# Improved Space-Efficient Approximate Nearest Neighbor Search Using Function Inversion

Samuel McCauley
Williams College Computer Science, Williamstown, MA, USA

## Abstract

Approximate nearest neighbor search (ANN) data structures have widespread applications in machine learning, computational biology, and text processing. The goal of ANN is to preprocess a set $S$ so that, given a query $q$, we can find a point $y$ whose distance from $q$ approximates the smallest distance from $q$ to any point in $S$. For most distance functions, the best-known ANN bounds for high-dimensional point sets are obtained using techniques based on locality-sensitive hashing (LSH).

Unfortunately, space efficiency is a major challenge for LSH-based data structures. Classic LSH techniques require a very large amount of space, oftentimes polynomial in $|S|$. A long line of work has developed intricate techniques to reduce this space usage, but these techniques suffer from downsides: they must be hand tailored to each specific LSH, are often complicated, and their space reduction comes at the cost of significantly increased query times.

In this paper we explore a new way to improve the space efficiency of LSH using function inversion techniques, originally developed in (Fiat and Naor 2000).

We begin by describing how function inversion can be used to improve LSH data structures. This gives a fairly simple, black box method to reduce LSH space usage.

Then, we give a data structure that leverages function inversion to improve the query time of the best known near-linear space data structure for approximate nearest neighbor search under Euclidean distance: the ALRW data structure of (Andoni, Laarhoven, Razenshteyn, and Waingarten 2017). ALRW was previously shown to be optimal among "list-of-points" data structures for both Euclidean and Manhattan ANN; thus, in addition to giving improved bounds, our results imply that list-of-points data structures are not optimal for Euclidean or Manhattan ANN.

**2012 ACM Subject Classification** Theory of computation → Sorting and searching; Theory of computation → Data compression

**Keywords and phrases** similarity search, locality-sensitive hashing, randomized algorithms, data structures, space efficiency, function inversion

**Digital Object Identifier** 10.4230/LIPIcs.ESA.2024.88

**Related Version** *Full Version*: `https://arxiv.org/abs/2407.02468`

**Funding** This work was funded in part by NSF CCF 2103813.

**Acknowledgements** I would like to thank Rasmus Pagh and Martin Aumüller for helpful discussions, and the anonymous reviewers for their comments and suggestions.

## 1 Introduction

Modern data science relies increasingly on sophisticated ways to query datasets. A fundamental class of these queries is similarity search: given a query $q$, find the item in the dataset that is most "similar" to $q$. Similarity search was originally introduced by Minsky and Papert in their seminal textbook [48]. Similarity search problems have applications in wide-ranging areas, including clustering [20, 34], pattern recognition [27], data management [52], and compression [40].

Unfortunately, for many types of similarity search queries, all known approaches require space or query time exponential in the dimension of the data [35]. This is often called the curse of dimensionality. For datasets with high – say $\Omega(\log n \log\log n)$ – dimensions, this cost quickly becomes infeasible. Fine-grained complexity results reinforce this barrier: sublinear-time algorithms for many similarity search problems would imply that the Strong Exponential Time Hypothesis is false [4, 51, 54].

Therefore, to obtain good theoretical guarantees, a long line of research has focused on approximate similarity search [38, 35]. Rather than giving the most similar point, an approximate similarity search data structure simply guarantees a point whose similarity approximates the similarity of the most similar point.

Specifically, in this work we focus on *Approximate Nearest Neighbor search (ANN).* In ANN, the data structure preprocesses a set $S$ (a subset of a universe $U$) of $n$ $d$-dimensional points for a distance function $d(\cdot,\cdot)$, a radius $r$, and an approximation factor $c > 1$. On a query $q \in U$, the data structure gives the following guarantee: if there exists a point $x \in S$ with $d(q,x) \leq r$, then with probability at least .9, the data structure returns a point $y \in S$ with $d(q,y) \leq cr$. The goal is to obtain solutions parameterized by the approximation ratio $c$, obtaining polynomial performance in $n$ and $d$ for any constant $c > 1$.

Classic results show that the definition of ANN given above can be generalized without significantly increasing the cost. The correctness guarantee of .9 is arbitrary; creating independent copies of the structure can drive the success probability arbitrarily close to 1. Past results allow $S$ to change dynamically or to work without knowing the value of $r$ up front [38], and to obtain all near neighbors (rather than just one) [1, 37]. Moreover, the approximation guarantee of ANN can be viewed as a beyond-worst-case guarantee: if the dataset is well-spaced so that there is a single point within distance $cr$ of the query, ANN guarantees that it will be returned (this was generalized further in [28]). This may explain in part why ANN solutions are effective at finding exact nearest neighbors in practice, as was seen in [28, 8].

**ANN Distance Functions.** The most widely investigated ANN problems are Euclidean and Manhattan ANN. Euclidean and Manhattan ANN have $U = \mathbb{R}^d$; Euclidean ANN uses the standard $\ell_2$ distance function for $d(\cdot,\cdot)$, while Manhattan ANN uses $\ell_1$ for $d(\cdot,\cdot)$.

There are several motivations for studying Euclidean ANN in particular. Euclidean distance is a natural and well-known metric. It is particularly useful for real-world similarity search problems [8]. Furthermore, Euclidean ANN can be used to solve other problems. For example, Manhattan ANN can be solved using a Euclidean ANN data structure using a classic embedding [11, 44]. In fact, this embedding gives the state-of-the-art Manhattan ANN bounds [11]. Euclidean ANN is similarly used in the state-of-the-art method for finding the closest point under cosine similarity [8].

**Locality-Sensitive Hashing for ANN.** Many theoretical results for ANN are based on *locality-sensitive hashing*, originally developed in [38, 35]. An LSH is a hash function where points at distance $r$ hash to the same value with probability at least $p_1$, while points at distance more than $cr$ hash to the same value with probability at most $p_2$. Such a hash immediately implies an ANN data structure with query time $\Theta(n^\rho)$ and space $\Theta(n^{1+\rho})$, where $\rho$ is defined as $\rho = \log p_1 / \log p_2$ – see Section 2 for a full exposition.

Locality-sensitive hashes have been developed for many distance functions, including Jaccard similarity [22, 25], Edit Distance [46, 45], Frechet distance [29], and $\chi^2$ distance [33], among many others.

**Space-Efficient ANN Data Structures.** Despite its popularity and applicability, one downside of LSH stands out: an LSH-based data structure requires $\Omega(n^{1+\rho})$ space. This space usage quickly becomes prohibitive. For example, for $\ell_1$, the classic LSH of Sar-Peled, Indyk, and Motwani [38, 35] obtains $\rho = 1/c$, so if $c = 2$ the data structure requires $\Omega(n^{3/2})$ space.

A long line of research has investigated ways to improve the space usage for LSH; much of this work focused on the Euclidean distance [50, 5, 39, 24, 11, 53, 18, 17]. Of particular interest is the near-linear-space regime, where the space required by the data structure is close to $O(nd)$.

Space-efficient LSH approaches are difficult to design. This is because, at a high level, space-efficient ANN methods usually store points based on a space partition much like in an LSH; the data structure saves space by storing the points in fewer locations. The data structure then must probe more locations on each query to ensure correctness. Designing a correct approach along these lines – ensuring that the query probes in the correct locations to guarantee correctness – is technically challenging, and each approach must be carefully tailored to the specific space partition being used (see the discussion of techniques and related work in [39] for example). Perhaps for this reason, many ANN problems have no known space-efficient solutions with theoretical query guarantees.

**Best Known Bounds.** Andoni et al. [11] obtain the current state-of-the-art bounds for space-efficient Euclidean ANN and Manhattan ANN. Their results include a smooth tradeoff between the space usage and query time of the data structure, achieving state-of-the-art bounds along the entire curve.

Interestingly, there is a matching conditional lower bound given in [11]. They define a type of data structure, a *list-of-points data structure*, to encompass "LSH-like" approaches. In short, a list of points data structure explicitly stores lists of points; each query must choose a subset of lists to look through for a point at distance $\leq cr$. Most high-dimensional ANN data structures are list-of-points data structures, and Andoni et al. conjecture that their lower bound can be generalized to handle the few exceptions [12].

The data structures presented in this paper are not list-of-points data structures, as they do not store lists of points explicitly. Instead, our data structures implicitly store lists of points while retaining good query time using a sublinear-space function inversion data structure (defined below). This will allow us to improve their query time, breaking the list of points lower bound.

**Function Inversion.** Our results work by applying function inversion to LSH. Function inversion data structures were initially proposed by Hellman [36], and later analyzed by Fiat and Naor [30]. In short, these data structures allow us to preprocess a function $f : \{1, \ldots, N\} \to \{1, \ldots, N\}$ using $o(N)$ space so that for a given $y \in \{1, \ldots, N\}$, we can find an $x$ with $f(x) = y$ in $o(N)$ time (see Lemma 6 for specifics).

A recent, exciting line of work has looked at how these function inversion data structures can be applied to achieve space- and time-efficient solutions to classic data structure problems. Function inversion can be applied to an online 3SUM variant to give new time-space tradeoffs [41, 31]. Aronov et al. [15] gave results for the closely-related problem of collinearity testing. Bille et al. [21] use the 3SUM method as a black box to give improve string indexing methods. Finally, Aronov et al. [14] recently gave a toolbox for using function inversion for implicit set representations, with a number of applications.

We show that ANN data structures interface particularly well with function inversion. Specifically, in Section 3, we take advantage of the repeated functions of LSH: since we want to invert many functions, we can store extra metadata (to be shared by all functions) to help queries. Meanwhile, in Section 4, we will show how to combine the data structure of Andoni et al. [11], with function inversion to achieve improved Euclidean and Manhattan ANN performance.

## 1.1 Results

Two of the most significant questions remaining in the area of space-efficient ANN are:

1. Is it possible to obtain a black-box method to improve the space usage of an LSH-based ANN data structure?
2. Is it possible to improve the query time for space-efficient Euclidean ANN beyond the lower bound for list-of-points data structures?

Our results give positive answers to these two questions.

First, we show how to use function inversion to improve the space efficiency of any locality-sensitive hashing method.

▶ **Theorem 1.** *For any locality-sensitive hash family $\mathcal{L}$ (where $\mathcal{L}$ has $\rho = \log p_1 / \log p_2$ and evaluation time $T$, and storing a given $\ell \in \mathcal{L}$ requires $O(n^{1-\rho})$ space), approximation ratio $c$, and space-saving parameter $s < \rho$, there exists an ANN data structure with preprocessing time $O(n^{1+\rho})$, expected space $\widetilde{O}(n^{1+\rho-s})$ and expected query time $\widetilde{O}(Tn^{\rho+3s})$.*

This black box method is the first space-efficient method for many ANN problems. Even for the well-studied Euclidean ANN problem, our simple black-box method can be combined with the classic LSH of Andoni and Indyk [6] to give a linear-space data structure with $n^{4/c^2+o(1)}$ query time; this is competitive with, or even improves upon, some previous algorithms (see Section 1.3).

Second, we show how to use function inversion to obtain a data structure that gives improved state-of-the-art query times for near-linear-space Euclidean ANN and Manhattan ANN.

▶ **Theorem 2.** *There exists a Euclidean ANN data structure requiring $n^{1+o(1)}$ space and at most $n^{1.013+o(1)}$ preprocessing time that can answer queries in $n^{\alpha(c)+o(1)}$ expected time with*

$$\alpha(c) = \frac{2c^2-1}{c^4}\left(1 - \frac{(c^2-1)^2}{4c^4+(c^2-1)^2}\right).$$

In short, the idea of our data structure is to take the data structure of Andoni et al. [11] (which we refer to as ALRW), and replace the lists of points with a function-inversion data structure. Unfortunately, ALRW itself – even without the lists – requires far too much space. Thus, achieving our bounds requires trimming the ALRW data structure and carefully running function inversion on the result.

## 1.2 Comparing Results

In this section we compare our results to past work, focusing our comparison specifically on how our query time improves on the state of the art for Euclidean ANN. The previous state of the art query time by Andoni et al. [11]; we refer to it as $n^{ALRW(c)+o(1)}$, with $ALRW(c) = (2c^2-1)/c^4$. In Theorem 2 we achieve query time $n^{\alpha(c)+o(1)}$ with $\alpha(c) = \frac{2c^2-1}{c^4}\left(1 - \frac{(c^2-1)^2}{4c^4+(c^2-1)^2}\right)$.

Since $(c-1)^2$ and $4c^2$ are positive, we immediately have $\alpha(c) < ALRW(c)$. Furthermore, as $c$ gets large, $(c^2-1)^2/(4c^4+(c^2-1)^2)$ approaches $1/5$; more concretely, one can verify that if $c > 2.6$ we have $\alpha(c) < .85ALRW(c)$.

For a more complete picture, we compare the bounds obtained by each in Figure 1 and Table 1. Specifically, this gives a sense of how much improvement is obtained for various values of $c$. We see that for values of $c$ close to 2 we obtain a query time improvement of roughly $n^{.04}$. In Table 1, we also give the exact exponent of the preprocessing time; for large

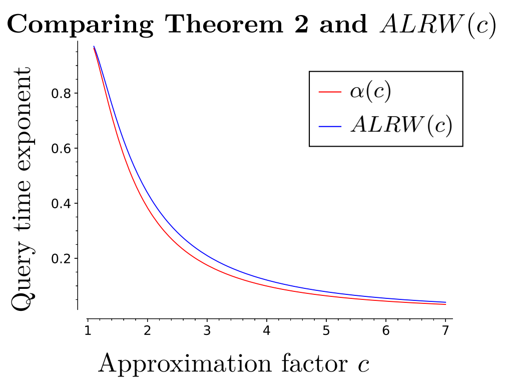


**Figure 1** A figure comparing $\alpha(c)$ from Theorem 2 to $ALRW(c)$. The $y$-axis represents the exponent of the query time: our results obtain a linear-space data structure with query time $n^{\alpha(c)+o(1)}$, compared to the state of the art in [11] with query time $n^{ALRW(c)+o(1)}$.

**Table 1** A table comparing the exponent of the query time of our linear-space approach vs that of [11]. All values are rounded to the third decimal place. In the final column, we give the exponent of the preprocessing time.

| $c$ | $\alpha(c)$ | $ALRW(c)$ | preproc. |
|---|---|---|---|
| 1.05 | .989 | .991 | 1.001 |
| 1.5 | .641 | .691 | 1.011 |
| 1.79 | .471 | .527 | 1.012 |
| 2 | .383 | .438 | 1.012 |
| 3 | .175 | .210 | 1.007 |
| 10 | .016 | .020 | 1.001 |

or small $c$ this is noticeably better than the $n^{1.013}$ upper bound from Theorem 2. While our improved query time comes at a cost of increased preprocessing time (ALRW requires $n^{1+o(1)}$ preprocessing time), this increase is fairly mild. We include an entry at $c = 1.79$ as it maximizes $ALRW(c) - \alpha(c)$.

## 1.3 Related Work

**ANN.** We briefly describe past work on ANN. See the survey by Andoni and Indyk [7] for a more thorough exposition.

As mentioned above, theoretical bounds for ANN without exponential dependance on $d$ usually rely on locality-sensitive hashing. LSH-based methods (including data-dependent results) have seen extensive work on Euclidean ANN in particular [35, 28, 6, 9, 13, 11, 2]. Further work has applied LSH to many other metrics [25, 46, 45, 29, 33, 3]; this includes the well-known (and widely used) Bit Sampling LSH (for Hamming distance) [35] and MinHash (for Jaccard similarity) [22].

In terms of practical performance, LSH-based methods are competitive, though they can be outperformed by heuristic methods on structured data (for a full comparison see e.g. the benchmark of Aumüller et al. [16]). There has been work on giving worst-case bounds for the performance of these heuristics on certain types of datasets [47, 43, 8, 1].

**Space-Efficient ANN.** Theoretical bounds in the low-space regime have focused largely on Euclidean ANN. Panigrahy gave a data structure for Euclidean ANN using $\widetilde{O}(n)$ space, and achieving query time at most $\widetilde{O}(n^{2.09/c})$ (this bound can be improved for small $c$; e.g. if $c = 2$ the query time is roughly $n^{.69}$) [50]. Andoni and Indyk gave $\widetilde{O}(n)$-space data structure with an improved query time of $n^{O(1/c^2)}$, where the constant in the exponent of $n$ is not specified [6]; this is discussed further in Andoni's thesis [5]. Kapralov gave the first data structure that can trade off smoothly between space and time [39]. Setting the parameters to $O(dn)$ space, this data structure requires $n^{4/(c^2+1)+o(1)}$ query time. A sequence of followup papers [42, 24, 10] (see also [18]) culminated in Andoni et al. [11] giving a data structure achieving smooth tradeoffs between time and space for Euclidean and Manhattan distances, along with a matching lower bound for "list-of-points" data structures. Setting the space to be $n^{1+o(1)}$, they achieve query time $n^{(2c^2-1)/c^4+o(1)}$.

**Function Inversion.** The problem of giving smooth time-space tradeoffs for inverting a black-box function was initiated by Hellman [36]. This idea was generalized and improved by Fiat and Naor [30], and recently improved further by Golovnev et al. [32]. There are known lower bounds as well: Hellman's original result is nearly optimal for inverting random functions when limited to a restricted class of algorithms [19], and there are tight upper and lower bounds for inverting permutations [55].

As mentioned earlier, function inversion has recently been applied to data structures, achieving improved tradeoffs for 3SUM [41, 31], as well as the related problems of collinearity testing [15] and string indexing [21]. The results in this paper extend this line of work, applying function inversion to ANN.

## 2 Preliminaries

**Locality-Sensitive Hashing.** We begin by formally defining a locality-sensitive hash family.

▶ **Definition 3.** *A hash family $\mathcal{H}$ is $(p_1, p_2, r, cr)$-sensitive for a distance function $d(\cdot,\cdot)$ if for any points $x$, $y$ with $d(x,y) \leq r$, $\Pr_{h\in\mathcal{H}}[h(x) = h(y)] \geq p_1$, and for any points $x'$, $y'$ with $d(x',y') \geq cr$, $\Pr_{h\in\mathcal{L}}[h(x') = h(y')] \leq p_2$.*

We describe how a locality-sensitive hash can be used for ANN search as was originally described by Indyk and Motwani [38, 35].

To begin, we select a sequence of hash functions. The first step is to create a new LSH by concatenating $\lceil \log_{1/p_2} n \rceil$ independently-chosen functions from $\mathcal{H}$; let $\mathcal{L}$ be the family of possible hash functions resulting from this concatenation. Thus, if $d(x',y') \geq cr$, $\Pr_{\ell\in\mathcal{L}}[\ell(x') = \ell(y')] \leq 1/n$. These parameters are set so that there is one expected point $z$ with $d(z,q) \geq cr$ and $\ell(z) = \ell(q)$.

ANN performance is generally given in terms of $\rho = \log p_1 / \log p_2$. We have for $x$, $y$ with $d(x,y) \leq r$, $\Pr_\ell[\ell(x) = \ell(y)] \geq p_1/n^\rho$. Thus, we repeat the above steps $R = \Theta(n^\rho/p_1)$ times to obtain $R$ hash functions $\ell_1, \ldots, \ell_R$, each sampled independently from $\mathcal{L}$.

Now, preprocessing. To preprocess, create a *reverse lookup table* for $\ell_i$ for all $i \in \{1, \ldots, R\}$. A reverse lookup table is a key-value store (for example, we can implement it using a hash table). For each $y$ such that there exists an $x$ with $\ell_i(x) = y$, the reverse lookup table stores $y$ as the key, and $\{x \mid \ell(x) = y\}$ as the value. Thus, each reverse lookup table takes $\Theta(n)$ space (giving $\Theta(n^{1+\rho}/p_1)$ space in total), and using these tables we can find $\ell_j^{-1}(q)$ for a given $q$ and $i$ in $O(1 + |\ell_i^{-1}(q)|)$ expected time.

To perform a query $q$, we look up $\ell_i^{-1}(q)$ for all $i \in \{1, \ldots, R\}$, using the reverse lookup table for each hash function. We call all points found this way *candidate points*; there are $O(n^\rho/p_1)$ candidate points with distance more than $cr$ from $q$ in expectation. (If we find a point at distance less than $cr$ we return it, so such points only increase the number of candidate points by an additive $O(1)$.)

If $x$ has $d(x,q) \leq r$, then the query fails only if $x$ is not in the candidate points. This occurs with probability $(1 - p_1/n^\rho)^R$; this is at most .1 if the constant when setting $R = \Theta(n^\rho/p_1)$ is sufficiently large.

The space usage of the data structure is $O(nR) = O(n^{1+\rho}/p_1)$ for the reverse lookup tables, plus the space to store $\ell_1, \ldots, \ell_R$. (Usually a particular function $\ell_i$ can be stored in $n^{o(1)}$ space, so this is a lower-order term.) A query takes $O(1)$ expected time for each reverse hash table lookup if $\ell_i$ can be evaluated in constant time, for $O(R) = O(n^\rho/p_1)$ time overall; the query time increases linearly if an $\ell_i$ sampled from $\mathcal{L}$ is slower to evaluate in expectation.

**Data-dependent ANN Solutions.** Locality-sensitive hashing as defined above is *data-independent*: we choose the hashes independent of $S$.

Interestingly, it is possible to obtain improved ANN bounds using *data-dependent* techniques [9]: the point set is partitioned into lists much like LSH, but these partitions are chosen using, in part, properties of the data itself. Thus, generating the data structure requires making random choices that depend not only on $n$, $r$, and $c$, but also properties of the point set $S$ itself. The state of the art ANN bounds for Euclidean space use data-dependent techniques [11].

**Definitions.** Throughout the paper, we assume the dataset $S$ is in some arbitrary order; i.e. $S = x_1, x_2, \ldots, x_n$. We assume without loss of generality that $n$ is a power of 2.

We use $\widetilde{O}(f(n))$ to represent $O(f(n)\text{polylog}n)$. Many of our bounds have an $n^{o(1)}$ term; since $\text{polylog}n = n^{o(1)}$, we generally drop $O$ notation when this term is present. We say that a data structure is *near-linear-space* if it requires space $n^{1+o(1)}$ for a dataset of size $n$.

We use $\circ$ to denote concatenation, and $[N]$ to represent $\{1, \ldots, N\}$. For any list $A$, we use $A[i]$ to denote the $i$th element of $A$ (0-indexed).

For any function $f : X \to Y$, we call $X$ the *domain* and $Y$ the *codomain* of $f$. We define $f^{-1}(y) = \{x \in X \mid f(x) = y\}$.

We say that an event occurs *with high probability* if, for any $C$, it occurs with probability bounded below by $1 - 1/n^C$. (Generally, the event is parameterized by a constant $C_2$, usually hidden in $O$ notation; adjusting $C_2$ according to the desired $C$ can achieve the probability bound – see Corollary 5 for example.)

If an event occurs with high probability then we assume it occurs. (If any with high probability event does not occur, we revert to a trivial data structure that scans $S$ for each query; this increases the expected query cost by $o(1)$.)

**Formulas.** We use the following well-known formulas. First, $\binom{x}{y} \leq (ex/y)^y$. Furthermore, $(1 + 1/n)^n \leq e$, and $(1 - 1/n)^n \leq 1/e$.

We use Chernoff bounds throughout the proofs; we reiterate them here for completeness.

▶ **Lemma 4** ([23, 49]). *Let $X = X_1 + X_2 + \ldots$ be the sum of identical independent 0/1 random variables. Then for any $\delta \geq 0$, $\Pr[X \geq (1 + \delta)] \leq e^{-\delta^2 \operatorname{E}[X]/(2+\delta)}$, and for any $1 \geq \delta \geq 0$, $\Pr[X \leq (1 - \delta)] \leq e^{-\delta^2 \operatorname{E}[X]/2}$.*

The following parameter setting is particularly useful.

▶ **Corollary 5.** *Let $X = X_1 + X_2 + \ldots$ be the sum of identical independent 0/1 random variables with $\operatorname{E}[X] = \Omega(\log n)$. Then $X = \Theta(\operatorname{E}[X])$ with high probability.*

We refer to the following bound as the *union bound* (sometimes called Boole's inequality): for any $k$ events $E_1, \ldots, E_k$, we have $\Pr[E_1 \text{ or } \ldots \text{ or } E_k] \leq \Pr[E_1] + \ldots + \Pr[E_k]$.

Most of the proofs in this paper have been removed for space.

## 3 Function Inversion for LSH

In this section we give our basic function inversion data structure and apply it to LSH.

## 3.1 Function Inversion

The basic building block of this paper is the function inversion data structure of Fiat and Naor [30], which gave theoretical bounds for time-space tradeoffs to invert any function $f : [N] \to [N]$.

▶ **Lemma 6** ([30]). *For any function $f : [N] \to [N]$ that can be evaluated in $T(f)$ time, and any $\sigma > 0$, there exists a data structure that requires $\widetilde{O}(N/\sigma)$ space that can, for any $q$, find an $x$ such that $f(x) = q$ with constant probability in $\widetilde{O}(T(f)\sigma^3)$ time.*

This result was recently generalized by Golovnev, Guo, Peters, and Stephens-Davidowitz to give improved bounds for large $\sigma$, namely, query time $\widetilde{O}(T(f)\min\{\sigma^3, \sigma\sqrt{N}\})$ [32]. However, their model allows either access to a large random string that does not count toward the space bound, or a large fixed advice string for each $N$. In contrast, Lemma 6 in [30] uses explicit hash functions. It is plausible that the results of [32] would work in such a setting as well; this would immediately improve Theorems 1 and 7 for large $\sigma$.

**Obstacles to Applying Function Inversion to LSH.** Function inversion has immediate potential to improve LSH performance: rather than storing a reverse lookup table to find $\ell_i^{-1}(q)$, we can use function inversion to find $\ell_i^{-1}(q)$ instead. We obtain the same candidate points, so correctness is guaranteed, but we improve the space usage by a factor $\widetilde{\Theta}(\sigma)$. (We describe this strategy in more detail in Section 3.2 below.)

However, Lemma 6 cannot be immediately applied to LSH. The first issue is that the codomain $D$ of an LSH is unlikely to be $[N]$. This can be handled (in short) by hashing the output using a hash function $h : D \to [N]$ from a universal hash family; this technique is standard in the literature (see [14, 26, 31, 41, 32]).

Even after reducing the codomain of the function, Lemma 6 does not suffice for our purposes. When using locality-sensitive hashing, we must compare the query to *all* points in its preimage, whereas Lemma 6 only returns a single point. In the remainder of this subsection, we show in fact we can obtain all $x$ with $f(x) = q$. Note that this result requires an additive $O(N)$ space – it is only useful when we want to invert multiple hash functions over the same set $[N]$.

**The All-Function Inversion Data Structure.** We now give our data structure, the *all-function-inversion data structure*, which generalizes Lemma 6 to handle the above issues. We first describe the data structure, and then prove its performance in Theorem 7.

We describe how we can use sampling to find all points that a function $f : [N] \to D$ maps to a given value, and briefly outline the idea behind why our strategy is correct (the proof of Theorem 7 argues correctness more formally). Along the way, we will handle the case where $f$ has large codomain.

Let's focus on inverting a single $f$ for a query $q$. Assume momentarily that we are given $\kappa = |f^{-1}(q)|$ and that $1 < \kappa < o(N/\log N)$ (if $\kappa = 1$ then Lemma 6 suffices; if $\kappa = \Omega(N/\log N)$ then we can find $f^{-1}(q)$ by applying $f$ to all possible $N = \widetilde{O}(|f^{-1}(q)|)$ elements in the domain). Then we create $\Theta(\kappa \log N)$ sets, each obtained by sampling every element of $[N]$ with probability $1/\kappa$; denote these sets $\mathcal{N}^k$ for $k = 1 \ldots \Theta(\kappa \log N)$.

With high probability, since $N/\kappa = \Omega(\log N)$, each set $\mathcal{N}^k$ has $C_1 N/\kappa$ elements for some constant $C_1$.[1] Let us store all elements of $\mathcal{N}^k$ in an array of size $|\mathcal{N}^k|$ – that way we can find the $i$th element of $\mathcal{N}^k$, denoted $\mathcal{N}^k[i]$, in $O(1)$ time (this requires an additional $\widetilde{O}(N)$

[1] Assume $C_1$ is chosen so that $C_1 N/\kappa$ is an integer.

space). Then we can define a function $f_k : [C_1 N/\kappa] \to [C_1 N/\kappa]$ which simulates the behavior of $f$ on $\mathcal{N}^k$ as $f_k(i) = h_k(f(\mathcal{N}^k[i]))$ where $h_k : D \to [C_1 N/\kappa]$ is from a universal family. We build the data structure from Lemma 6 for each $f_k$.

Call an element $x \in f^{-1}(q)$ a *singleton* for $\mathcal{N}^k$ if $x$ is the only element in $f_k^{-1}(q)$ (see also [41]). By standard Chernoff bounds (Corollary 5), $x$ is a singleton for $\Theta(\log n)$ sets $\mathcal{N}^k$. If $x$ is a singleton, then the data structure from Lemma 6 returns $x$ with constant probability; again by Corollary 5, $x$ is returned for some $\mathcal{N}^k$ with high probability.

We remove the assumption that $\kappa$ is known up front using repeated doubling. We begin with $\kappa = 2$. We run the above algorithm for $f_k$ for $k = 1, \ldots, \kappa \log N$. Specifically, for each $k$, we query for a $j = f_k^{-1}(h_k(f(q)))$, look up $y = \mathcal{N}^k[j]$ in the array, and check that $f(y) = f(q)$. If over all $k$ queries at least $\kappa$ distinct elements $x$ are found with $f(x) = f(q)$, we double $\kappa$ and repeat; otherwise we return all elements found so far as $f^{-1}(q)$.

We call the above data structure the *all-function-inversion data structure*.

▶ **Theorem 7.** *Consider the all-function inversion data structure built with parameter $\sigma$ for a set of $R \le N$ functions $f_1, \ldots f_R$, where $f_i : [N] \to D$ and $f_i$ can be calculated in $T(f)$ time for all $i$. This data structure requires $\widetilde{O}(N + NR/\sigma)$ space, can be built in $\widetilde{O}(NR)$ time, and can find $f_i^{-1}(q)$ for any query $q \in [N]$ and any $f_i$ with high probability. Each query requires*

$$\widetilde{O}\left(T(f)\left(1 + |f_i^{-1}(q)|\right)\sigma^3\right)$$

*expected time.*

Before proving Theorem 7 we prove a useful intermediate lemma that treats the idea of a singleton in more generality. In the above discussion, we used $\kappa$ functions $f_k$ to invert some function $f$; from now on, we refer to these as the $k$ functions $f_{i,k}$ used to invert each function $f_i$.

▶ **Lemma 8.** *For any $q \in [N]$, any $\kappa \ge 2$, and any $k$ (with $f_i$ and $f_{i,k}$ as defined above), for any set $X \subseteq f_i^{-1}(q)$ with $|f_i^{-1}(q) \setminus X| \le 2\kappa$, the probability that some element of $X$ is returned when querying the function inversion data structure for $f_{i,k}$ is $\Omega(1)$ if $|X| > \kappa/2$, and $\Omega(|X|/\kappa)$ if $|X| \le \kappa/2$.*

**Proof of Theorem 7.** We begin with space and preprocessing time: we show that for each $\kappa$, the data structure requires $\widetilde{O}(N/\sigma)$ space and $\widetilde{O}(N)$ preprocessing time with high probability. The function inversion data structure built on each $f_{i,k}$ requires $\widetilde{O}(|\mathcal{N}^k|/\sigma)$ space and $\widetilde{O}(|\mathcal{N}^k|)$ preprocessing time, plus $O(\log N)$ space to store $h_{i,k}$. By Chernoff bounds (Corollary 5), $\sum |\mathcal{N}^k| = O(N \log N)$ with high probability; summing over all $\log N$ values of $\kappa$ gives the bound.

Now, correctness. We split into two claims: first, that if $\kappa > |f_i^{-1}(q)|$, then (across all queries to $f_{i,k}^{-1}$) all elements in $f_i^{-1}(q)$ are found with high probability; second, that if $\kappa \le |f_i^{-1}(q)|$, that $\kappa$ distinct elements from $f^{-1}(q)$ are found (and thus $\kappa$ is doubled and the search continues) with high probability.

First, consider the case where $\kappa$ is the smallest power of 2 satisfying $\kappa > |f^{-1}(q)|$. Since there are fewer than $\kappa$ elements in $f^{-1}(q)$, the all-function-inversion data structure must return after this round; thus we are left to show that all elements of $f^{-1}(q)$ are returned in this round. Fix an $x \in f^{-1}(q)$. By Lemma 8 with $X = \{x\}$ (and thus $|Y| = |f^{-1}(q)| \le 2\kappa$), $x$ is returned by the data structure for $f_{i,k}$ with probability $\Omega(1/\kappa)$. Since we choose a new $\mathcal{N}^k$ and $h_{i,k}$ for each $f_{i,k}$ these events are independent; thus by Corollary 5, over all $\Theta(\kappa \log N)$ values of $k$, one returns $x$ with high probability. Taking a union bound over all $< N$ values of $x$, all are returned with high probability.

Now, consider $\kappa \leq f_i^{-1}(q)$. A technical note on this case: the proof of in Lemma 6 from [30] does not as-is guarantee that a uniform random element is returned from $f^{-1}(q)$. A stronger version of this lemma that did provide such a guarantee would simplify this proof.

Partition the $\Theta(\kappa \log n)$ values of $k$ into *epochs* of $C_2\kappa$ consecutive values. There are $\Theta(\log n)$ epochs. For the $j$th epoch, let $F_j$ be the set of elements in $f^{-1}(q)$ that have been found by the data structure so far, and let $r_j = \kappa - |F_j|$. We say that the $j$th epoch is *successful* if $r_{j+1} \leq r_j/2$. After $\log \kappa + 2 = O(\log N)$ successful epochs, $\kappa$ elements have been found and we are done. We now show that an epoch is successful with constant probability; a Chernoff bound (Corollary 5) over the $\Theta(\log N)$ epochs gives the proof.

Fix an epoch $j$ with $|F_j| < \kappa$; we want to show that it is successful with constant probability. For any $k$ within epoch $j$, let $F$ be the elements found so far; assume that $F$ satisfies $\kappa - |F| > r_j/2$ (i.e. assume the epoch is not yet successful). Note that since $r_j \geq 0$ this implies that $|F| < \kappa$. Let $X \leftarrow f^{-1}(q) \setminus F$; since $|f^{-1}(q)| \geq \kappa$ we have that $|X| \geq \kappa - |F| > r_j/2$. Applying Lemma 8 to $X$ (with $F = f^{-1}(q) \setminus X$; notice $|F| < \kappa - r_j/2 \leq 2\kappa$), we have that a new element is found for a given $f_{i,k}$ with probability $\Omega(r_j/(2\kappa))$. Consider the random variable representing if each $f_{i,k}$ increases $|F|$. These are independent Bernoulli trials (they are independent since we choose a new $h_k$ and build a new function inversion data structure for each $f_{i,k}$); the expected number of elements found in the epoch is $\Omega(r_j)$, and the standard deviation is $\Theta(\sqrt{r_j})$; thus with constant probability the number of elements found is $\Omega(r_j)$ (i.e. is within one standard deviation of the expectation) and the epoch is successful.

The time required is (defining $\ell = \log_2 \kappa$ and $\lg x = \lfloor \log_2 \min\{x, 2\} \rfloor$):

$$\sum_{\ell=1}^{1+\lg |f^{-1}(q)|} \widetilde{O}(2^\ell \log N \sigma^3).$$

Summing gives the desired bound. ◀

## 3.2 Applying Function Inversion to LSH

We reiterate how LSH can help us solve ANN. Consider a locality-sensitive hash family $\mathcal{L}$. To solve ANN using $\mathcal{L}$, classically one selects[2] $R = \Theta(n^\rho)$ hashes $\ell_1 \ldots, \ell_R$ from $\mathcal{L}$. For each such $\ell_i$, one stores a reverse lookup table allowing us to find $x$ given $\ell_i(x)$, for all $x \in S$. This requires $O(n)$ space per lookup table, giving $O(n^{1+\rho})$ space overall. On a query, one iterates through $i \in \{1, \ldots, R\}$, using the lookup table to find all $x \in S$ with $\ell_i(x) = \ell_i(q)$, giving $O(n^\rho)$ expected query time.

Function inversion gives a space-efficient replacement for the lookup table – after all, the point of the lookup table is really to find $\ell_i^{-1}(\ell_i(q))$. We now describe this strategy in more detail.

Consider $R = O(n^\rho)$ hash functions $\ell_1, \ldots, \ell_R$ from a locality-sensitive hash family $\mathcal{L}$. We define a new sequence of functions $\widehat{\ell}_1, \ldots, \widehat{\ell}_R$: for all $i \in [R]$ and $j \in [n]$, let $\widehat{\ell}_i(j) = \ell_i(x_j)$. (Thus, the domain of $\widehat{\ell}_i$ is $[n]$ for all $i$.)

We apply Theorem 7 to invert all functions $\widehat{\ell}_i$ with space savings $\sigma \leftarrow n^s$. The query algorithm follows immediately: for $i \in \{1, \ldots, R\}$, rather than looking up all values of $x$ with $\ell_i(x) = \ell_i(q)$ in the lookup table, we can instead query $\widehat{\ell}^{-1}(\ell(q))$ to obtain the same candidate points, in $\widetilde{O}(n^{3s})$ time per returned candidate point.

[2] From now on we assume $p_1$ is a constant for $\mathcal{L}$ for simplicity; our results easily generalize.

Thus, we store the following: the original set $S$, the hash functions $\ell_1, \dots, \ell_R$, and an all-function-inversion data structure for each of $\widehat{\ell}_1, \dots, \widehat{\ell}_R$. We emphasize that we do not store the reverse lookup table for $\ell_1, \dots, \ell_R$ – we just store enough information to evaluate each function.

With this strategy we obtain Theorem 1.

▶ **Theorem 1.** *For any locality-sensitive hash family $\mathcal{L}$ (where $\mathcal{L}$ has $\rho = \log p_1 / \log p_2$ and evaluation time $T$, and storing a given $\ell \in \mathcal{L}$ requires $O(n^{1-\rho})$ space), approximation ratio $c$, and space-saving parameter $s < \rho$, there exists an ANN data structure with preprocessing time $O(n^{1+\rho})$, expected space $\widetilde{O}(n^{1+\rho-s})$ and expected query time $\widetilde{O}(Tn^{\rho+3s})$.*

**Discussion.** Setting $s = \rho$ gives a near-linear space data structure using any locality-sensitive hash family, with query time $O(Tn^{4\rho})$ where $T$ is the time to evaluate a function from the family. Thus we obtain a black box near-linear space data structure with nontrivial query time for *any* LSH family with $n^{o(1)}$ evaluation time, $O(n^{1-\rho})$ space, and $\rho < 1/4$. We point out that most LSH families we are aware of easily meet these requirements: for example, a hash sampled from the classic Indyk-Motwani LSH family [38] has $O(\log n)$ evaluation time and $O(\log n)$ space (as a hash from their family consists of $O(\log n)$ sampled dimensions).

For Euclidean ANN, Andoni and Indyk achieved an LSH with $\rho = 1/c^2$ and evaluation time $n^{o(1)}$ [6]. Applying Theorem 1 gives a data structure with $n^{1+o(1)}$ space and query time $n^{4/c^2+o(1)}$. This is already competitive with many past space-efficient Euclidean ANN data structures.

That said, the performance of this black box approach is worse than can be obtained by more recent results, e.g. [24, 11]. Improving on [24, 11] requires more sophisticated techniques, which we give in Section 4.

## 4 Near-Linear-Space Euclidean ANN

In this section we use function inversion to improve the performance of the most performant near-linear-space ANN data structure for Euclidean distance. Specifically, we improve performance of the space-efficient data structure for Euclidean ANN given by Andoni et al. in [11]; we call this data structure ALRW.

First, let us give some intuition for this improvement. ALRW is able to achieve space $n^{1+\rho_u+o(1)}$ and expected query time $n^{\rho_q+o(1)}$ for any $\rho_u, \rho_q$ satisfying

$$c^2\sqrt{\rho_q} + (c^2-1)\sqrt{\rho_u} \geq \sqrt{2c^2-1}. \tag{1}$$

We can set $\rho_u = 0$ and obtain that linear-space ALRW achieves query time $n^{ALRW(c)+o(1)}$ with $ALRW(c) = \rho_q = (2c^2-1)/c^4$. Note that this is significantly better than we would get by applying Theorem 1 to these bounds directly: Equation 1 allows $\rho_q = \rho_u = 1/(2c^2-1)$, after which applying function inversion to obtain near-linear space would give query time $n^{4/(2c^2-1)+o(1)}$.

The key idea behind our improvement is to note that the query time of ALRW begins increasing quickly as $\rho_u$ approaches 0. For example, let's fix $c = 2$. Then near-linear-space ALRW achieves query time $\approx n^{.44}$. If we increase $\rho_u$ just slightly, the query time drops: setting $\rho_u = .01$ in Equation 1, we obtain $\rho_q \approx .34$ (and thus query time $\approx n^{.34}$).

This leaves room for function inversion to help: consider instantiating ALRW with $\rho_q \approx .34$ and $\rho_u = .01$, with the plan to reduce the space to near-linear by applying Theorem 1 with $s = \rho_u$. If this worked, we would incur total query time $n^{\rho_q+4s} = n^{.38} \ll n^{.4375}$.

In the rest of this section we give a more thorough exposition of these ideas. In particular, there are several obstacles we must handle. First, ALRW is not an LSH – it consists of a tree with a list of points at each leaf. Therefore, Theorem 1 does not directly apply. Moreover, the tree alone in ALRW requires $n^{1+\rho_u+o(1)}$ space: thus, we do not even have enough space to store the internals of ALRW, even if we use function inversion to efficiently store the lists of points. We must describe how to reduce the space usage of ALRW, as well as how function inversion can be applied to a tree rather than a hash to recover the lists of points at the leaves. Finally, we must set parameters: we want to find the value of $\rho_u$ to optimize performance.

In the rest of this section we combine function inversion with the ALRW data structure to obtain improved state-of-the-art query times for near-linear-space Euclidean ANN.

Throughout this section we will use $d(\cdot,\cdot)$ to refer to the Euclidean distance (we will extend our results to Manhattan distance using a standard reduction [11, 44]). Our data structure begins with an instance of ALRW and modifies it to achieve our bounds. Throughout this section we use $\rho_u$ to denote the parameter used to construct ALRW. (We will set the optimal $\rho_u$ for a given $c$ in Section 4.3.3.)

## 4.1 The Optimal List-of-Points Data Structure

In this subsection we summarize ALRW. We focus on the aspects of ALRW that are relevant to applying function inversion to their data structure; the reader should reference [12] for a full description and proof of correctness. Note that we use the full version of the paper [12], as opposed to the conference version [11], when referencing details of the algorithm.

ALRW is built using parameters $\rho_u$ and $\rho_q$ which provide the space/query time tradeoff: ALRW requires $n^{1+\rho_u+o(1)}$ space, and queries can be performed in $n^{\rho_q+o(1)}$ time, so long as $\rho_u$ and $\rho_q$ satisfy Equation 1.

### 4.1.1 High-level Description of ALRW

ALRW consists of a single tree. Each leaf of the tree has a pointer to a list of points; each internal node contains metadata to help with queries. During preprocessing, the tree is constructed, along with the list for each leaf of the tree. We say that a leaf $\ell$ *contains* a point $x$ if $x$ is in the list pointed to by $\ell$.

During a query $q$, a subtree of ALRW is traversed, beginning with the root. We say all nodes in this subtree are *traversed by* $q$. For each internal node traversed, in $n^{o(1)}$ time it is possible to find the children of the node that are recursively traversed for $q$. When reaching a leaf, for each point in the list pointed to by the leaf, $d(x, q)$ is calculated. If $d(x, q) \le cr$, then $x$ is returned.

A similar process defines what leaf contains each $x \in S$: a subtree of ALRW is traversed using the metadata stored at each internal node (in $n^{o(1)}$ time per node); each time a leaf is traversed it contains $x$.[3] We say that any node in this subtree is *traversed by* $x$.

[3] In reality, the ALRW tree is built and the lists are created simultaneously using a single recursive process. However, it is useful for our exposition to imagine each point as if it were stored using a separate process after the tree was already constructed.

### 4.1.2 ALRW Details and Parameters

The internal ALRW nodes are of two types: *ball nodes* and *sphere nodes*; these correspond to recursive calls during construction to ProcessBall (for dense clusters) and ProcessSphere (for recursing using LSH) in [12].[4] Any node in the tree has at most $b_b$ ball node children, with $b_b = (\log\log\log n)^{O(1)}$, and at most $b_s$ sphere node children, with $b_s = n^{o(1)}$.

Consider a sphere node $v$ traversed when preprocessing a point $x \in S$. The number of children of $v$ that are also sphere nodes, and the probability that $x$ traverses each child, vary as we traverse the tree – however, they are set so that the expected number of sphere node children that $x$ traverses recursively remains the same. Let $b_u$ be this quantity: the expected number of sphere node children traversed when determining what lists to use to store $x$. By the proof of [12, Claim 4.12], for any sphere node, $b_u = n^{(\rho_u + o(1))/K}$.

The number of sphere nodes on a root-to-leaf path in ALRW is at most $K$ with $K = \Theta(\sqrt{\log n})$. Furthermore, all root-to-leaf paths have at most $K_b = \widetilde{O}(\log\log n)$ ball nodes [12, Lemma 4.2].[5]

In fact, if $\rho_u$ is a positive constant that does not depend on $n$, the number of children traversed by any $x$ is very close to $b_u$ with high probability. (This result does not appear explicitly in [12], but can be obtained by applying Chernoff bounds to their analysis.)

▶ **Lemma 9.** *Consider an ALRW data structure with parameter $\rho_u > 0$ with $\rho_u = \Omega(1)$. For any sphere node $v$ traversed when considering a point $x$, the number of sphere node children of $v$ also traversed when considering $x$ is at most $b_u(1 + 1/K)$ with high probability.*

The candidate points for a given $q$ consist of the points contained by all leaves traversed by $q$. Thus, the following lemma immediately gives correctness.

▶ **Lemma 10** ([12], Lemma 4.9)**.** *For any query $q$, let $x \in S$ be a point with $d(q, x) \leq r$. Then with probability $> .9$, some leaf $\ell$ of ALRW traversed by $q$ is traversed by $x$.*

## 4.2 Making the Tree Space-Efficient

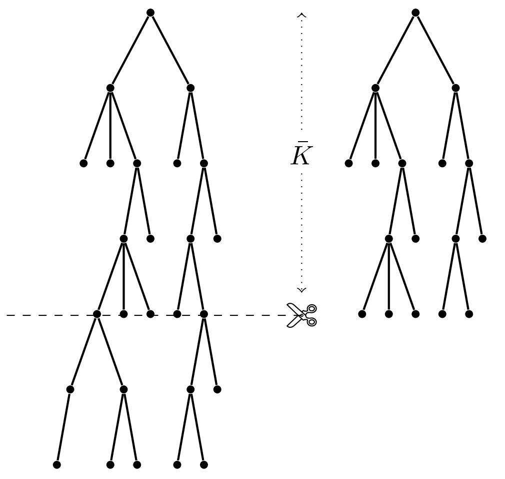
ALRW requires $n^{1+\rho_u(c)+o(1)}$ space, even if we ignore the lists of points at each leaf (i.e., there are $\Omega(n^{1+\rho_u(c)})$ nodes in the tree). Furthermore, this high space usage appears to be integral to the data structure: since the tree is recursively data-dependent, each tree node stores extra information about the data being stored. In this section we describe how to reduce the space of ALRW while still retaining enough data to answer queries efficiently.

We create a *truncated tree* which is a subtree of ALRW. The main idea is as follows. ALRW consists of $n^{1+\rho_u+o(1)}$ leaves, each of which contains $O(1)$ points on average. The goal of the truncated tree is to have $n^{1+o(1)}$ leaves, each with $O(n^{\rho_u})$ points on average. The truncated tree is created carefully to satisfy some slightly stronger properties: we also want to bound the number of sphere nodes along the path from the root to any leaf in the tree (for e.g. the proof of Lemma 12); furthermore, we want to be sure that any leaf traversed by a query contains $O(n^{\rho_u})$ expected points (for e.g the proof of Lemma 13).

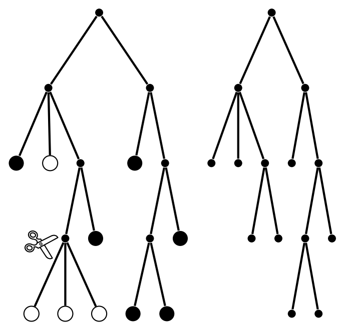
We create the truncated tree in two steps. First, let $\overline{K} = K/(1 + \rho_u)$ and consider the largest subtree of ALRW such that there are at most $\overline{K}$ sphere nodes on any root-to-leaf path of the truncated tree. Call this subtree the *middle step tree*; see Figure 2a.

[4] These two types of nodes are the reason why ALRW is data-dependent; they are integral to the bounds achieved by ALRW.

[5] Technically we are using $K$ slightly differently than in [12] – they use $K$ to bound the number of edges along a root-to-leaf path where both parent and child are sphere nodes; since $K_b = o(K)$ this does not affect our analysis.



**(a)** We begin with ALRW (on the left), and remove all nodes with with more than $\bar{K}$ ancestor sphere nodes to obtain the middle step tree (on the right).



**(b)** Recursively removing small leaves (outlined) of the middle step tree (on the left) gives the truncated tree (on the right).

**Figure 2** Creating the truncated tree in two steps.

We can define lists for each leaf of the middle step tree by running the recursive process from ALRW for each $x \in S$. (Since this process is recursive, the points in the list of in a middle step tree leaf are exactly the points in its descendant leaves in ALRW.)

Now, we trim the middle step tree to obtain the truncated tree. We proceed bottom-up through the middle step tree, starting with the second-to-last level. Call a leaf of the middle step tree *large* if it contains at least $n^{\rho_u/(1+\rho_u)}$ points and *small* otherwise. If all children of a node $v$ are small leaves, we remove all children of $v$ (so $v$ is now a leaf), repeating until no nodes have only small leaves as children. We call this structure the *truncated tree.*

The following lemma shows that the truncated tree achieves near-linear space.

▶ **Lemma 11.** *The truncated tree contains $n^{1+o(1)}$ nodes in expectation, each of which requires $n^{o(1)}$ space.*

Our data structure stores a distinct label (e.g. from 1 to $n^{1+o(1)}$) for each node of the truncated tree. We occasionally refer to a leaf using its label (if $\ell$ is a label we may say "leaf $\ell$"). We retain the definition that a leaf $\ell$ *contains* $x \in S$ in the truncated tree if $x$ traverses $\ell$, even though we do not store the actual lists.

Note that Lemma 10 immediately applies to the truncated tree: if $d(x, q) \leq cr$, then with probability $\geq .9$, $x$ is contained in a leaf of the truncated tree traversed by $q$.

## 4.3 Applying Function Inversion

### 4.3.1 Truncated tree functions

The job of the truncated tree functions is to recover the points contained by each leaf in the truncated tree. Specifically if $x_j$ is contained in some leaf with label $\ell$, we want there to be a truncated tree function $\tau$ such that $\tau(j) = \ell$; thus, $\tau^{-1}(\ell) = j$.

The challenge is that each $x \in S$ maps to many points in the truncated tree, whereas each function must output a single leaf. Fortunately, we have shown that the way $x$ traverses the tree is highly regular: if a node $v$ is traversed by $x$, with high probability, the number of children of $v$ traversed by $x$ is at most $b_u(1 + 1/K)$. This allows us to efficiently iterate over the leaves traversed by $x$.

A *tree route* $I$ is a vector of length $\bar{K} + K_b$ where each entry in $I$ is a number between 1 and $b_u(1 + 1/K) + b_b$. Let $R = (b_u(1 + 1/K) + b_b)^{\bar{K}+K_b}$ be the number of tree routes.

▶ **Lemma 12.** *The number of possible tree routes is $R = n^{\rho_u/(1+\rho_u)+o(1)}$.*

For all possible tree routes $I$, we define a function $\tau_I(j)$ using the following process. We begin with vertex $v$ equal to the root of the truncated tree. In step $i$ (for $i$ from 1 to $\bar{K}$), we set $v$ equal to the $I[i]$th child of $v$ traversed by $x_j$. (There are at most $b_u(1 + 1/K) + b_b$ such children by Lemma 9 and definition of $b_b$.) If at any point there are less than $I[i]$ children of $v$ traversed by $x_j$, then $\tau_I(j) = -1$. If $v$ is a leaf with label $\ell$, then $\tau_I(j) = \ell$.

We immediately obtain the following. First, if $x$ is contained in $\ell$, then there is a tree route $I$ with $\tau_I(j) = \ell$. Second, for any $j \in [N]$ and any tree route $I$, we can calculate $\tau_I(j)$ in $n^{o(1)}$ time. (This follows from the definition of ALRW: each node has $n^{o(1)}$ total children which we consider one by one, and for each child we can determine if $x$ traverses that child in $n^{o(1)}$ time; multiplying by $K$ retains $n^{o(1)}$ time.)

### 4.3.2 The Data Structure

First, preprocessing. We begin by creating the truncated tree. Then, for all $R$ possible tree routes $I$, we build the all-function-inversion data structure (Theorem 7) on $\tau_I$ with $\sigma = R$. This data structure requires $n^{1+o(1)}$ space (noting that the domain of all $\tau_I$ is $[n]$).

To query, we use the function inversion data structure to recover the contents of all leaves traversed by $q$ in the truncated tree as follows. We calculate the labels of leaves of the truncated tree traversed by $q$; call this set of labels $L$. For each $\ell \in L$ and each tree route $I$, we query $\tau_I^{-1}(\ell)$. We compare $q$ to each point returned; if any has distance at most $cr$ from $q$, it is returned. If no such point is found, we return that there is no close point.

### 4.3.3 Analysis

Correctness follows immediately from ALRW: if $d(q, x) \le r$, then with probability $\ge .9$, $x$ is in a leaf traversed by $q$ in the truncated tree. The all-function-inversion data structure will return this leaf with high probability.

Now we give performance. The following is the key performance lemma, bounding the number of leaves traversed by the query in expectation, as well as the total number of points at distance $> cr$ they contain.

▶ **Lemma 13.** *The number of leaves of the truncated tree traversed by $q$ in expectation is at most $n^{\rho_q/(1+\rho_u)+o(1)}$. The expected number of points $x$ satisfying: (1) $x$ is contained in a leaf traversed by $q$, and (2) $d(x, q) > cr$, is at most $n^{\rho_u/(1+\rho_u)+o(1)}$.*

We can now give the final bounds.

▶ **Lemma 14.** *The above data structure requires $n^{1+o(1)}$ space, preprocessing time $nR = n^{1+\rho_u/(1+\rho_u)+o(1)}$, and expected query time $n^{(\rho_q+4\rho_u)/(1+\rho_u)}$.*

**Proof.** The truncated tree requires a total of $n^{1+o(1)}$ space. The time to build the middle step tree and truncated tree is $n^{1+\rho_u/(1+\rho_u)+o(1)}$. (We cannot build all of ALRW and truncate it in this time – instead, we must build the middle step tree recursively as in [12] with maximum depth $\bar{K}$.) Finally, preprocessing the all-function-inversion data structure requires $\widetilde{O}(nR)$ time by Theorem 7.

Recalling that a node can be traversed in $n^{o(1)}$ time, we can find the set of leaves $L$ traversed by $q$ in $n^{\rho_q/(1+\rho_u)+o(1)}$ time by Lemma 13.

Now, we bound the time to complete the function inversion queries. By Theorem 7, for any $I$ and any leaf $\ell$ of the conceptual tree, we can find $\tau_i^{-1}(\ell)$ in $\widetilde{O}(R^3(|\tau_I^{-1}(\ell)| + 1))$ expected time.

Summing over all $\ell$ and $I$, the query time is $\widetilde{O}\left(R^4|L| + R^3 \sum_{\ell \in L,I} |\tau_I^{-1}(\ell)|\right)$. Note that when we find a point at distance $\leq cr$ we return it; thus we need only include points at distance $> cr$ in each $\tau_I^{-1}(\ell)$ term.

By Lemma 12, $R^3|L| = n^{(\rho_q+3\rho_u)/(1+\rho_u)+o(1)}$. By Lemma 13,

$$\mathrm{E}\left[\sum_{\ell \in L,I} |\tau_I^{-1}(\ell)|\right] = n^{(\rho_u+\rho_q)/(1+\rho_u)+o(1)}$$

Summing we obtain the lemma. ◀

Now, we set $\rho_u$ to optimize the above results. Note that the following bounds are somewhat loose. For query time, we ignore the $1 + \rho_u$ term in the denominator of the exponent. For preprocessing time, we give the worst case for any $c$; if $c$ is small or large the bound is fairly pessimistic. That said, both of these losses have, ultimately, a modest effect on the running time since we will choose small values for $\rho_u$.

We use $\alpha(c)$ to bound our running time (rather than the more traditional $\rho$) to prevent confusion with $\rho_u$ and $\rho_q$ used by ALRW, and $\rho$ used in Theorem 1.

► **Theorem 2.** *There exists a Euclidean ANN data structure requiring $n^{1+o(1)}$ space and at most $n^{1.013+o(1)}$ preprocessing time that can answer queries in $n^{\alpha(c)+o(1)}$ expected time with*

$$\alpha(c) = \frac{2c^2-1}{c^4}\left(1 - \frac{(c^2-1)^2}{4c^4+(c^2-1)^2}\right).$$

**Manhattan ANN.** As mentioned in Section 1, we can obtain bounds for Manhattan ANN by a classic embedding; see [44]. This immediately gives a Manhattan ANN data structure with query time $n^{\alpha_M(c)+o(1)}$ with

$$\alpha_M(c) = \frac{2c-1}{c^2}\left(1 - \frac{(c-1)^2}{4c^2+(c-1)^2}\right).$$

The preprocessing time remains bounded above by $n^{1.013+o(1)}$. The proof is essentially identical to that of Theorem 2.

## References

1 Thomas D Ahle, Martin Aumüller, and Rasmus Pagh. Parameter-free locality sensitive hashing for spherical range reporting. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 239–256. SIAM, 2017.

2 Thomas Dybdahl Ahle. Optimal las vegas locality sensitive data structures. In *2017 IEEE 58th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 938–949. IEEE, 2017.

3 Thomas Dybdahl Ahle, Rasmus Pagh, Ilya Razenshteyn, and Francesco Silvestri. On the complexity of inner product similarity join. In *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, pages 151–164, 2016.

4 Josh Alman and Ryan Williams. Probabilistic polynomials and hamming nearest neighbors. In *2015 IEEE 56th Annual Symposium on Foundations of Computer Science*, pages 136–150. IEEE, 2015.

5 Alexandr Andoni. *Nearest neighbor search: the old, the new, and the impossible*. PhD thesis, Massachusetts Institute of Technology, 2009.

6 Alexandr Andoni and Piotr Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. In *47th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 459–468. IEEE, 2006.

7 Alexandr Andoni and Piotr Indyk. Nearest neighbors in high-dimensional spaces. In *Handbook of Discrete and Computational Geometry*, pages 1135–1155. Chapman and Hall/CRC, 2017.

8 Alexandr Andoni, Piotr Indyk, Thijs Laarhoven, Ilya Razenshteyn, and Ludwig Schmidt. Practical and optimal lsh for angular distance. *Advances in neural information processing systems*, 28, 2015.

9 Alexandr Andoni, Piotr Indyk, Huy L Nguyen, and Ilya Razenshteyn. Beyond locality-sensitive hashing. In *Proceedings of the twenty-fifth annual ACM-SIAM symposium on Discrete algorithms*, pages 1018–1028. SIAM, 2014.

10 Alexandr Andoni, Thijs Laarhoven, Ilya Razenshteyn, and Erik Waingarten. Lower bounds on time-space trade-offs for approximate near neighbors. *arXiv preprint arXiv:1605.02701*, 2016.

11 Alexandr Andoni, Thijs Laarhoven, Ilya Razenshteyn, and Erik Waingarten. Optimal hashing-based time-space trade-offs for approximate near neighbors. In *Proceedings of the twenty-eighth annual ACM-SIAM symposium on discrete algorithms*, pages 47–66. SIAM, 2017.

12 Alexandr Andoni, Thijs Laarhoven, Ilya P. Razenshteyn, and Erik Waingarten. Optimal hashing-based time-space trade-offs for approximate near neighbors. *CoRR*, 2016. `arXiv:1608.03580`.

13 Alexandr Andoni and Ilya Razenshteyn. Optimal data-dependent hashing for approximate near neighbors. In *Proceedings of the forty-seventh annual ACM symposium on Theory of computing*, pages 793–801, 2015.

14 Boris Aronov, Jean Cardinal, Justin Dallant, and John Iacono. A general technique for searching in implicit sets via function inversion. *arXiv preprint arXiv:2311.12471*, 2023.

15 Boris Aronov, Esther Ezra, Micha Sharir, and Guy Zigdon. Time and space efficient collinearity indexing. *Computational Geometry*, 110:101963, 2023.

16 Martin Aumüller, Erik Bernhardsson, and Alexander Faithfull. Ann-benchmarks: A benchmarking tool for approximate nearest neighbor algorithms. *Information Systems*, 87:101374, 2020.

17 Martin Aumuller, Sariel Har-Peled, Sepideh Mahabadi, Rasmus Pagh, and Francesco Silvestri. Fair near neighbor search via sampling. *ACM SIGMOD Record*, 50(1):42–49, 2021.

18 Martin Aumüller, Rasmus Pagh, and Francesco Silvestri. Fair near neighbor search: Independent range sampling in high dimensions. In *Proceedings of the 39th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, pages 191–204, 2020.

19 Elad Barkan, Eli Biham, and Adi Shamir. Rigorous bounds on cryptanalytic time/memory tradeoffs. In *Annual International Cryptology Conference*, pages 1–21. Springer, 2006.

20 Pavel Berkhin. A survey of clustering data mining techniques. In *Grouping multidimensional data: Recent advances in clustering*, pages 25–71. Springer, 2006.

21 Philip Bille, Inge Li Gørtz, Moshe Lewenstein, Solon P Pissis, Eva Rotenberg, and Teresa Anna Steiner. Gapped string indexing in subquadratic space and sublinear query time. *arXiv preprint arXiv:2211.16860*, 2022.

22 Andrei Z Broder. On the resemblance and containment of documents. In *Proceedings. Compression and Complexity of SEQUENCES 1997 (Cat. No. 97TB100171)*, pages 21–29. IEEE, 1997.

23 Herman Chernoff. A measure of asymptotic efficiency for tests of a hypothesis based on the sum of observations. *The Annals of Mathematical Statistics*, pages 493–507, 1952.

24 Tobias Christiani. A framework for similarity search with space-time tradeoffs using locality-sensitive filtering. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 31–46. SIAM, 2017.

25 Tobias Christiani and Rasmus Pagh. Set similarity search beyond minhash. In *Proceedings of the 49th annual ACM SIGACT symposium on theory of computing*, pages 1094–1107, 2017.

26 Henry Corrigan-Gibbs and Dmitry Kogan. The function-inversion problem: Barriers and opportunities. In *Theory of Cryptography Conference*, pages 393–421. Springer, 2019.

27 Thomas Cover and Peter Hart. Nearest neighbor pattern classification. *IEEE transactions on information theory*, 13(1):21–27, 1967.

28 Mayur Datar, Nicole Immorlica, Piotr Indyk, and Vahab S Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *Proceedings of the twentieth annual symposium on Computational geometry*, pages 253–262, 2004.

29 Anne Driemel and Francesco Silvestri. Locality-sensitive hashing of curves. In *33rd International Symposium on Computational Geometry (SoCG 2017)*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2017.

30 Amos Fiat and Moni Naor. Rigorous time/space trade-offs for inverting functions. *SIAM Journal on Computing*, 29(3):790–803, 2000.

31 Alexander Golovnev, Siyao Guo, Thibaut Horel, Sunoo Park, and Vinod Vaikuntanathan. Data structures meet cryptography: 3sum with preprocessing. In *Proceedings of the 52nd annual ACM SIGACT symposium on theory of computing*, pages 294–307, 2020.

32 Alexander Golovnev, Siyao Guo, Spencer Peters, and Noah Stephens-Davidowitz. Revisiting time-space tradeoffs for function inversion. In *Annual International Cryptology Conference*, pages 453–481. Springer, 2023.

33 David Gorisse, Matthieu Cord, and Frederic Precioso. Locality-sensitive hashing for chi2 distance. *IEEE transactions on pattern analysis and machine intelligence*, 34(2):402–409, 2011.

34 Christian Hachenberg and Thomas Gottron. Locality sensitive hashing for scalable structural classification and clustering of web documents. In *Proceedings of the 22nd ACM international conference on Information & Knowledge Management*, pages 359–368, 2013.

35 Sariel Har-Peled, Piotr Indyk, and Rajeev Motwani. Approximate nearest neighbor: Towards removing the curse of dimensionality. *Theory OF Computing*, 8:321–350, 2012.

36 Martin Hellman. A cryptanalytic time-memory trade-off. *IEEE transactions on Information Theory*, 26(4):401–406, 1980.

37 Piotr Indyk. *High-dimensional computational geometry*. PhD thesis, Stanford University, 2001.

38 Piotr Indyk and Rajeev Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, pages 604–613, 1998.

39 Michael Kapralov. Smooth tradeoffs between insert and query complexity in nearest neighbor search. In *Proceedings of the 34th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, pages 329–342, 2015.

40 Kifayat Ullah Khan, Batjargal Dolgorsuren, Tu Nguyen Anh, Waqas Nawaz, and Young-Koo Lee. Faster compression methods for a weighted graph using locality sensitive hashing. *Information Sciences*, 421:237–253, 2017.

41 Tsvi Kopelowitz and Ely Porat. The strong 3sum-indexing conjecture is false. *arXiv preprint arXiv:1907.11206*, 2019.

42 Thijs Laarhoven. Tradeoffs for nearest neighbors on the sphere. *arXiv preprint arXiv:1511.07527*, 2015.

43 Thijs Laarhoven. Graph-based time-space trade-offs for approximate near neighbors. In *34th International Symposium on Computational Geometry (SoCG 2018)*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2018.

44 Nathan Linial, Eran London, and Yuri Rabinovich. The geometry of graphs and some of its algorithmic applications. *Combinatorica*, 15:215–245, 1995.

45 Guillaume Marçais, Dan DeBlasio, Prashant Pandey, and Carl Kingsford. Locality-sensitive hashing for the edit distance. *Bioinformatics*, 35(14):i127–i135, 2019.

46 Samuel McCauley. Approximate similarity search under edit distance using locality-sensitive hashing. In *24th International Conference on Database Theory*, 2021.

47 Samuel McCauley, Jesper W Mikkelsen, and Rasmus Pagh. Set similarity search for skewed data. In *Proceedings of the 37th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, pages 63–74, 2018.

48 Marvin Minsky and Seymour A. Papert. *Perceptrons: An Introduction to Computational Geometry*. MIT Press, Cambridge, 1969.

49 Michael Mitzenmacher and Eli Upfal. *Probability and computing: Randomization and probabilistic techniques in algorithms and data analysis*. Cambridge university press, 2017.

50 Rina Panigrahy. Entropy based nearest neighbor search in high dimensions. In *Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithm*, pages 1186–1195, 2006.

51 Aviad Rubinstein. Hardness of approximate nearest neighbor search. In *Proceedings of the 50th annual ACM SIGACT symposium on theory of computing*, pages 1260–1268, 2018.

52 Gerard Salton, Anita Wong, and Chung-Shu Yang. A vector space model for automatic indexing. *Communications of the ACM*, 18(11):613–620, 1975.

53 Francisco Santoyo, Edgar Chavez, and Eric S Tellez. Compressing locality sensitive hashing tables. In *2013 Mexican International Conference on Computer Science*, pages 41–46. IEEE, 2013.

54 Ryan Williams. On the difference between closest, furthest, and orthogonal pairs: Nearly-linear vs barely-subquadratic complexity. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1207–1215. SIAM, 2018.

55 Andrew Chi-Chih Yao. Coherent functions and program checkers. In *Proceedings of the twenty-second annual ACM symposium on Theory of computing*, pages 84–94, 1990.