zhao et al., 2020; Yu et al., 2022; Groh et al., 2023; Ootomo et al., 2024). Currently, the fastest of these is CAGRA (Ootomo et al., 2024). Unfortunately, CAGRA's preprocessing routine starts by applying a (hardware-accelerated) quadratic-time algorithm. Moreover, one of its key hardware-leveraging methods (step 0 of their described algorithm) seems unlikely to be helpful for SANNS. Consequently, we believe it is likely that CAGRA's hardware-acceleration methods would not obtain the same speedup over CPU, although this would need to be tested. Intuitively, searching over a graph (even a regular graph) is not a very efficient operation for a GPU, so there may be other more-effective families of methods for hardware acceleration.

## Acknowledgements

Research of the third author supported in part by an NSERC PGSD. This material is based upon preliminary work supported by the Google Cloud Research Credits for PhD students.

## References


Aguerrebere, C., Bhati, I., Hildebrand, M., Tepper, M., and Willke, T. Similarity search in the blink of an eye with compressed indices. *Proceedings of the VLDB Endowment*, 16(11):3433–3446, 2023.

Aloise, D., Deshpande, A., Hansen, P., and Popat, P. Np-hardness of euclidean sum-of-squares clustering. *Mach. Learn.*, 75(2):245–248, 2009. doi: 10.1007/S10994-009-5103-0. URL https://doi.org/10.1007/s10994-009-5103-0.

An, H.-C. and Svensson, O. Recent developments in approximation algorithms for facility location and clustering problems. In *Combinatorial Optimization and Graph Algorithms: Communications of NII Shonan Meetings*, pp. 1–19. Springer, 2017.

Arthur, D. and Vassilvitskii, S. Worst-case and smoothed analysis of the icp algorithm, with an application to the k-means method. In *2006 47th Annual IEEE Symposium on Foundations of Computer Science (FOCS'06)*, pp. 153–164, 2006. doi: 10.1109/FOCS.2006.79.

Aumüller, M., Bernhardsson, E., and Faithfull, A. Ann-benchmarks: A benchmarking tool for approximate nearest neighbor algorithms. *Information Systems*, 87:101374, 2020. URL https://github.com/erikbern/ann-benchmarks/.

Avrithis, Y., Kalantidis, Y., Anagnostopoulos, E., and Emiris, I. Z. Web-scale image clustering revisited. In *2015 IEEE International Conference on Computer Vision, ICCV 2015, Santiago, Chile, December 7-13, 2015*, pp. 1502–1510. IEEE Computer Society, 2015. doi: 10.1109/ICCV.2015.176. URL https://doi.org/10.1109/ICCV.2015.176.

Awasthi, P., Charikar, M., Krishnaswamy, R., and Sinop, A. K. The hardness of approximation of euclidean k-means. In Arge, L. and Pach, J. (eds.), *31st International Symposium on Computational Geometry, SoCG 2015, June 22-25, 2015, Eindhoven, The Netherlands*, volume 34 of *LIPIcs*, pp. 754–767. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2015. doi: 10.4230/LIPICS.SOCG.2015.754. URL https://doi.org/10.4230/LIPIcs.SOCG.2015.754.

Bachrach, Y., Finkelstein, Y., Gilad-Bachrach, R., Katzir, L., Koenigstein, N., Nice, N., and Paquet, U. Speeding up the xbox recommender system using a euclidean transformation for inner-product spaces. In *Proceedings of the 8th ACM Conference on Recommender systems*, pp. 257–264, 2014.

Bahmani, B., Moseley, B., Vattani, A., Kumar, R., and Vassilvitskii, S. Scalable k-means++. *Proc. VLDB Endow.*, 5(7):622–633, mar 2012. ISSN 2150-8097. doi: 10.14778/2180912.2180915. URL https://doi.org/10.14778/2180912.2180915.

Baranchuk, D., Babenko, A., and Malkov, Y. Revisiting the inverted indices for billion-scale approximate nearest neighbors. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pp. 202–216, 2018.

Belkin, M. and Niyogi, P. Laplacian eigenmaps and spectral techniques for embedding and clustering. In Dietterich, T. G., Becker, S., and Ghahramani, Z. (eds.), *Advances in Neural Information Processing Systems 14 [Neural Information Processing Systems: Natural and Synthetic, NIPS 2001, December 3-8, 2001, Vancouver, British Columbia, Canada]*, pp. 585–591. MIT Press, 2001. URL https://proceedings.neurips.cc/paper/2001/hash/f106b7f99d2cb30c3db1c3cc0fde9ccb-Abstract.html.

Beretta, L., Cohen-Addad, V., Lattanzi, S., and Parotsidis, N. Multi-swap k-means++. *Advances in Neural Information Processing Systems*, 36, 2024.

Borodin, A., Ostrovsky, R., and Rabani, Y. Lower bounds for high dimensional nearest neighbor search and related problems. In Vitter, J. S., Larmore, L. L., and Leighton, F. T. (eds.), *Proceedings of the Thirty-First Annual ACM Symposium on Theory of Computing, May 1-4, 1999, Atlanta, Georgia, USA*, pp. 312–321. ACM, 1999. doi: 10.1145/301250.301330. URL https://doi.org/10.1145/301250.301330.

Chen, M., Zhang, K., He, Z., Jing, Y., and Wang, X. S. Roargraph: A projected bipartite graph for efficient cross-modal approximate nearest neighbor search. *Proc. VLDB Endow.*, 17(11):2735–2749, 2024. doi: 10.14778/3681954.3681959. URL https://www.vldb.org/pvldb/vol17/p2735-chen.pdf.

Chen, Q., Zhao, B., Wang, H., Li, M., Liu, C., Li, Z., Yang, M., and Wang, J. Spann: Highly-efficient billion-scale approximate nearest neighborhood search. *Advances in Neural Information Processing Systems*, 34:5199–5212, 2021.

Chubet, O. A., Parikh, P., Sheehy, D. R., and Sheth, S. S. Proximity search in the greedy tree. In Kavitha, T. and Mehlhorn, K. (eds.), *2023 Symposium on Simplicity in Algorithms, SOSA 2023, Florence, Italy, January 23-25, 2023*, pp. 332–342. SIAM, 2023. doi: 10.1137/1.9781611977585.CH29. URL https://doi.org/10.1137/1.9781611977585.ch29.

Dasgupta, S. and Freund, Y. Random projection trees and low dimensional manifolds. In Dwork, C. (ed.), *Pro-*

*ceedings of the 40th Annual ACM Symposium on Theory of Computing, Victoria, British Columbia, Canada, May 17-20, 2008*, pp. 537–546. ACM, 2008. doi: 10.1145/1374376.1374452. URL https://doi.org/10.1145/1374376.1374452.

Dong, W., Charikar, M., and Li, K. Efficient k-nearest neighbor graph construction for generic similarity measures. In Srinivasan, S., Ramamritham, K., Kumar, A., Ravindra, M. P., Bertino, E., and Kumar, R. (eds.), *Proceedings of the 20th International Conference on World Wide Web, WWW 2011, Hyderabad, India, March 28 - April 1, 2011*, pp. 577–586. ACM, 2011. doi: 10.1145/1963405.1963487. URL https://doi.org/10.1145/1963405.1963487.

Douze, M., Guzhva, A., Deng, C., Johnson, J., Szilvasy, G., Mazaré, P., Lomeli, M., Hosseini, L., and Jégou, H. The faiss library. *CoRR*, abs/2401.08281, 2024. doi: 10.48550/ARXIV.2401.08281. URL https://doi.org/10.48550/arXiv.2401.08281.

Elkan, C. Using the triangle inequality to accelerate k-means. In Fawcett, T. and Mishra, N. (eds.), *Machine Learning, Proceedings of the Twentieth International Conference (ICML 2003), August 21-24, 2003, Washington, DC, USA*, pp. 147–153. AAAI Press, 2003. URL http://www.aaai.org/Library/ICML/2003/icml03-022.php.

Friggstad, Z., Rezapour, M., and Salavatipour, M. R. Local search yields a ptas for $k$-means in doubling metrics. *SIAM Journal on Computing*, 48(2):452–480, 2019. doi: 10.1137/17M1127181. URL https://doi.org/10.1137/17M1127181.

Fu, C., Xiang, C., Wang, C., and Cai, D. Fast approximate nearest neighbor search with the navigating spreading-out graph. *Proc. VLDB Endow.*, 12(5):461–474, 2019. doi: 10.14778/3303753.3303754. URL http://www.vldb.org/pvldb/vol12/p461-fu.pdf.

Fu, C., Wang, C., and Cai, D. High dimensional similarity search with satellite system graph: Efficiency, scalability, and unindexed query compatibility. *IEEE Trans. Pattern Anal. Mach. Intell.*, 44(8):4139–4150, 2022. doi: 10.1109/TPAMI.2021.3067706. URL https://doi.org/10.1109/TPAMI.2021.3067706.

Gong, Y., Pawlowski, M., Yang, F., Brandy, L., Bourdev, L. D., and Fergus, R. Web scale photo hash clustering on a single machine. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2015, Boston, MA, USA, June 7-12, 2015*, pp. 19–27. IEEE Computer Society, 2015. doi: 10.1109/CVPR.2015.7298596. URL https://doi.org/10.1109/CVPR.2015.7298596.

Gottlieb, L. and Krauthgamer, R. Proximity algorithms for nearly doubling spaces. *SIAM J. Discret. Math.*, 27 (4):1759–1769, 2013. doi: 10.1137/120874242. URL https://doi.org/10.1137/120874242.

Groh, F., Ruppert, L., Wieschollek, P., and Lensch, H. P. A. GGNN: graph-based GPU nearest neighbor search. *IEEE Trans. Big Data*, 9(1):267–279, 2023. doi: 10.1109/TBDATA.2022.3161156. URL https://doi.org/10.1109/TBDATA.2022.3161156.

Grunau, C., Özüdogru, A. A., Rozhon, V., and Tetek, J. A nearly tight analysis of greedy k-means++. In Bansal, N. and Nagarajan, V. (eds.), *Proceedings of the 2023 ACM-SIAM Symposium on Discrete Algorithms, SODA 2023, Florence, Italy, January 22-25, 2023*, pp. 1012–1070. SIAM, 2023. doi: 10.1137/1.9781611977554.CH39. URL https://doi.org/10.1137/1.9781611977554.ch39.

Guennebaud, G., Jacob, B., et al. Eigen v3. http://eigen.tuxfamily.org, 2010.

Guo, R., Sun, P., Lindgren, E., Geng, Q., Simcha, D., Chern, F., and Kumar, S. Accelerating large-scale inference with anisotropic vector quantization. In *International Conference on Machine Learning*, pp. 3887–3896. PMLR, 2020.

Hamerly, G. Making k-means even faster. In *Proceedings of the SIAM International Conference on Data Mining, SDM 2010, April 29 - May 1, 2010, Columbus, Ohio, USA*, pp. 130–140. SIAM, 2010. doi: 10.1137/1.9781611972801.12. URL https://doi.org/10.1137/1.9781611972801.12.

Hamerly, G. and Drake, J. *Accelerating Lloyd's Algorithm for k-Means Clustering*, pp. 41–78. Springer International Publishing, Cham, 2015. ISBN 978-3-319-09259-1. doi: 10.1007/978-3-319-09259-1_2. URL https://doi.org/10.1007/978-3-319-09259-1_2.

Har-Peled, S. and Mendel, M. Fast construction of nets in low-dimensional metrics and their applications. *SIAM J. Comput.*, 35(5):1148–1184, 2006. doi: 10.1137/S0097539704446281. URL https://doi.org/10.1137/S0097539704446281.

Harris, C. R., Millman, K. J., van der Walt, S. J., Gommers, R., Virtanen, P., Cournapeau, D., Wieser, E., Taylor, J., Berg, S., Smith, N. J., Kern, R., Picus, M., Hoyer, S., van Kerkwijk, M. H., Brett, M., Haldane, A., del Río, J. F., Wiebe, M., Peterson, P., Gérard-Marchant, P., Sheppard, K., Reddy, T., Weckesser, W., Abbasi, H., Gohlke, C., and Oliphant, T. E. Array programming with NumPy. *Nature*, 585(7825):357–362, September 2020. doi: 10.1038/s41586-020-2649-2. URL https://doi.org/10.1038/s41586-020-2649-2.

Hu, J., Shen, L., Albanie, S., Sun, G., and Wu, E. Squeeze-and-excitation networks. *IEEE Trans. Pattern Anal. Mach. Intell.*, 42(8):2011–2023, 2020. doi: 10.1109/TPAMI.2019.2913372. URL https://doi.org/10.1109/TPAMI.2019.2913372.

Hu, Q., Wu, J., Bai, L., Zhang, Y., and Cheng, J. Fast k-means for large scale clustering. In Lim, E., Winslett, M., Sanderson, M., Fu, A. W., Sun, J., Culpepper, J. S., Lo, E., Ho, J. C., Donato, D., Agrawal, R., Zheng, Y., Castillo, C., Sun, A., Tseng, V. S., and Li, C. (eds.), *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management, CIKM 2017, Singapore, November 06 - 10, 2017*, pp. 2099–2102. ACM, 2017. doi: 10.1145/3132847.3133091. URL https://doi.org/10.1145/3132847.3133091.

Ikotun, A. M., Ezugwu, A. E., Abualigah, L., Abuhaija, B., and Heming, J. K-means clustering algorithms: A comprehensive review, variants analysis, and advances in the era of big data. *Information Sciences*, 622:178–210, 2023. ISSN 0020-0255. doi: https://doi.org/10.1016/j.ins.2022.11.139. URL https://www.sciencedirect.com/science/article/pii/S0020025522014633.

Indyk, P. and Xu, H. Worst-case performance of popular approximate nearest neighbor search implementations: Guarantees and limitations. *Advances in Neural Information Processing Systems*, 36:66239–66256, 2023.

Jafari, O., Maurya, P., Nagarkar, P., Islam, K. M., and Crushev, C. A survey on locality sensitive hashing algorithms and their applications. *CoRR*, abs/2102.08942, 2021. URL https://arxiv.org/abs/2102.08942.

Jayaram Subramanya, S., Devvrit, F., Simhadri, H. V., Krishnawamy, R., and Kadekodi, R. Diskann: Fast accurate billion-point nearest neighbor search on a single node. *Advances in Neural Information Processing Systems*, 32, 2019.

Jegou, H., Douze, M., and Schmid, C. Product quantization for nearest neighbor search. *IEEE transactions on pattern analysis and machine intelligence*, 33(1):117–128, 2010.

Jégou, H., Tavenard, R., Douze, M., and Amsaleg, L. Searching in one billion vectors: re-rank with source coding. In *2011 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 861–864. IEEE, 2011.

Jégou, H., Tavenard, R., Douze, M., and Amsaleg, L. Searching in one billion vectors: Re-rank with source coding. In *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing, ICASSP 2011, May 22-27, 2011, Prague Congress Center, Prague, Czech Republic*, pp. 861–864. IEEE, 2011. doi: 10.1109/ICASSP.2011.5946540. URL https://doi.org/10.1109/ICASSP.2011.5946540.

Johnson, J., Douze, M., and Jégou, H. Billion-scale similarity search with gpus. *IEEE Trans. Big Data*, 7(3):535–547, 2021. doi: 10.1109/TBDATA.2019.2921572. URL https://doi.org/10.1109/TBDATA.2019.2921572.

Johnson, W. B. and Lindenstrauss, J. Extensions of lipschitz mappings into a hilbert space. In *Conference on Modern Analysis and Probability*, volume 26, pp. 189–206. American Mathematical Society, 1984.

Kanungo, T., Mount, D., Netanyahu, N., Piatko, C., Silverman, R., and Wu, A. An efficient k-means clustering algorithm: analysis and implementation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 24(7):881–892, 2002a. doi: 10.1109/TPAMI.2002.1017616.

Kanungo, T., Mount, D. M., Netanyahu, N. S., Piatko, C. D., Silverman, R., and Wu, A. Y. A local search approximation algorithm for k-means clustering. In *Proceedings of the Eighteenth Annual Symposium on Computational Geometry*, SCG '02, pp. 10–18, New York, NY, USA, 2002b. Association for Computing Machinery. ISBN 1581135041. doi: 10.1145/513400.513402. URL https://doi.org/10.1145/513400.513402.

Karpukhin, V., Oguz, B., Min, S., Lewis, P., Wu, L., Edunov, S., Chen, D., and Yih, W.-t. Dense passage retrieval for open-domain question answering. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 6769–6781, Online, November 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.emnlp-main.550. URL https://www.aclweb.org/anthology/2020.emnlp-main.550.

KDAB. Hotspot, August 2020. URL https://github.com/KDAB/hotspot.

Linux Kernel Organization. *perf: Linux profiling with performance counters*, 2024. URL https://perf.wiki.kernel.org/.

Liu, D. A strong lower bound for approximate nearest neighbor searching. *Inf. Process. Lett.*, 92(1):23–29, 2004. doi: 10.1016/J.IPL.2004.06.001. URL https://doi.org/10.1016/j.ipl.2004.06.001.

Liu, Q., Dong, X., Xiao, J., Chen, N., Hu, H., Zhu, J., Zhu, C., Sakai, T., and Wu, X.-M. Vector quantization for recommender systems: A review and outlook, 2024. URL https://arxiv.org/abs/2405.03110.

Lloyd, S. P. Least squares quantization in PCM. *IEEE Trans. Inform. Theory*, 28(2):129–137, 1982. ISSN 0018-9448. doi: 10.1109/TIT.1982.1056489. URL https://doi.org/10.1109/TIT.1982.1056489.

Lowe, D. G. Distinctive image features from scale-invariant keypoints. *Int. J. Comput. Vis.*, 60(2):91–110, 2004. doi: 10.1023/B:VISI.0000029664.99615.94. URL https://doi.org/10.1023/B:VISI.0000029664.99615.94.

Ma, H. and Li, J. A true o(nlog n) algorithm for the all-k-nearest-neighbors problem. In Li, Y., Cardei, M., and Huang, Y. (eds.), *Combinatorial Optimization and Applications - 13th International Conference, COCOA 2019, Xiamen, China, December 13-15, 2019, Proceedings*, volume 11949 of *Lecture Notes in Computer Science*, pp. 362–374. Springer, 2019. doi: 10.1007/978-3-030-36412-0_29. URL https://doi.org/10.1007/978-3-030-36412-0_29.

Malkov, Y. A. and Yashunin, D. A. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE transactions on pattern analysis and machine intelligence*, 42(4):824–836, 2018.

Manohar, M. D., Shen, Z., Blelloch, G., Dhulipala, L., Gu, Y., Simhadri, H. V., and Sun, Y. Parlayann: Scalable and deterministic parallel graph-based approximate nearest neighbor search algorithms. In *Proceedings of the 29th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, pp. 270–285, 2024.

Matsui, Y., Ogaki, K., Yamasaki, T., and Aizawa, K. Pqk-means: Billion-scale clustering for product-quantized codes. In Liu, Q., Lienhart, R., Wang, H., Chen, S. K., Boll, S., Chen, Y. P., Friedland, G., Li, J., and Yan, S. (eds.), *Proceedings of the 2017 ACM on Multimedia Conference, MM 2017, Mountain View, CA, USA, October 23-27, 2017*, pp. 1725–1733. ACM, 2017. doi: 10.1145/3123266.3123430. URL https://doi.org/10.1145/3123266.3123430.

Matsui, Y., Uchida, Y., Jégou, H., and Satoh, S. A survey of product quantization. *ITE Transactions on Media Technology and Applications*, 6(1):2–10, 2018.

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

Milvus. Milvus: Vector database, 2024. URL https://github.com/milvus-io/milvus.

Mitzenmacher, M. and Vassilvitskii, S. *Algorithms with Predictions*, pp. 646–662. Cambridge University Press, 2021.

Ootomo, H., Naruse, A., Nolet, C., Wang, R., Feher, T., and Wang, Y. CAGRA: highly parallel graph construction and approximate nearest neighbor search for gpus. In *40th IEEE International Conference on Data Engineering, ICDE 2024, Utrecht, The Netherlands, May 13-16, 2024*, pp. 4236–4247. IEEE, 2024. doi: 10.1109/ICDE60146.2024.00323. URL https://doi.org/10.1109/ICDE60146.2024.00323.

Pan, J. J., Wang, J., and Li, G. Survey of vector database management systems. *VLDB J.*, 33(5):1591–1615, 2024. doi: 10.1007/S00778-024-00864-X. URL https://doi.org/10.1007/s00778-024-00864-x.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Köpf, A., Yang, E. Z., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. Pytorch: An imperative style, high-performance deep learning library. In Wallach, H. M., Larochelle, H., Beygelzimer, A., d'Alché-Buc, F., Fox, E. B., and Garnett, R. (eds.), *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pp. 8024–8035, 2019. URL https://proceedings.neurips.cc/paper/2019/hash/bdbca288fee7f92f2bfa9f7012727740-Abstract.html.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., and Duchesnay, E. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

Philbin, J., Chum, O., Isard, M., Sivic, J., and Zisserman, A. Object retrieval with large vocabularies and fast spatial matching. In *2007 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR 2007), 18-23 June 2007, Minneapolis, Minnesota, USA*. IEEE Computer Society, 2007. doi: 10.1109/CVPR.2007.383172. URL https://doi.org/10.1109/CVPR.2007.383172.

Qdrant. Qdrant - vector database, 2024. URL https://github.com/qdrant/qdrant.

Raschka, S., Patterson, J., and Nolet, C. Machine learning in python: Main developments and technology trends in

data science, machine learning, and artificial intelligence. *arXiv preprint arXiv:2002.04803*, 2020.

Sculley, D. Web-scale k-means clustering. In Rappa, M., Jones, P., Freire, J., and Chakrabarti, S. (eds.), *Proceedings of the 19th International Conference on World Wide Web, WWW 2010, Raleigh, North Carolina, USA, April 26-30, 2010*, pp. 1177–1178. ACM, 2010. doi: 10.1145/1772690.1772862. URL https://doi.org/10.1145/1772690.1772862.

Simhadri, H. V., Williams, G., Aumüller, M., Douze, M., Babenko, A., Baranchuk, D., Chen, Q., Hosseini, L., Krishnaswamny, R., Srinivasa, G., Subramanya, S. J., and Wang, J. Results of the neurips'21 challenge on billion-scale approximate nearest neighbor search. In Kiela, D., Ciccone, M., and Caputo, B. (eds.), *Proceedings of the NeurIPS 2021 Competitions and Demonstrations Track*, volume 176 of *Proceedings of Machine Learning Research*, pp. 177–189. PMLR, 06–14 Dec 2022. URL https://proceedings.mlr.press/v176/simhadri22a.html.

Simhadri, H. V., Aumüller, M., Ingber, A., Douze, M., Williams, G., Manohar, M. D., Baranchuk, D., Liberty, E., Liu, F., Landrum, B., Karjikar, M., Dhulipala, L., Chen, M., Chen, Y., Ma, R., Zhang, K., Cai, Y., Shi, J., Chen, Y., Zheng, W., Wan, Z., Yin, J., and Huang, B. Results of the big ANN: neurips'23 competition. *CoRR*, abs/2409.17424, 2024. doi: 10.48550/ARXIV.2409.17424. URL https://doi.org/10.48550/arXiv.2409.17424.

Sivic, J. and Zisserman, A. Video google: A text retrieval approach to object matching in videos. In *9th IEEE International Conference on Computer Vision (ICCV 2003), 14-17 October 2003, Nice, France*, pp. 1470–1477. IEEE Computer Society, 2003. doi: 10.1109/ICCV.2003.1238663. URL https://doi.org/10.1109/ICCV.2003.1238663.

Sun, P., Simcha, D., Dopson, D., Guo, R., and Kumar, S. Soar: Improved indexing for approximate nearest neighbor search. *Advances in Neural Information Processing Systems*, 36, 2024.

Uieda, L., Soler, S. R., Rampin, R., van Kemenade, H., Turk, M., Shapero, D., Banihirwe, A., and Leeman, J. Pooch: A friend to fetch your data files. *Journal of Open Source Software*, 5(45):1943, 2020. doi: 10.21105/joss.01943. URL https://doi.org/10.21105/joss.01943.

Vardanian, A. USearch, October 2023. URL https://github.com/unum-cloud/usearch.

Vattani, A. k-means requires exponentially many iterations even in the plane. In Hershberger, J. and Fogel, E. (eds.), *Proceedings of the 25th ACM Symposium on Computational Geometry, Aarhus, Denmark, June 8-10, 2009*, pp. 324–332. ACM, 2009. doi: 10.1145/1542362.1542419. URL https://doi.org/10.1145/1542362.1542419.

Wang, J., Wang, J., Ke, Q., Zeng, G., and Li, S. Fast approximate k-means via cluster closures. In Baughman, A. K., Gao, J. J., Pan, J., and Petrushin, V. A. (eds.), *Multimedia Data Mining and Analytics - Disruptive Innovation*, pp. 373–395. Springer, 2015. doi: 10.1007/978-3-319-14998-1_17. URL https://doi.org/10.1007/978-3-319-14998-1_17.

Wang, M., Xu, X., Yue, Q., and Wang, Y. A comprehensive survey and experimental comparison of graph-based approximate nearest neighbor search. *Proc. VLDB Endow.*, 14(11):1964–1978, 2021. doi: 10.14778/3476249.3476255. URL http://www.vldb.org/pvldb/vol14/p1964-wang.pdf.

Wang, Z. Pyanns, 2023. URL https://github.com/veaaaab/pyanns. NeurIPS 2023 talk: https://nips.cc/virtual/2023/84327.

Weaviate. Weaviate, 2024. URL https://github.com/weaviate/weaviate.

Yu, S. *Parallel Algorithms, Optimizations, and Benchmarks for Metric and Graph Clustering*. PhD thesis, Massachusetts Institute of Technology, 2024. URL https://hdl.handle.net/1721.1/156653.

Yu, S., Engels, J., Huang, Y., and Shun, J. Pecann: Parallel efficient clustering with graph-based approximate nearest neighbor search. *arXiv preprint arXiv:2312.03940*, 2023.

Yu, Y., Wen, D., Zhang, Y., Qin, L., Zhang, W., and Lin, X. Gpu-accelerated proximity graph approximate nearest neighbor search and construction. In *38th IEEE International Conference on Data Engineering, ICDE 2022, Kuala Lumpur, Malaysia, May 9-12, 2022*, pp. 552–564. IEEE, 2022. doi: 10.1109/ICDE53745.2022.00046. URL https://doi.org/10.1109/ICDE53745.2022.00046.

Zhao, W., Tan, S., and Li, P. SONG: approximate nearest neighbor search on GPU. In *36th IEEE International Conference on Data Engineering, ICDE 2020, Dallas, TX, USA, April 20-24, 2020*, pp. 1033–1044. IEEE, 2020. doi: 10.1109/ICDE48307.2020.00094. URL https://doi.org/10.1109/ICDE48307.2020.00094.

# A. Further Background

In this section, we discuss the additional background information omitted from Section 2.

## A.1. $k$-Means

A typical implementation of $k$-means clustering in a larger application (for example, SciKit-Learn (Pedregosa et al., 2011)) would be the following:

1. (Optionally) use a sub-sampling technique to reduce the dataset size.
2. Initialize the centroids.
3. Use a local search technique to improve the solution.
4. Terminate after a pre-specified number of iterations or amount of time.

Each of these steps has a variety of avenues for improvement, see the survey by Ikotun et al. (2023). For each, we will summarize only some of the most important methods potentially relevant to our case of large $k$. In particular, sub-sampling is quite limited for large $k$, since the number of points per cluster may already be quite small.

**Existing work on initialization** The easiest and most efficient initialization method is to uniformly sample points. However, strong initialization methods are often desirable since they can obtain provable approximation ratios. Some particularly popular and easily implementable algorithms include `k-means++` (Arthur & Vassilvitskii, 2006), `scalable k-means++`/`k-means||` (Bahmani et al., 2012), and `multi-swap k-means++` (Beretta et al., 2024)). However, even the theoretically fastest of these algorithms (`k-means||`) takes time at least $\Omega(k^2)$ (a tighter lower bound in terms of some slightly different parameters is given in (Bahmani et al., 2012)). For large values of $k$, this is still far slower than uniformly random initializations. Standard implementations support this observation: SciKit-Learn's (Pedregosa et al., 2011) implementation of `k-means++`, the reference implementation of `k-means||` (Bahmani et al., 2012), and the cuML implementation of each (Raschka et al., 2020), all seem to be fairly slow on even moderately sized datasets.

**Approximation guarantees** There has been prior work on providing approximation guarantees for the $k$-means problem using techniques related to local search. One example is a $(9+\epsilon)$-factor approximation algorithm by Kanungo et al. (2002b), and a PTAS when the dimension $d$ is fixed by Friggstad et al. (2019). It is also known that there is no PTAS for arbitrary dimension (Awasthi et al., 2015).

**Existing work on Local Search Methods** We previously mentioned Lloyd's algorithm as a popular local search method for $k$-means clustering. There are also other local-search methods that have been well-studied. However, Lloyd's algorithm has remained standard and has continued to show strong results in practice, particularly for high-dimensional datasets. One reason for this is that Lloyd's algorithm is highly parallelizable. See the survey by Ikotun et al. (2023) for an overview. That said, all techniques we will discuss in this work could be generalized to many variations of Lloyd's algorithm clustering that subsample data points in each step, but studying the effectiveness of such techniques is more difficult.

Although Lloyd's algorithm obtains good results in practice, it is known that there are two-dimensional datasets (Vattani, 2009) for which Lloyd's algorithm takes an exponential number of iterations to converge. Moreover, without a careful initialization, it may produce an arbitrarily bad clustering (Arthur & Vassilvitskii, 2006). Practical implementations often use a time limit or an iteration limit (see implementations in the popular SciKit-Learn (Pedregosa et al., 2011) and FAISS (Douze et al., 2024) libraries), instead of waiting for convergence.

A natural question is whether the execution speed of Lloyd iterations can be improved, which is essentially equivalent to asking if step 2 (assignment) can be accelerated. This question has been studied in several contexts. For low-dimensional data, various methods are quite effective at accelerating this step exactly, including the use of k-d trees (Kanungo et al., 2002a), and methods based on the triangle inequality (Elkan, 2003; Hamerly, 2010; Hamerly & Drake, 2015)[2]. However, in many practical applications, we would like to run $k$-means in higher dimensions for applications such as natural language

[2]For a comprehensive discussion of these techniques, their limitations in high dimensions, and other methods of acceleration (like the use of parallelism), see the book chapter by Hamerly & Drake (2015).

processing (e.g. word2vec recommends 100 to 300 dimensions (Mikolov et al., 2013), and the dense retriever model of Karpukhin et al. (2020) uses 768 dimensions) and neural network embeddings (e.g. the image embeddings of Hu et al. (2020) are 154-dimensional). Unfortunately, there are known lower bounds for exact techniques to accelerate this step in high-dimensional datasets (Borodin et al., 1999). A few works have attempted to bypass exact nearest-neighbor search by leveraging classical techniques for approximate nearest-neighbor search. We discuss these works in Appendix A.2.

### A.2. Related Work

Compared to the vast literature on $k$-means clustering as a whole, we are only aware of a handful of works that have attempted to apply any form of approximate nearest-neighbor search to any form of clustering.

**Works using ANNS to Accelerate Lloyd's algorithm** A few existing works have applied methods for approximate nearest-neighbor search in a black-box fashion to accelerate Lloyd's algorithm (or variants) in various contexts of $k$-means clustering. Note that we will omit discussion of methods that are essentially just dimension reduction techniques, for which there are many works.

Several of these use a similar approach to the general "black-box" methodology we thoroughly test in Section 3.2. Philbin et al. (2007) presented a method greedily traversing randomized k-d trees as an ANNS heuristic for this purpose, but they do not test their solution w.r.t. $k$-means clustering score (although some later work uses their solution as a baseline). Gong et al. (2015) applied techniques for locality-sensitive hashing (LSH) — which is a subclass of space-partitioning methods (see the survey by Jafari et al. (2021)) — to binary data under Hamming distance with "mini-batch" $k$-means local search (Sculley (2010) gives a discussion of mini-batches, which can be seen as a modified form of Lloyd's algorithm that maintains parallelizability). Hu et al. (2017) present a similar method, instead using Hamming LSH with a "reranking" step to cluster Euclidean data via binary code quantization. Note that ANNS over Hamming distance is generally easier than Euclidean distance, since it a special case. Moreover, locality-sensitive hashing is now significantly outperformed by modern ANNS techniques in practice (Aumüller et al., 2020), so this is likely not the most effective use of this black-box approach (as we will see in Section 3.2). As part of an implementation for a variant of "Product Quantization", Baranchuk et al. (2018) applied HNSW to $k$-means clustering over relatively small chunks of data. Their method for doing so is similar to our initial "black-box" methodology presented in Section 3.2, although they focus on a much smaller-scale case, and they did not provide any empirical justification for their choice of HNSW over other ANNS methods, nor did they empirically test their methods w.r.t. the $k$-means clustering objective (nor was obtaining a good clustering score their goal).

A few works have also explored a different method of applying ANNS techniques to the assignment stage of Lloyd iterations: Instead of assigning dataset points in $P$ to their nearest center in $C$, the centers can instead "flood fill" the dataset using certain types of ANNS data structures constructed over $P$ instead of $C$. We will call this the **inverse assignment** method. Kanungo et al. (2002a) applied the inverse assignment method to compute *exact* assignments. Avrithis et al. (2015) applied the inverse assignment method by essentially projecting into a quantized two-dimensional space (thereby doing a quantized dimension reduction). Wang et al. (2015) also employ a variation of the inverse assignment method by constructing an approximate neighborhood graph, which they then leverage to prune distance computations. In particular, they construct their graph using random-projection trees (Dasgupta & Freund, 2008), an ANNS space-partitioning method. Unfortunately, the inverse assignment method is limited to (small) data sets that can fit wholly in-memory, since it requires a more careful traversal of a specialized ANNS data structure built for $P$, rather than $C$. This is in contrast to methods that build structures over the "forward" assignment method (including ours), which only need to build and store a structure for $k$ points in the metric space — even for excessively large values of $k$ (e.g. $k = |P|/100$), this is still a very significant difference in memory usage for massive datasets. There is one method for which this limitation can be overcome for the inverse method: Matsui et al. (2017) suggest using product quantization on the input vectors. Since this is a quantization method, rather than another form of ANNS method, applying the inverse method does not actually prune any distance computations, but rather just speeds them up individually. The experimental results of Matsui et al. (2017) suggest that, although faster, their method cannot achieve the same score as Lloyd's algorithm, very quickly arriving at poor local optimums even for quite small values of $k$. In particular, since their method amounts to a brute force with quantization methods, this suggests that any similar approaches applying an ANNS technique to Lloyd's algorithm involving any sort of quantization is likely to result in poor local optimums. As we will see, this appears to be true in our results as well.

Compared to all of these approaches, we present a more complete analysis of ANNS methods for Lloyd's algorithm, and we furthermore determine that they are not effective without further work. Moreover, we complete this further work, eventually devising seeded search-graphs.

**Other Forms of Clustering** To the best of our knowledge, only one academic work has studied the application of approximate nearest-neighbor search to large-scale clustering that is not $k$-means clustering: PECANN (Yu et al., 2023; Yu, 2024) studies the application of black-box approximate nearest-neighbor search methods to hierarchical density-based clustering.

Although not an academic work, the software library USearch (Vardanian, 2023) can produce a hierarchical clustering using HNSW, although the developer has not publicized any experimental results or detailed documentation on their methodology, and the feature is still marked as in-development. Their method appears to involve treating each point on a non-zero level as a cluster "center", and performing a simultaneous flood fill from all points in a non-zero level to the points of the dataset in the level below. This would be similar to performing a random sample initialization of $k$-means, with no local search iterations, and performing the "assignment" stage in a way so as to perform graph-based clustering (i.e., approximating geodesic distance over a manifold approximated by the graph).

| Dataset | Type | Dim | Points | Size |
|---|---|---|---|---|
| **SIFT1B** | Image | 128 | 1 bil. | 512.00 GB |
| **SIFT20M** | Image | 128 | 20 mil. | 10.24 GB |
| **Text2Image10M** | Image | 200 | 10 mil. | 8.00 GB |
| **DPR5M** | Text | 768 | 5 mil. | 15.36 GB |

Table 2: Description of the datasets used in experiments.

## B. Experimental Setup

We ran multiple rounds of experiments, so in this section, we present the shared details of our experimental setup.

**Environment** We conducted the experiments on a workstation machine with Ubuntu 22.04.5 LTS, equipped with an AMD Ryzen 9 7950x CPU, 64GB of RAM, an Nvidia RTX 3090 GPU, and datasets stored in a 2TB SSD. Note that this large amount of RAM is primarily for testing baseline code and existing libraries — our own methods will not use a significant amount of RAM for any dataset, and (most importantly) will not require storing the entire dataset in memory at any time. Note also that, at the time of writing, this GPU is significantly more expensive (by a factor of $2\times$ in most marketplaces) than the CPU.

All timed CPU-based algorithms were allowed to use a maximum of 12 threads to reduce possible conflicts with operating system processes. No limits were placed on the algorithms using the GPU.

**Software Libraries** We will make use of the following libraries for various baselines:

- SciKit Learn (Pedregosa et al., 2011) on CPU
- cuML (Raschka et al., 2020) on GPU
- PyTorch (Paszke et al., 2019) on GPU
- FAISS (Douze et al., 2024) on CPU

We use SciKit Learn, cuML, and PyTorch for reference implementations of Lloyd's algorithm and initialization routines. In particular, SciKit Learn and cuML both offer built-in implementations of Lloyd's algorithm and initialization methods, while PyTorch enabled us to write a straightforward, short, and highly-efficient GPU-based implementation of Lloyd's algorithm. We use FAISS for its implementations of various baseline approximate nearest-neighbor search routines. Note that none of these libraries are used for our own code, for which we discuss implementation details in Appendix E.

**Datasets** We tested on two image-embedding datasets, and one text-embedding dataset:

- Yandex's **Text2Image10M** dataset (Simhadri et al., 2022) which consists of images embeddings produced by the `Se-ResNext-101 model` (Hu et al., 2020). This data set was used for benchmarking for the NeurIPS 2023 large-scale ANNS competition (Simhadri et al., 2024). Typically, this is used as a cross-modal data set, with an ANNS query set derived from text embeddings, but in this paper we use Text2Image10M purely for its image embeddings, since clustering these points is what one would do for our suggested application (see Section 2.1.1).
- The **SIFT1B** dataset (Jégou et al., 2011), and a 20M slice of the SIFT1B dataset that we will call **SIFT20M**. These datasets are 128-dimensional image descriptors in SIFT format (Lowe, 2004). This dataset is also frequently used for benchmarking large-scale out-of-core ANNS algorithms (Baranchuk et al., 2018; Johnson et al., 2021; Jayaram Subramanya et al., 2019).
- The **DPR10M** dataset generated by Aguerrebere et al. (2023) from 768-dimensional dense passage retriever model of (Karpukhin et al., 2020). To make the dataset of comparable size to the other ones, we created **DPR5M** by taking a 5M slice of DPR10M. This is a higher dimensional text-based dataset, to contrast with the other datasets.

Since these datasets are quite large, we are unfortunately unable to distribute them ourselves. However, we provide instructions for reproduction of these datasets as part of our code. For development purposes, we also used **SIFT10K** and **SIFT1M**. For these smaller datasets, we leveraged the Pooch (Uieda et al., 2020) library for retrieval and caching.

We note that our comparison plots use the $10$-million and $5$-million sized datasets only. This is primarily due to limitations in our baselines, rather than any limitations in SHEESH.

## C. Full Results

In this section, we present the full set of results for our experiments.

For black-box acceleration, we tested the following families of popular methods:

- Baseline: Lloyd's algorithm (3 implementations, 2 of which were on GPU)
- Quantization-only techniques: Scalar Quantization (Liu et al., 2024), Product Quantization (Matsui et al., 2018)
- Clustering-only techniques: IVF (Sivic & Zisserman, 2003), ScaNN (Guo et al., 2020) (with quantization disabled, amounting to a $k$-means tree)
- Combined clustering+quantization techniques: ScaNN (Guo et al., 2020), IVFPQ (Jegou et al., 2010), IVFPQR (Jégou et al., 2011)
- Search-graph techniques: NN-Descent (Dong et al., 2011), HNSW (Malkov & Yashunin, 2018), NSG[3] (Fu et al., 2019)

Whenever possible, we leveraged implementations given in FAISS (Douze et al., 2024). For ScaNN, we used the reference implementation, which itself is called ScaNN (Guo et al., 2020; Sun et al., 2024). For a baseline CPU implementation, we used SciKit-Learn (Pedregosa et al., 2011)'s CPU implementation of Lloyd's algorithm. We also compared against two GPU implementations of Lloyd's algorithm: The GPU implementation in cuML (Raschka et al., 2020), as well as a simple implementation of our own using PyTorch (Paszke et al., 2019). We note that, in some sense, comparing the CPU-based approaches to GPU implementations is unfair, especially since our GPU is significantly more expensive than our CPU (as noted in Appendix B). However, practical implementations of $k$-means clustering very frequently use GPU acceleration (e.g., (Douze et al., 2024)), so this is an important comparison to make when attempting to devise a method to be used in practice. Moreover, since our experiments suggest that our final methods are far superior to the GPU accelerated implementations, they serve as a good reference point.

We had a number of parameters that varied for each black-box technique, as well as some that varied for our own techniques discussed in Section 4. Most notably, we have the "avoid_regress" parameter (the simple improvement discussed in Section 3.2). For the quantization-only techniques, we opted not to re-run with avoid_regress=true, since they were excessively slow. For the ScaNN library, we tried many combinations of parameters, including turning on/off quantization. For our own techniques, we also had several parameters. In particular, we toggled several of the different strategies discussed in Section 4 to study their effectiveness. We give a legend of all parameter variations in Table 3, which can be used as reference for all of our experimental plots.

For the algorithm parameters we did not vary, we generally applied sane/recommended defaults. In particular, for our methods, as well as the black-box tests with HNSW, we used the following parameters:

- $\texttt{ef\_build} = 200$
- $\texttt{M} = 60$
- $\texttt{ef\_search} = 10 \times \texttt{num\_prev\_assignments}$
- $\texttt{min\_iterations} = 2 \times \texttt{ef\_search} + 1 = 21$

There is almost certainly some improvement to be gained by better tuning these parameters to each dataset, but we have not done so.

We tested each algorithm on each dataset (listed in Appendix B) with each variation of parameters, using a 500 second timeout — we record the score up to and including the first iteration that exceeds the 500 second threshold, at which point we halt the algorithm. Note that many such scores vastly exceed the 500 second threshold for cases in which the algorithm takes

[3]While we attempted to thoroughly test NSG, which showed promising preliminary results similar to HNSW, we were not able to evaluate it in most of our experiments, for the following reason: At the time of writing, there appears to be an occasional bug in the FAISS implementation of NSG that can result in an infinite loop, so we disabled it while performing most of our experiments. It still appears in one of our plots, where it shows good performance.

| Tagged Label | Parameters |
|---|---|
| ivf-PQ[a] | avoid_regress: true |
| ivf-PQ[b] | avoid_regress: false |
| hnsw[a] | avoid_regress: true |
| hnsw[b] | avoid_regress: false |
| ivf-PQr[a] | avoid_regress: true |
| ivf-PQr[b] | avoid_regress: false |
| ivf-flat[a] | avoid_regress: true |
| ivf-flat[b] | avoid_regress: false |
| scann[a] | num_leaves: 200, num_leaves_to_search: 10, use_score_ah: true, reorder_size: 100, avoid_regress: false |
| scann[b] | num_leaves: 200, num_leaves_to_search: 10, use_score_ah: false, reorder_size: null, avoid_regress: false |
| scann[c] | num_leaves: 500, num_leaves_to_search: 10, use_score_ah: true, reorder_size: 100, avoid_regress: false |
| scann[d] | num_leaves: 500, num_leaves_to_search: 10, use_score_ah: false, reorder_size: null, avoid_regress: false |
| scann[e] | num_leaves: 200, num_leaves_to_search: 10, use_score_ah: true, reorder_size: 100, avoid_regress: true |
| scann[f] | num_leaves: 200, num_leaves_to_search: 10, use_score_ah: false, reorder_size: null, avoid_regress: true |
| scann[g] | num_leaves: 500, num_leaves_to_search: 10, use_score_ah: true, reorder_size: 100, avoid_regress: true |
| scann[h] | num_leaves: 500, num_leaves_to_search: 10, use_score_ah: false, reorder_size: null, avoid_regress: true |
| nndescent[a] | avoid_regress: false |
| nndescent[b] | avoid_regress: true |
| sheesh[a] | use_rebuilds: true, num_prev_assignments: 10, enable_seeds: true, enable_bulk: true, enable_min_iter: true |
| sheesh[b] | use_rebuilds: false, num_prev_assignments: 10, enable_seeds: true, enable_bulk: true, enable_min_iter: true |
| sheesh[c] | use_rebuilds: false, num_prev_assignments: 10, enable_seeds: true, enable_bulk: false, enable_min_iter: true |
| sheesh[d] | use_rebuilds: false, num_prev_assignments: 10, enable_seeds: true, enable_bulk: false, enable_min_iter: false |
| sheesh[e] | use_rebuilds: false, num_prev_assignments: 1, enable_seeds: true, enable_bulk: false, enable_min_iter: false |
| sheesh[f] | use_rebuilds: false, num_prev_assignments: 1, enable_seeds: false, enable_bulk: false, enable_min_iter: false |

Table 3: Legend for algorithms with multiple parameter variations.

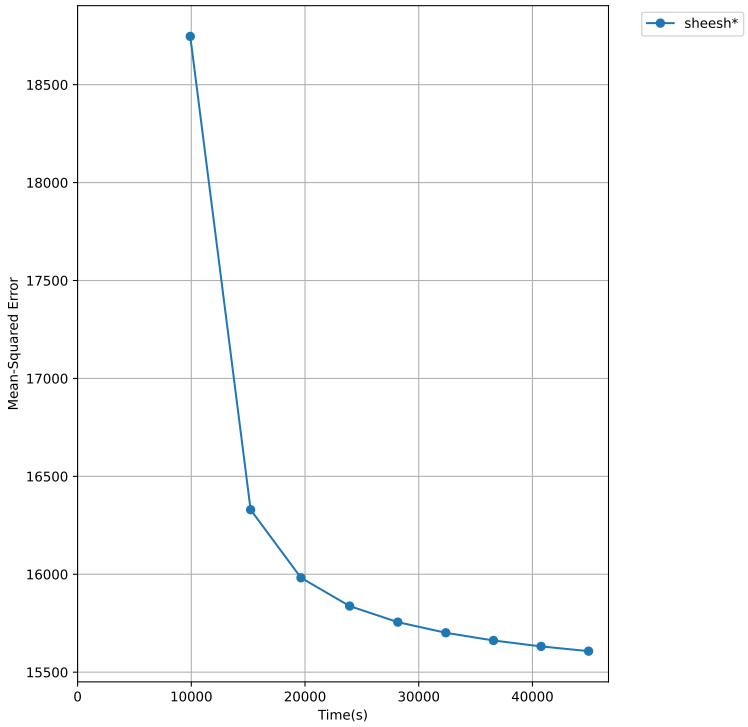
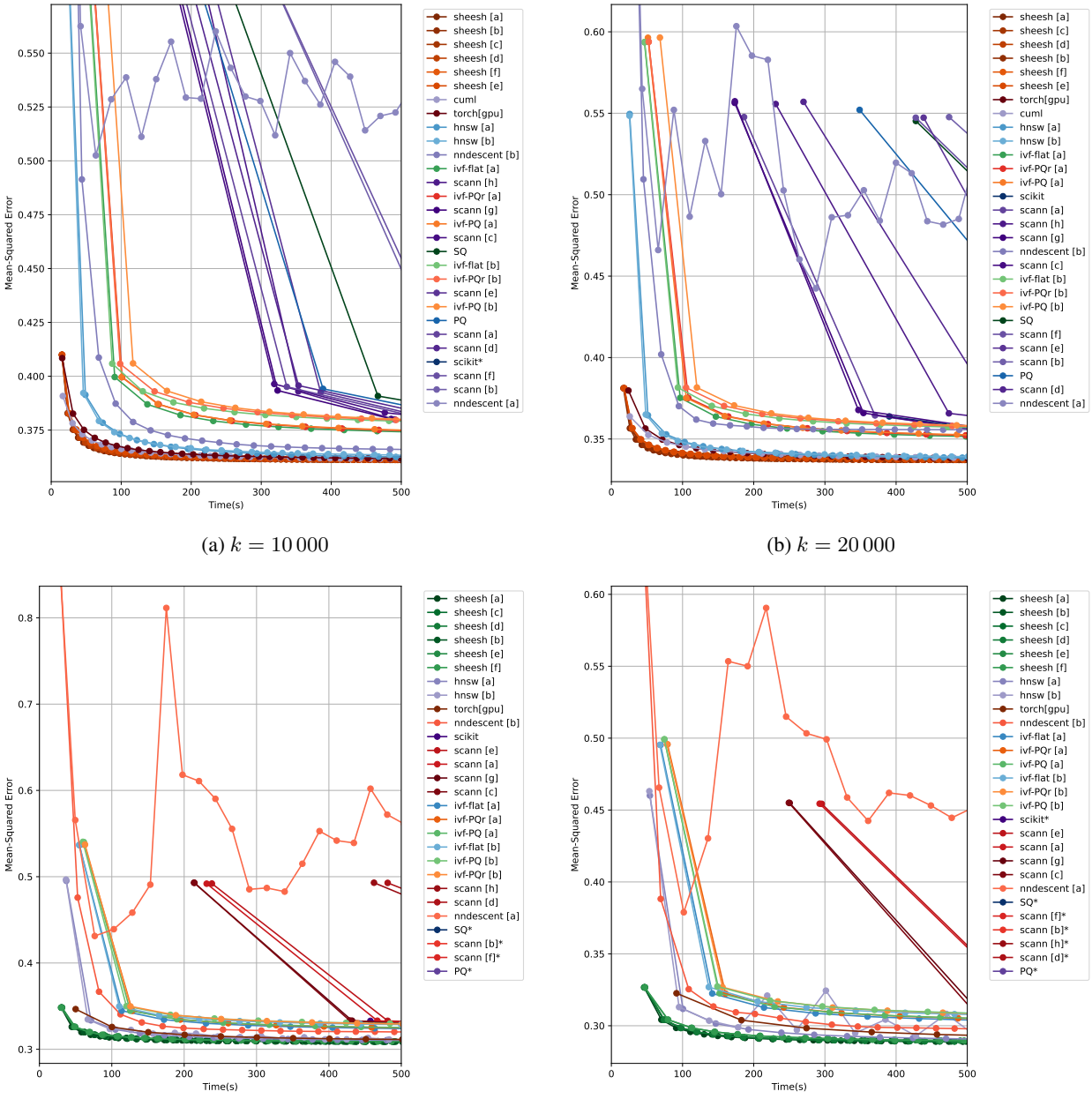



Figure 4: A plot of SHEESH running on SIFT1B with $k = 1\,000\,000$ for just over 12 hours. Initialization is uniformly random. We estimate SciKit-Learn would take roughly 9.5 days to run a single iteration in this case.

a long time to compute a single iteration. Note also that the cuML implementation of Lloyd's algorithm ran out of VRAM for several cases (from which it is omitted), but generally exhibited similar performance to our PyTorch implementation in those where it did not. We have plotted all of our data in Figures 5 to 7. We also performed one additional limited test, to demonstrate the scalability of our algorithm, plotted in Figure 4. To aid in the reading of our plots, we have sorted all entries in each legend by their best score. This is true of *every plot in the paper*. In particular, with this information, one can see that our algorithm (with all features enabled, sheesh [a] from the table) achieved the best score in every single experiment we performed. In addition, any plot entries marked with a "*" timed out (took $> 500$ seconds to finish their first iteration).



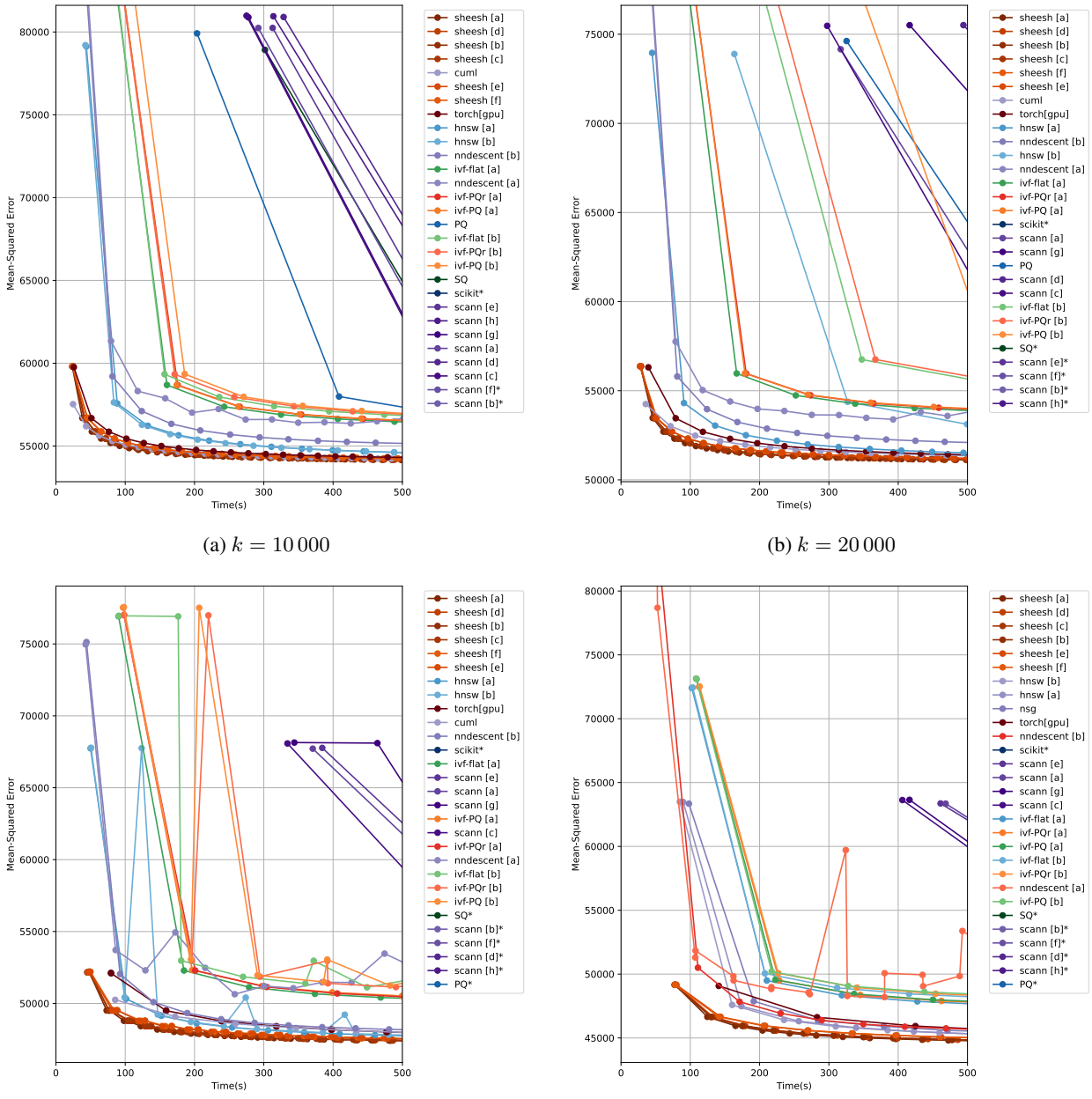

(a) $k = 10\,000$

(b) $k = 20\,000$

(c) $k = 50\,000$

(d) $k = 100\,000$

Figure 5: Comparisons of all methods on the Text2Image10M dataset, for all tested values of $k$. Initialization is uniformly random.



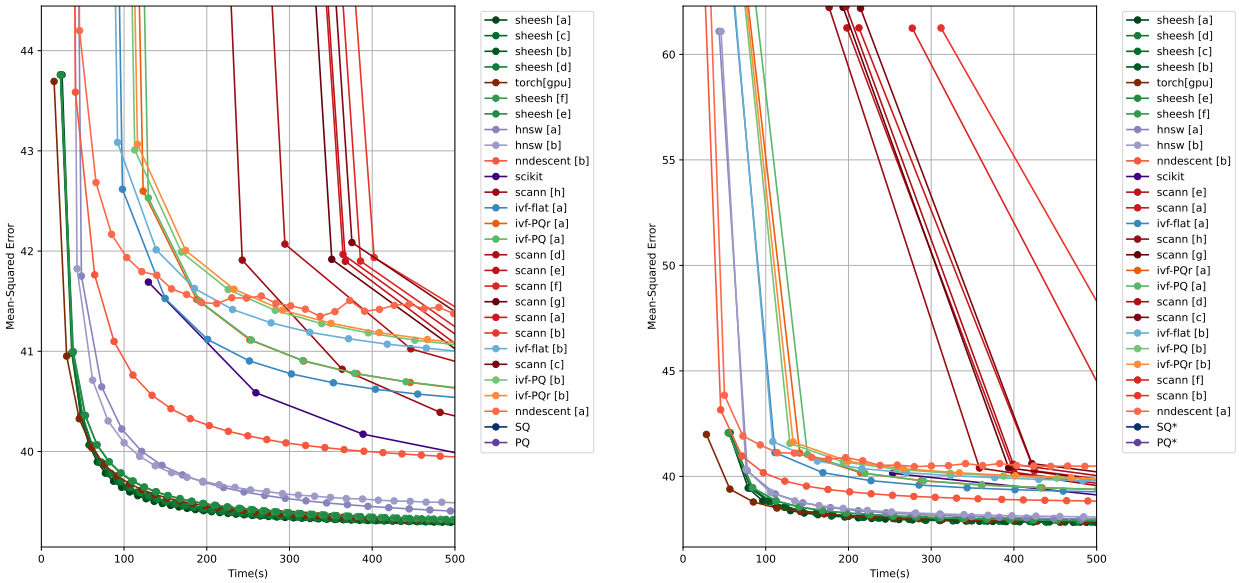

(a) $k = 10\,000$

(b) $k = 20\,000$

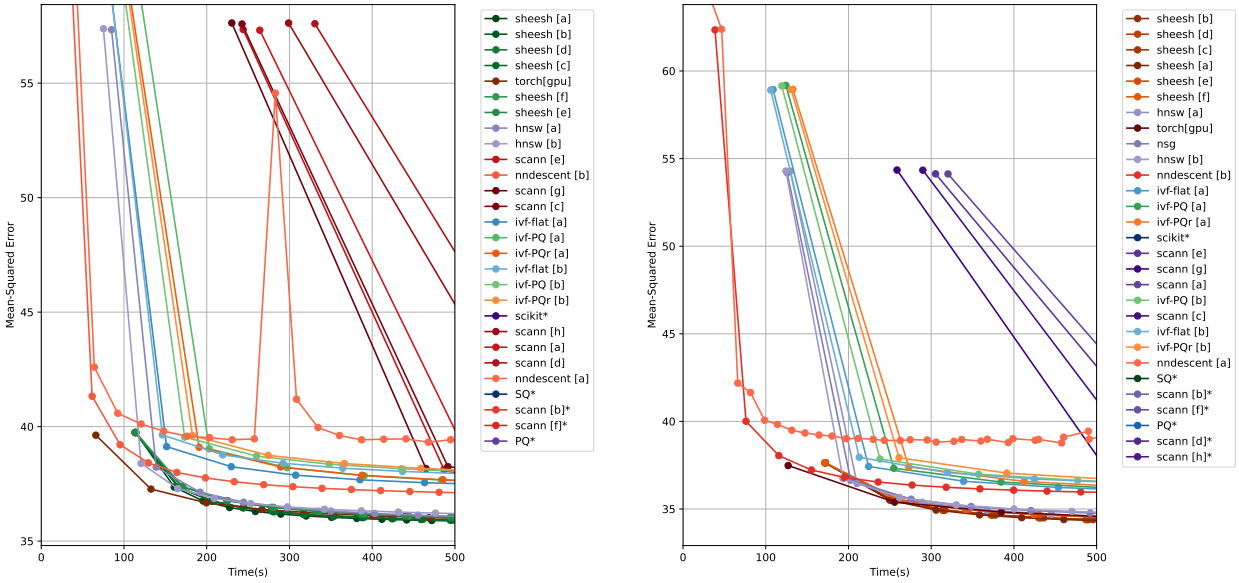
(c) $k = 50\,000$

(d) $k = 100\,000$

Figure 6: Comparisons of all methods on the SIFT20M dataset, for all tested values of $k$. Initialization is uniformly random.




(a) $k = 10\,000$ (b) $k = 20\,000$




(c) $k = 50\,000$ (d) $k = 100\,000$

Figure 7: Comparisons of all methods on the DPR5M dataset, for all tested values of $k$. Initialization is uniformly random.

## D. A Graph-Search Algorithm for SANNS with Provable Guarantees

In this section, we observe that analysis for an existing search-graph algorithm with provable guarantees for ANNS (given by Indyk & Xu (2023)) naturally extends to become a seeded search-graph algorithm with provable guarantees for SANNS. That is, we show that the algorithm easily adapts to become a form of learning-augmented algorithm, with robustness and consistency guarantees.

Indyk & Xu (2023) present a modified form of the "Vamana" data structure given by Jayaram Subramanya et al. (2019) for ANNS over a pointset $P \subset \mathbb{R}^d$ with some provable guarantees. For clarity, we will refer to the modified data structure as **Vamana with slow preprocessing**, or simply **VamanaSP**. In particular, their provable guarantees are in terms of several parameters we must introduce:

- The **aspect ratio** $\Delta$ is a property of $P$. Specifically, it is the ratio $D_{\max}/D_{\min}$ between the distance of the furthest pair $D_{\max}$ and the distance of the closest pair $D_{\min}$ (where $D$ is the distance function of the metric space, usually Euclidean distance in the rest of our paper).
- The **doubling dimension** $d'$ is also a property of $P$. For simplicity of presentation, we omit its format definition here, but it can be considered a measure of "intrinsic dimensionality" of the dataset in the same sense as discussed at the end of Section 2.1.
- The parameter $\alpha > 1$ is a preprocessing-time parameter of both Vamana and VamanaSP.
- The parameter $\epsilon > 0$ is a query-time parameter for tuning the approximation ratio of the nearest-neighbor returned by VamanaSP.

Indyk & Xu (2023) give a preprocessing algorithm for VamanaSP running in $O(|P|^3)$ time. This constructs a search-graph, which they show has maximum degree $O((4\alpha)^{d'} \log \Delta)$. Using Algorithm 1 for queries with $b = 1$, they show that only $O\left(\log_\alpha \frac{\Delta}{(\alpha-1)\varepsilon}\right)$ node visits are sufficient to find a $\left(\frac{\alpha+1}{\alpha-1} + \varepsilon\right)$-approximate nearest-neighbor. Note that each node visit requires $O((4\alpha)^{d'} \log \Delta)$ distance computations. Note that doubling dimension is NP-hard to compute (Gottlieb & Krauthgamer, 2013), so it is unclear if it is expected to be a small quantity in a typical dataset.

We claim that, if seeded with a "learned" element of $P$, running Algorithm 1 on their constructed graph constitutes a form of learning-augmented algorithm. In particular, it has the following two high-level properties, in terms of the tradeoff between iteration count and approximation ratio:

- **Robustness**: It maintains worst-case guarantees.
- **Consistency**: If the seed point already has a good approximation ratio, then the tradeoff between iteration count and approximation ratio *improves*.

To prove this, we will simply leverage the techniques of Indyk & Xu (2023). In particular, they showed that the aforementioned guarantees for Algorithm 1 on VamanaSP hold *regardless of the initial starting vertex*, meaning we obtain robustness for free from their analysis.

We now present a brief proof that (seeded) VamanaSP also has a form of consistency, which we will again prove by leveraging the analysis of Indyk & Xu (2023):

**Theorem D.1.** *Let $q$ be a query point whose nearest-neighbor in $P$ is $a$. Let $s \in P$ be a point so that $1 + \delta \geq \frac{D(s,q)}{D(a,q)}$, for some value $\delta > 0$. Then Algorithm 1 starting at $s$ returns a $\left(\frac{\alpha+1}{\alpha-1} + \varepsilon\right)$-approximate nearest-neighbor in $\left\lceil \log_\alpha \frac{1+\delta}{\epsilon} \right\rceil$ node visits.*

*Proof.* Let $p_i$ be the $i$th point visited during the execution of Algorithm 1. Let $d_i$ be the distance $D(p_i, q)$. As part of their proof, Indyk & Xu (2023) show that $d_i \leq \frac{D(s,q)}{\alpha^i} + \frac{\alpha+1}{\alpha-1} D(a, q)$. In particular, it also follows from their argument that the algorithm will only terminate without intervention if it has found an $\left(\frac{\alpha+1}{\alpha-1}\right)$-approximate nearest-neighbor. We leverage this

in a straightforward manner: If $i \geq \log_\alpha \frac{1+\delta}{\epsilon}$, then $\varepsilon \geq \frac{1+\delta}{\alpha^i} \geq \frac{D(s,q)}{D(a,q)\alpha^i}$. Thus:

$$d_i \leq \frac{D(s,q)}{\alpha^i} + \frac{\alpha+1}{\alpha-1}D(a,q) \leq \varepsilon D(a,q) + \frac{\alpha+1}{\alpha-1}D(a,q) = \left(\frac{\alpha+1}{\alpha-1} + \varepsilon\right) D(a,q).$$

□

Although Indyk & Xu (2023) note that the aspect ratio of typical datasets tends to be quite small, this is not always necessarily the case. Moreover, they note that their algorithm essentially cannot have the worst-case $\log \Delta$ factor in the number of node visits replaced with a $\log |P|$ factor (they give a more formal argument for this than we provide here). Their proof of this relies on the fact that they have no guarantees about the starting vertex. Hence, it is reasonable to assume that we may sometimes be able to leverage our consistency guarantee to offer a slightly different algorithm with an analogous improved ratio. In fact, we can sometimes do even better than this, by applying methods that can obtain $O(\log n)$-approximate nearest-neighbors in well-known metric spaces. In particular:

**Corollary D.2.** *Assume an oblivious adversary. For data embedded in $\mathbb{R}^d$, under the Euclidean, inner-product, or cosine distances, there exists an algorithm that, with $O(nd^2)$ preprocessing time, can, in $O(d^2 + \log n)$ time per query, produce a seed point for VamanaSP requiring expected at most $\log_\alpha \frac{\min(O(\log n)}{\varepsilon}$ node visits to produce a $\left(\frac{\alpha+1}{\alpha-1} + \varepsilon\right)$-approximate nearest-neighbor.*

In particular, such a seed point can be easily achieved with a random projection of $P$, and the proof follows. Note that Euclidean/inner-product/cosine distance are all isometric up to the inclusion of one extra dimension (Bachrach et al., 2014). Note also that many other forms of (fixed approximation-ratio) ANNS could be applied to further accelerate this method. However, we have highlighted the ability to use random projection in particular since it is somewhat related to our use of random projection in the main body of our work for BSANNS (see Section 4.1). For instance, we believe it would be interesting to determine if a "bulk" method using *only* random projection (no grouping step) could obtain interesting amortized guarantees. That is, an approach where a bulk query of $O(|P|)$ points could be randomly projected into $\mathbb{R}$ for sorting purposes, and then the results of each query could be provided as seed points for the next.

Overall, in this section, we have given a nice extension of VamanaSP to SANNS. Unfortunately, the preprocessing step of VamanaSP is too slow for our overall goal of $k$-means clustering with large $k$, but this result is still of interest from the theoretical viewpoint of the SANNS problem. We believe a promising avenue for future theoretical work on SANNS would be to work with *greedy trees* (Chubet et al., 2023). For fixed doubling dimension, Chubet et al. (2023) show that greedy trees have better approximation guarantees than VamanaSP, and the work of Har-Peled & Mendel (2006) can be leveraged to show that they can be computed with near-linear preprocessing time.

## E. Implementation Details

In this section, we discuss some details of our specific implementation that aided us in obtaining our final performance.

### E.1. HNSW Implementation

Our base HNSW implementation has been carefully tuned. The majority of its runtime when analyzed in a profiler (Linux perf (Linux Kernel Organization, 2024) and KDBA Hotspot (KDAB, 2020)) is taken up by distance computations. In particular, most of these distance computations take place within Algorithm 1. The distance computations are accelerated with careful manual **prefetching** (preemptive insertions into cache) and SIMD-acceleration. We implemented SIMD-accelerated distance queries with the Eigen `C++` library (Guennebaud et al., 2010) for simplicity. Manual prefetching is important since search-graph approaches to ANNS inherently have almost zero data-locality — they involve comparing high-dimensional vectors according to an unpredictable graph traversal. Consequently, compilers and hardware prefetchers are not well-equipped to predict cache lines to prefetch.

In particular, the reference implementation for HNSW (Malkov & Yashunin, 2018) (‘`hnswlib`”) did not implement prefetching in the optimal manner. We summarize their prefetching scheme for beam search in Algorithm 3 (slightly simplified for presentation). We identified three areas for improvement in their implementation:

- It only ever prefetches exactly one vector ahead (it is unparameterized).
- For each vector, it only fetches the first cache line containing the vector. For sufficiently high-dimensional data, it does not manually prefetch entire next vector (only first cache line). This may be mitigated by smart compilers or hardware prefetchers, but (as far as we know) such mitigations are not guaranteed.
- When the current vector has already been visited, it does not prefetch any part of the next vector, even if that vector is unvisited.

**Algorithm 3** beam search with hnswlib’s prefetching scheme

**Input:** $P \subset \mathbb{R}^d$, search-graph $G = (P, E)$, $p^* \in P$, $q \in \mathbb{R}^d$, $b \in \mathbb{Z}_{\geq 1}$
Initialize sets $C, N = \{p^*\}$ (candidates, nearest).
Mark $p^*$ as visited.
**repeat**
  Extract the element $c$ from $C$ nearest to $q$.
  **if** $|N| = b$ and $d(c, q) > d(n, q)$ for all $n \in N$ **then**
    **break**
  **end if**
  **Prefetch:** The first cache line of the data for the first neighbor of $c$.
  **for** each (outgoing) neighbor $v$ of $c$ in $G$ **do**
    **if** $v$ is not marked as visited **then**
      Mark $v$ as visited
      **Prefetch:** The first cache line of the data for the next neighbor of $c$ after $v$.
      **if** $|N| < b$ or $d(v, q) < d(n, q)$ for some $n \in N$ **then**
        Add $v$ to $C$ and $N$
        If $|N| > b$ or $|C| > b$, remove the furthest element.
      **end if**
    **end if**
    Mark $v$ as visited.
  **end for**
**until** $C$ is empty
**Output:** $N$, the $b$ points in $P$ closest to $q$.

Our own implementation is not based on `hnswlib`, and we use a more careful (and simpler) prefetching scheme, outlined in Algorithm 4 (again, slightly simplified for presentation). The main ideas are as follows:

- We make a list containing the indices of all unvisited neighbor *before* iterating through the list.
- We parameterize the distance ahead that vectors are prefetched.

**Algorithm 4** beam search with our prefetching scheme

**Input:** $P \subset \mathbb{R}^d$, search-graph $G = (P, E)$, $p^* \in P$, $q \in \mathbb{R}^d$, $b \in \mathbb{Z}_{\geq 1}$
Initialize sets $C, N = \{p^*\}$ (candidates, nearest), and an empty list $L$ (neighbor list).
Mark $p^*$ as visited.
**repeat**
  Extract the element $c$ from $C$ nearest to $q$.
  **if** $|N| = b$ and $d(c, q) > d(n, q)$ for all $n \in N$ and a sufficient number of iterations have occurred **then**
    **break**
  **end if**
  **for** each (outgoing) neighbor $v$ of $c$ in $G$ **do**
    **if** $v$ is not marked as visited **then**
      Add $v$ to $L$ and mark $v$ as visited.
    **end if**
  **end for**
  **Prefetch:** All cache lines for the first $4$ elements in $L$.
  **for** each $v$ in $L$ **do**
    **Prefetch:** All cache lines for the next element of $L$ that not yet prefetched.
    Compute $d(v, q)$.
    **if** $|N| < b$ or $d(v, q) < d(n, q)$ for some $n \in N$ **then**
      Add $v$ to $C$ and $N$.
      If $|N| > b$, remove the furthest element.
    **end if**
  **end for**
  Clear $L$.
**until** $C$ is empty
**Output:** $N$, the $b$ points in $P$ closest to $q$.

One of the NeurIPS 2023 Big-ANN competition (Simhadri et al., 2024) winners, PyANNS (Wang, 2023), also took a similar approach to prefetching. In particular, they also automatically tuned the added parameter. In contrast, we simply use a sane default of prefetching $4$ vectors ahead; a number of values seemed to exhibit essentially the same performance.

We have not carefully examined the prefetching schemes of other ANNS search-graph implementations.

### E.2. Data Streaming

Since we are dealing with datasets too large to fit in RAM, we require some form of multi-threaded data streaming system. We adopted a simple and straightforward approach leveraging the C++ template system to create an abstract container we simply call a "bucket" implementing some kind of data streaming routine using callbacks. We created several implementations of buckets, including one wrapping NumPy containers (Harris et al., 2020), which is what we used for all of our benchmarking. One could create similar bucket implementations for a database or similarly bulky tool, although we have opted to implement only simple variations, since our methods are all quite compute-limited, and the IO patterns are extremely simple (we simply stream through the dataset in order during each iteration).