Finding Nearest Neighbors in Growth-restricted Metrics

David R. Karger Matthias Ruhl

MIT Laboratory for Computer Science
Cambridge, MA 02139, USA

{karger,ruhl}@theory.lcs.mit.edu

ABSTRACT

Most research on nearest neighbor algorithms in the literature has been focused on the Euclidean case. In many practical search problems however, the underlying metric is non-Euclidean. Nearest neighbor algorithms for general metric spaces are quite weak, which motivates a search for other classes of metric spaces that can be tractably searched.

In this paper, we develop an efficient dynamic data structure for nearest neighbor queries in *growth-constrained* metrics. These metrics satisfy the property that for any point q and distance d the number of points within distance $2d$ of q is at most a constant factor larger than the number of points within distance d. Spaces of this kind may occur in networking applications, such as the Internet or Peer-to-peer networks, and vector quantization applications, where feature vectors fall into low-dimensional manifolds within high-dimensional vector spaces.

1. INTRODUCTION

Finding the nearest neighbor of a point in a given metric is a classic algorithmic problem with many practical applications. Some such applications are database queries, in particular for complex data such as multimedia data or biological structures, e.g. on protein structures or genome data. Other uses are in lossy data compression, where data can be encoded by the closest representative from a fixed set of representatives. The common characteristic of these examples is that comparing two elements is costly, so one would like to develop data structures that allow for nearest neighbor searching with a small number of comparisons.

In the formal setting, one is given a metric space $\mathcal{M} = (M,d)$ (where d is symmetric and satisfies the triangle-inequality), and a subset $S \subseteq M$ of n points in the space. Allowing for some preprocessing one wants to efficiently answer queries of two kinds:

(i) Nearest Neighbor: Given a point $q \in M$, return the point in S that is closest to q among all points in S.

(ii) Range Query: Given a point $q \in M$ and $r \geq 0$, return all points $p \in S$ that satisfy $d(p,q) \leq r$.

It is also desirable that the data structure support efficient insertion and deletion of points from S.

STOC'02, May 19-21, 2002, Montreal, Quebec, Canada.

These problems are quite hard for general metrics, evidenced by the fact that known data structures perform poorly (requiring time $\Omega(n^{1-\delta})$) for range queries with non-trivial r, or for nearest neighbor searches where the query point is relatively far from its nearest neighbor (see "related work" below). This seemingly is because general metrics do not provide enough structure to solve these problems efficiently.

In the instances where search problems arise in practice, however, the underlying metric usually is far from general, but satisfies additional constraints. Developing efficient search data structures for these metric spaces is an important problem.

Previous research on this problem has focused on the Euclidean case, i.e. $\mathcal{M} = (\mathbb{R}^d, L_p)$, which is particularly important since many practical applications deal with "feature vectors" that are naturally embedded in Euclidean space. A large number of data structures have been developed that perform very well (with logarithmic time per operation) in *low dimensional* Euclidean spaces. There is, however, a significant number of problems where the data cannot easily be embedded into low-dimensional Euclidean space, or such an embedding results in the loss of information.

In this paper, we are concerned with sample sets S that have a certain smooth-growth property. Throughout this paper, we let $B_p(r) := \{s \in S \mid d(p,s) \leq r\}$ be the ball of radius r around p in S.

Definition 1 (Expansion Rate)
We say that S has (ρ, c)-expansion *iff for all $p \in M$ and $r > 0$,*

$$|B_p(r)| \geq \rho \implies |B_p(2r)| \leq c \cdot |B_p(r)|.$$

In the bulk of this paper, we will set $\rho = O(\log|S|)$ and refer to c as the expansion rate *of S.* $\square$

Intuitively, for any space satisfying this property, points from S "come into view" at a constant rate when we expand a ball around any point $p \in M$.

The factor 2 in the expansion definition can be replaced by any other constant with a corresponding change in c. For intuition, consider the set of points in a uniform d-dimensional grid under the L_1 metric. Balls in this metric are d-dimensional hypercubes. Multiplying the ball radius by 2 corresponds to increasing each side length by this amount, which increases the volume of the cube, and thus the number of points in it, by 2^d. Thus, the L_1 metric on the grid has $(1, 2^d)$-expansion.

Based on this grid intuition, we can consider the expansion rate to be a kind of "dimensionality" measurement for our metric space. Expansion rate is incomparable to standard dimension, however: a balanced binary tree, which can be embedded in 2 dimensions, has a huge expansion rate. A low dimensional manifold in a high dimensional space can have a very low expansion rate (as in the applications discussed below). Under standard dimensionality, a subset of points has no higher dimension than its containing set; this powerful fact is not true for the expansion rate. However, we will see below that a weaker and still useful result does hold: a *random* subset of points from a low-expansion metric space has low expansion. Thus, for example, a collection of points randomly distributed in a d-dimensional Euclidean cube has expansion rate $O(d)$.

The expansion property, applied recursively, shows that the number of points in a ball of radius r is at most polynomial in r. The converse is not true, however: the expansion property requires this growth to be reasonably smooth – it rules out the possibility (consistent with the polynomial bound) that as the ball grows, we encounter a few points, then a long period with no points, then suddenly a tremendous number of points.

Our result. Our main result is a randomized data structure, the *metric skip list*, that allow for nearest neighbor queries in spaces with constant expansion rates ($c = O(1)$). The data structure can answer nearest neighbor queries in $O(\log n)$ time, and range queries in time $O(\log n + k)$ (where k is the number of returned elements) with high probability. The data structure can be constructed in $O(n \log n)$ time, uses $O(n \log n)$ space and allows for the addition and deletion of points in expected $O(\log n \log \log n)$ time. The assumption that c is constant simplifies notation; more generally, the running time is logarithmic in n, but polynomial in the expansion rate c.[1] The data structure is Las Vegas, i.e. always returns the correct result regardless of our random choice. The structure works even if the metric space does not have a bounded expansion rate, although its running time bounds degrade.

Our data structure is quite simple, deducing from the low expansion property that a random sample of a few points in a given ball around the query point is likely to yield a point inside a much closer ball.

Applications. Clearly, the main motivation for looking at spaces with low expansion rates is because they actually appear in real problems. We are aware of at least two applications where this is the case.

In Internet applications, it is often important for nodes to find other nodes that are "near" each other with respect to distance as measured by latency or bandwidth. This paper was motivated by work on the Chord system [7] which provides routing infrastructure for various peer-to-peer applications. In many such applications, it is useful for clients of the system to find a nearest Chord node that can proxy for them in the application. In data caching applications, it is useful to solve the more general problem of finding a nearby node that actually has a copy of the desired data. Plaxton et al. [6] tackled this problem. They describe a distributed system that stores replicated copies of a data item, and a protocol that lets any node retrieve a "nearby" copy of the data item – more precisely, their randomized scheme finds a copy whose expected distance is close to that of the nearest copy. Their scheme makes the same "constant expansion" assumption as we make here. In fact, they require more: that ratio of points in the larger ball to that in the smaller ball must be upper *and* lower bounded by constants exceeding one. We require only the upper bound. In an additional improvement, when data is replicated by the Chord protocol, our scheme can be used to find the closest (rather than just close in expectation) copy of a data item.

[1] Recall that on a grid the expansion rate c is exponential in the standard dimension d; thus being polynomial in c fits the outcome, common in geometric algorithms and data structures, of being exponential in the standard dimension.

A second application comes from machine learning. A current thread of machine learning research [9, 8] postulates that the feature vectors representing points being analyzed form a low-dimensional manifold of a high dimensional space. There is no a-priori specification of the manifold; rather, a large set of example points is provided. The distance metric on these points is given in the high dimensional space. Identifying near neighbors among the example points is useful – for example, to implement the standard k-nearest neighbors algorithm for classification, or to identify neighborhoods in the manifold in order to construct local parameterizations of the manifold. Under assumptions of limited curvature (which are also made in the AI literature) and random selection of example points from the manifold, the low-expansion property will hold (as it does for low dimensional Euclidean spaces) and our near-neighbor structure could be applied.

Related Work. As mentioned previously, most research on nearest neighbor search has focused on the case of vector spaces and/or Euclidean metrics [1, 2]. There has been a growing interest in general metrics, however. In a recent survey [4], Chávez et al. give an overview on the data structures developed for these applications. The most frequent approach is by "pivoting" [10, 11], i.e. the space is partitioned into two halves by picking a random pivot, and putting points into either half of the partition according to their distance to the pivot element. Variations use multiple pivoting elements per split [3]. While these structures answer queries in $O(\log n)$ time for a point that is actually in the set S, they cannot be used efficiently to find nearest neighbors, or perform range queries unless the radii involved are very small. This is because in general the search ranges can split at every pivoting step, requiring the exploration of a significant part of the search tree. (Comparable to the performance guarantee of nearest neighbor searches using quad-trees, which is $O(\sqrt{n})$.) Also, dynamic maintenance of the trees, in particular deletion, is difficult.

Clarkson [5] developed two data structures for non-Euclidean spaces. He assumes that the samples S and q are drawn from the same (unknown) probability distribution. While his data structures apply to the low-expansion spaces that we consider (as long as the non-trivial assumption of random inputs is satisfied), they have super-logarithmic query times, and do not allow for insertion or deletion of elements.

The already mentioned paper by Plaxton et al [6] contains a data structure for low expansion spaces that allows for locating data objects in a shared network in that space. The data structure cannot be directly used for nearest neighbor search, as it returns only approximately closest data items. Moreover, the construction makes additional crucial assumptions on the space, such as $|B_p(2r)| \geq 8|B_p(r)|$ for all $p \in S$ and $r > 0$, so does not necessarily work on all spaces with low expansion.

Outline. The paper is structured as follows. First, we will derive some additional properties of spaces with low expansion rates. We then describe and discuss our data structure for searching in low expansion metrics, describe an application to a peer-to-peer networking protocol, and conclude the paper with describing some directions for future research.

2. CONSEQUENCES OF LOW EXPANSION

Let us begin by introducing notation, and proving a few facts about spaces with low expansion that show why sampling is a good way to find nearest neighbors in these spaces.

We begin by proving the fact claimed in the introduction.

Lemma 2
A random subset of m points from a metric space with (ρ, c)-expansion will have $(\max(c\rho, O(\log m)), 2c)$-expansion with high probability (in the size of the subset).

Proof: We prove $(\max(c\rho,\mu), 2c)$-expansion for some $\mu = O(\log m)$. Let Z be the sample from the metric space S. Consider a particular ball $B_p(r)$ for some point p in the sample. Let us now condition on the number k of points in $Z \cap B_p(2r)$. If $k \le c\rho$ then the expansion property is vacuously satisfied. Similarly, if $k \le \mu$ we are done.

So we can assume $k \ge c\rho$ and $k \ge \mu$. This implies $B_p(2r)$ contains at least $k \ge c\rho$ points in Z and thus in S. From the expansion property for S we know that $B_p(2r)$ has at most c times as many points as $B_p(r)$ in S. Conditioned on k, the set of points included in the sample is chosen at random from $S \cap B_p(2r)$; thus, each such point is in $B_p(r)$ with probability at least $1/c$. Thus, the expected number of points k' in $Z \cap B_p(r)$ is at least k/c. Since $k \ge \mu$, a standard Chernoff bound implies that $k' \ge k/2c$ with probability $1 - e^{-\Omega(\mu)}$. By choosing $\mu = O(\log m)$ (varying the constant according to the precise desired probability bound) we deduce that $k' \ge k/2c$ with high probability in m.

Our analysis has shown that regardless of k, the ball obeys the expansion property with high probability in m. Thus, the same high probability result holds without conditioning.

This outcome holds with high probability for any particular $p \in Z$ and any particular r. It thus holds with high probability in m for any of the $\binom{m}{2}$ pairwise distances between points in Z. Since these particular distances are the only ones where ball-sizes change, the claim is proven. □

As can be seen in the proof, sampling creates a size $\log m$ neighborhood of a point within which expansion fails to be preserved; this motivated the parameter ρ that excepts small balls from the expansion-rate bound.

Now we turn to developing our data structure. In the following, we will assume that the metric space $\mathcal{M}$ is normalized (by scaling) such that the maximum distance between any two points in S is 1. As before, let n be the size of the subset S.

All the algorithms we describe below can be viewed as random walks on the set S. To find q, we start at an arbitrary point $p \in S$, and step through a sequence of points that quickly converges to q. In fact, in each step we expect to halve the number of points closer to q than our current position, resulting in a $O(\log n)$ query time.

The steps in the random walk are performed by sampling points from S in a ball around our current point p. We will now show how this yields good performance. First, we state a simple claim about inclusions of balls around a pair of points.

Lemma 3 (Sandwich Lemma)
If $d(p,q) \le r$, then $B_q(r) \subseteq B_p(2r) \subseteq B_q(4r)$.

Proof: For the first inclusion, if $s \in B_q(r)$, then $d(p,s) \le d(p,q) + d(q,s) \le r + r = 2r$. For the second inclusion, if $s \in B_p(2r)$, then $d(q,s) \le d(q,p) + d(p,s) \le r + 2r < 4r$. □

This simple observation leads us to the sampling lemma which is the basis for our algorithms.

Lemma 4 (Sampling Lemma)
Let $\mathcal{M}$ be a metric space, and $S \subseteq M$ be a subset of size n with (ρ, c)-expansion, where $\rho = \Omega(\log n)$. Then for all $p, q \in S$ and $r \ge d(p,q)$ with $|B_q(r/2)| \ge \rho$, the following is true.

When selecting $3c^3$ points in $B_p(2r)$ uniformly at random, with probability at least $9/10$, one of these points will lie in $B_q(r/2)$.

Proof: This follows from the Sandwich Lemma. Let $k := |B_q(r/2)|$ be the number of "good" points. Since $B_q(r/2) \subseteq B_q(r) \subseteq B_p(2r)$ by the Sandwich Lemma, all good points are possible results in our sampling. Also due to the Sandwiching Lemma, we have $|B_p(2r)| \le |B_q(4r)| \le c^3 |B_q(r/2)| = c^3 \cdot k$, the last inequality due to the limited expansion rate. Thus, the probability that one sample is good is at least $k/(c^3 k) = 1/c^3$. The probability that c^3 samples are all bad is at most

$$\left(1 - \frac{1}{c^3}\right)^{3c^3} \le \left(\frac{1}{e}\right)^3 \le 0.05.$$

Thus, we succeed with probability more than 90 %. Because the sample space had at least logarithmic size, the effect of sampling without replacement vs. with replacement is negligible. □

2.1 A simple local search algorithm

The Sampling Lemma immediately suggests a nearest neighbor search algorithm.

let p be an arbitrary point in S
while p is not the nearest neighbor of q in S
  let X = random sample of $3c^3$ elements of $B_p(2d(p,q))$
  let p = element of $X \cup \{p\}$ of minimal distance to q

Let us briefly discuss this algorithm. For simplicity, we analyze this scheme in terms of the ratio R between maximum and minimum pairwise distances between points in the metric space, and assume that the set has $(1,c)$-expansion (as opposed to the $(O(\log n), c)$-expansion considered in the rest of the paper).

Theorem 5
The local search algorithm completes with high probability in $O(\log R)$ time.

Proof: We start with a point at distance at most 1 from the query point (recall that we normalized our space this way). By the sampling lemma, each local search step (iteration of the while loop) will halve our distance to q with probability at least $9/10$ (and will never increase it). It follows that the expected number of iterations to halve our distance to q is at most $10/9$, and the expected number of iterations to produce $\log R$ halvings is $(10/9)\log R$. But after $\log R$ halvings, we will have a point at distance at most $1/R$ from q. Since by definition of R there are no points at this small a distance, we must terminate sooner with the nearest neighbor of q.

A standard Chernoff bound on the expectation yields the high probability result. □

A Space-Inefficient Structure

The local search algorithm relies on a random-sampling primitive which we have yet to implement. We provide a data structure that supports the necessary sampling. We saw above that local search takes only $O(\log R)$ time with high probability. It follows trivially that with high probability, we will need to examine only $O(\log R)$ samples from a ball around any particular point p in our search. Our data structure simply chooses these $O(\log R)$ samples in advance for all n points in the metric space.

Of course, we do not have advance knowledge of the query points, so we cannot predict the distance between a point p and the query point. Without such knowledge, we do not know what radius of ball around p to sample from. We get around this problem by choosing from a set of balls with power-of-two radii. One such ball will have radius within twice the current distance to the query point, and we can use the in-advance samples from that ball.

More precisely, for each integer k, each point p chooses a set of $3c^3 \log R$ *level k finger points* uniformly at random from the set

$B_p(2^{-k})$ of points within distance 2^{-k} of p. Denote this set of fingers as $F_k(p)$. This selection is done for each k up to $k = \log R$, since there are no points closer than this distance to p. Note also that the level 0 fingers are simply random points in the metric space. Given these finger points, we use the following algorithm.

QUERY(q)
let p be any point in S
while p is not the nearest neighbor of q
  let k be maximum such that $d(p,q) \le 2^{-k}$
  let $p =$ closest point to q in $F_{k-1}(p)$
return p

We can analyze this algorithm using the Sampling Lemma. Given some point p at distance r from q, with $2^{-(k+1)} < r \le 2^{-k}$, we use samples (fingers) from the ball of radius $2^{-k} < 2r$. The Sampling Lemma thus applies, telling us that we halve the distance of the current point to q with constant probability. The analysis of the previous section therefore applies to tell us the following theorem.

Theorem 6
There is a data structure of size $O(n \log^2 R)$ that answers near-neighbor queries with high probability in $O(\log R)$ time.

Proof: There are $\log R$ distinct powers of 2 between the maximum and minimum distances in the metric spaces. For each such power of 2, we need $O(\log R)$ fingers for each of our n points. □

The problem with this approach, however, is the fact that drawing samples uniformly at random from prescribed spheres is not easy. In particular, if we demand that all these samples are independent of each other, efficient dynamic maintenance of the data structure seems difficult.

If we do not require dynamic maintenance of the data structure, however, this approach can be developed into a data structure by independently choosing the samples in "advance" during construction time. This scheme produces a data structure we call a "metric search tree" with the same time bounds as our current structure that can be analyzed by an application of branching processes. We omit the details in this version of the paper.

Instead, we concentrate on a data structure where the "pre-chosen" samples are not completely independent. While allowing for easier insertion and deletion, this makes the analysis more complicated, as we will see.

3. METRIC SKIP LISTS

We now describe the *metric skip list* data structure that solves the nearest neighbor search problem in metric spaces with constant expansion rates. It follows the sampling paradigm described in the last section. To avoid the problem of creating (and maintaining) completely independent samples, we use a trick previously applied to the design of *treaps* (a dictionary data structure). To construct our data structure, we introduce a random ordering on the points in the sample space S. The construction of the data structure will then be deterministic given the ordering. But, using the fact that the ordering is truly random, we will show good performance guarantees.

For simplicity, we will assume in the following that all pair-wise distances of points in S are distinct (via perturbation).

We impose a random total order on $S = \{s_1, s_2, \ldots, s_n\}$. We call s_{i+1} the successor of s_i, and let s_1 be the successor of s_n, so one can actually imagine the points arranged on a circle.

The data structure consists of sets of pre-chosen samples for every point $s_i \in S$. We will refer to those samples as being "stored at" the corresponding point s_i. For each node s_i, we will store *finger lists*.

Definition 7
For $r \ge 0$ the radius r finger list *for s_i, denoted $F_r(s_i)$, contains the indices of the first $24c^3$ elements after s_i in the ordering that have a distance $\le r$ to s_i. If we reach the end of the ordering, we wrap around to the beginning, and if there are less than $24c^3$ elements of this kind in S, then $F_r(s_i)$ just contains all of them.*
The length k finger lists *are defined analogously, with the constant $24c^3$ replaced by k.*

In the remainder of this section we will analyze the space requirements of the data structure, prove that it can be used to find nearest neighbors in time $O(\log n \log\log n)$, and give an off-line $O(n \log n \log\log n)$ construction algorithm. We defer the problem of dynamic updates (addition and removal of points from S) to the following section. Later in section 4.2 we will also improve the running time of the FIND-algorithm to $O(\log n)$.

3.1 Space requirements

At first glance, it seems that the number of finger lists $F_r(s_i)$ we have to store at a node is not bounded. But it actually turns out that with high probability only $O(\log n)$ of the finger lists are distinct, as we will show now. Thus, it is enough to just store these, indexed by r. This leads to $O(\log n)$ storage per node, or $O(n \log n)$ for the whole data structure.

Lemma 8
Let $S = \{s_1, \ldots, s_n\}$ be a randomly ordered subset of a metric space $\mathcal{M} = (M,d)$, $k \in \mathbb{N}$, and $p \in M$. Then with high probability, there are only $O(k \log n)$ distinct length k finger lists for each s_i.

Proof: We give an algorithm that outputs all the elements in any $F_r(p)$, and analyze its behavior. Consider the following algorithm:

let $j = 0$, $F = \{s_1, s_2, \ldots, s_k\}$
for $i = k+1$ to n **do**
  if $d(p,s_i) < \max_{s \in F} d(p,s)$ **then** // new element closer
    let $j = j+1$,
      $f_j =$ element $s \in F$ maximizing $d(p,s)$,
      $F = (F \setminus \{f_j\}) \cup \{s_i\}$
output $F \cup \{f_1, f_2, \ldots, f_j\}$.

We claim that this algorithm outputs all elements that appear in any length k finger list of p. Clearly, an element s_i will not be in a finger list, if there are k elements before it in the ordering which are all closer to p than s_i. The set F maintained by the algorithm always contains the k closest element to p among $\{s_1, s_2, \ldots, s_{i-1}\}$. Using this invariant, we see that no elements besides the ones output at the end can appear in a finger list.

We will now show that the number of elements in the output is $O(k \log n)$ with high probability. To be output, an element must first enter F. An element s_i enters F if and only if it is one of the k closest elements to p among $\{s_1, \ldots, s_i\}$. These items are a random permutation of (some subset of) S, so s_i is one of the k closest to p with probability k/i. It follows immediately that the expected number of points entering F is $\sum k/i = O(k \log n)$. For the high probability bound, consider generating the list $\{s_1, \ldots, s_n\}$ backwards from the end by repeatedly choosing a random element not yet in the list. From this framework it can be seen that the probability of s_i becoming a finger is independent of all other such events. Thus, the number of fingers is a sum of independent indicator random variables with mean $O(k \log n)$, and thus is $O(k \log n)$ with high probability by the Chernoff bound.

Finally, note that the number of elements output upper-bounds the number of distinct length k finger lists, since the set F (which enumerates all finger lists) changes once each time we produce a new output element in the above algorithm. □

3.2 The FIND-operation

The FIND-operation on our data structure is the "obvious" application of the sampling search strategy given in section 2. The algorithm is defined as follows:

FIND(q) (finds nearest neighbor of q in S)
let $i = 1$ // i is current position
let $m = 1$ // m is minimum so far
while $i < n$ **do**
  let $r = d(s_i, q)$
  if $\exists\ j \in F_{2r}(s_i)$ such that
    $d(s_j, q) < d(s_m, q)$ or $d(s_j, q) \le r/2$ **then**
    let i be the smallest index $\in F_{2r}(s_i)$ with that property
    if $d(s_i, q) < d(s_m, q)$ **then let** $m = i$
  else let $i = \max F_{2r}(s_i)$
output s_m.

Let us first prove the correctness of the algorithm. For this we will not need the fact that S has low expansion, i.e. the algorithm will work correctly (if not particularly efficiently) on any metric space.

Lemma 9
The FIND*-algorithm always returns the nearest neighbor to q in S.*

Proof: We only move forward in the ordering, i.e. i is strictly increasing. An invariant of the algorithm is that s_m is always the point closest to q among $\{s_1, \ldots, s_i\}$. Assume, for contradictions sake, that after moving from i to j, this condition did not hold. Then there must be a point s_k $(i < k < j)$ with $d(s_k, q) < d(s_i, q) = r$. This implies $d(s_k, s_i) < 2r$. But such an s_k should have been included in the finger list $F_r(s_i)$, and would not have been skipped. The contradiction shows the correctness of the algorithm. □

3.3 Running time analysis for FIND

In the following running time analysis, we will make a simplifying assumption to be removed later. When we access a finger list, we do so by the corresponding radius r. The unbounded number of possible values for r prevents us from storing the finger lists indexed by r, however. Therefore we store the finger lists in an array ordered by r. The most straightforward way of locating the correct list is therefore binary search on r, which leads to an additional cost of $O(\log\log n)$ per finger list access. Later, we will reduce this time to constant per finger list.

Theorem 10
The FIND*-algorithm with high probability accesses only $O(\log n)$ finger lists. Thus, its running time is $O(\log n \log\log n)$ with high probability.*

Proof: First observe that the FIND-algorithm actually steps through all the elements that would appear in the length 1 finger lists of q (had q been inserted where we start the search). In particular, any time we take an element s_i with $d(s_i, q) < d(s_m, q)$, we are taking a 1-finger-list element. We will refer to these as "record" items. Note from Lemma 8 that there are $O(\log n)$ record items with high probability. However, the algorithm will also encounter nodes that are *not* record items. We must bound this work.

We consider the running time of the algorithm on the first third $S_{n/3} = \{s_1, s_2, \ldots, s_{n/3}\}$ of the elements, and prove that with high probability it is only $O(\log n)$. But "high probability" implies that the bound is true for any starting point, in particular the point where we end up after processing the first third of the points. This allows us to process the second (and eventually last) third of the points again in $O(\log n)$ each, yielding at total time bound of $O(\log n)$.

So consider the first third $S_{n/3}$. Suppose that our search is currently at item s_k, having passed the set $S_k = \{s_1, \ldots, s_{k-1}\}$. The set $S - S_k$ is (thanks to our random ordering) a random subset of S, of size at least $2n/3$. It follows from Lemma 2 that $S - S_k$ is a metric space with $(O(\log n), 2c)$-expansion. In other words, with high probability, for all $r > 0$ such that $|B_q(r)| = \Omega(\log n)$, we have $|B_q(2r) \cap (S \setminus S_k)| \le 2c \cdot |B_q(r) \cap (S \setminus S_k)|$.

We briefly defer the case of $|B_q(r)| = O(\log n)$. For larger balls, we have just argued that the elements in the finger list are drawn at random from a space with expansion rate $2c$. Thus drawing $24c^3 = 3(2c)^3$ of them will yield one with distance at most $d_i/2$ with probability at least $9/10$, by Lemma 4. This tells us that the **if** test in the FIND algorithm will be satisfied with probability at least $9/10$ in each iteration.

To outline our argument, we make two assumptions that must be revisited later. First, we assume any record-breaking step *also* reduces the distance by half. Second, we assume that the outcomes of the iterations are *independent*. Under these two assumptions, we can analyze our algorithm as a random walk.

Let $\langle d_1, d_2, \ldots \rangle$ be the distances to q of the elements s_i visited during the execution of FIND on the first third of the elements. We will model $\langle d_i \rangle$ as a random walk. The analysis above (and our assumption about record events) means that with probability $9/10$ the **if** test succeeds and we halve our distance to q. When the **if** test fails, by definition and the Sandwich Lemma, we have $d_{i+1} \le 4d_i$. It follows that $E[\log d_{i+1}] \le \log d_i - 9/10 + 2 \cdot 1/10 = \log d_i - 7/10$. In other words, the random walk has negative drift.

Such a random walk has the property that in $O(\log n)$ moves, it will move $O(\log n)$ times to a value below any previous encountered value. Such a move discovers a record breaking item. Since by Lemma 8 there are only $O(\log n)$ record-breakers, we will have found them all within $O(\log n)$ time steps.

We must now revisit our assumptions. Our argument that the distance halves with probability 9/10 per iteration ignored two cases. First, if we are on one of the $\rho = O(\log n)$ closest points to q, then the expansion rate need not hold. Second, in iterations where we encounter a record breaker, the distance need not halve. Note that each outcome (being on a close point, or encountering a record breaker) happens only $O(\log n)$ times. We model this by letting an adversary "cancel" $O(\log n)$ of the distance-halving steps that occur in our algorithm. The random walk analysis generalizes to this case and still shows that $O(\log n)$ record-breakers will be encountered within $O(\log n)$ steps.

Finally, it remains to justify our assumption above that the random walk steps are independent. We use the principle of deferred decisions. In an iteration starting at s_i, our algorithm selects the closest item $s_j \in F_{2r}(s_i)$ that satisfies the **if** test. We can identify s_j by walking forward on the list from f_i, adding valid items to the finger list until the finger list is full or we encounter an appropriate s_j. In this variant of the algorithm, we reach s_j before examining *any* node following s_j in the list. Thus, by the principle of deferred decisions, the fingers of s_j are independent of all our previous steps. □

3.4 Range Queries

In a range query we want to find all elements in S that are within a distance r of a query point q. These queries can be answered by a variant of our FIND-algorithm in time $O((\log n + k)\log\log n)$ with high probability, where k is the number of returned points.

We simply modify the FIND-algorithm to never query finger lists with a radius of less than $2r$. This ensures that we will never miss any of the points within the required ball around q. Similar to the original FIND, the behavior of this algorithm can be analyzed as a random walk, where the points within distance r of q are considered to be one set at distance r to q. It takes at most time $O(\log n)$ for the random walk to get within distance r of q, and then $O(k)$ time to visit all points in the query radius, since it takes expected constant time between visiting successive elements in the ball. This yields a total bound of $O(\log n + k)$ finger list accesses, for a $O((\log n + k)\log\log n)$ running time.

3.5 Offline construction

Before considering dynamic updates to the data structure, we first present a simple $O(n\log n\log\log n)$ off-line construction algorithm for our data structure.

Suppose we truncate all finger lists $F_r(s_i)$, so that they do not "wrap around" at the end of the ordering, but rather only include elements after s_i in the ordering. This data structure would still support searches, as long as they start at s_1, at the beginning of the ordering.

Constructing this data structure can be done by starting with an empty data structure, and repeatedly adding $s_n, s_{n-1}, s_{n-2}, \ldots, s_1$ to the beginning of the previously constructed data structure. This way, when we insert s_i, we only have to compute s_i's finger lists, while the finger lists of already inserted elements remain the same (since, by definition, they will not contain s_i). At each step, the data structure so far is a metric skip list on the (random) subset of items $\{s_{i+1}, \ldots, s_n\}$. This random sample has low expansion (Lemma 2) so the finger list computations for s_i are fast.

Constructing the new element's finger lists

The construction of the new element s_i's finger lists can be done by a modification of the FIND algorithm. We start a search for s_i at s_{i+1}, the successor of s_i. But instead of just maintaining the closest element s_m encountered so far, we keep the $24c^3$ closest elements seen so far, dropping the furthest element of that set as soon as a closer element becomes available. As seen in the proof of Lemma 8, this yields all elements of s_i's finger lists.

The running time analysis of this search is virtually identical to the one given for the FIND-algorithm. The main change is that the number of finger elements is now $24c^3$ times as large, which causes the random walk on the distances to be slower by that factor. But it still yields a $O(\log n\log\log n)$ bound with high probability.

Thus, we can construct our data structure in time $O(n\log n\log\log n)$. The problem is that this approach does not immediately lend itself to performing dynamic updates, because by always inserting at the "beginning" of the ordering, we would not guarantee that the ordering remains random.

4. DYNAMIC MAINTENANCE

In this section we describe how the metric skip list data structure can be maintained dynamically, i.e. how elements can added and deleted in $O(\log n\log\log n)$ amortized expected time each. We will focus mostly on the INSERT-operation to add a new node to the structure, as the analysis for the deletion of a node is very similar.

4.1 Augmenting the data structure

To make dynamic maintenance of the search data structure possible efficiently, we have to add two more pieces of data to the basic structure described in the previous section. These are:

Nearest Neighbor Lists $NN(s_i), R(s_i)$**:** For every point s_i we maintain a radius $R(s_i) > 0$ such that $c\log n \le |B_{s_i}(R(s_i))| \le 3c\log n$, and a list $NN(s_i)$ of the points in $B_{s_i}(R(s_i))$.

Query Lists $Q_r(s_i)$**:** For each finger list $F_r(s_i)$ we store an associated list of all elements s_j that queried $F_r(s_i)$ when constructing their own finger lists. The exception to this is that we do not record s_j's queries for its finger lists with radius less than $R(s_j)$.

These additions do not change the $O(n\log n)$ space requirement of the data structure. The nearest neighbor lists require $O(\log n)$ space per node. And since the number of queries performed when constructing an element's finger lists is $O(\log n)$ with high probability, the total number of entries in the query lists is $O(n\log n)$.

4.2 A faster FIND

It turns out that with only small changes, the FIND-algorithm can be modified to run time $O(\log n)$ with high probability, with a corresponding bound of $O(\log n + k)$ for the range search. We will use these bounds from now on to compute running times.

We change the FIND-algorithm so that once it queries a finger list $F_r(s_i)$ with $r < R(s_i)$, we stop the random walk, and just return the closest element to the query point in $NN(s_i)$, which takes time $O(\log n)$ by a linear pass.

The only reason that we incurred an additional $O(\log\log n)$ factor in the running time of the FIND-algorithm so far is that this was the time per accessed node to find the correct finger list. Suppose now that we further augment our data structure such that for each $j \in F_r(s_i)$ there is an associated pointer to $F_r(s_j)$, and also pointers from $F_r(s_i)$ to $F_{2r}(s_i)$. Maintaining these pointers can be subsumed in the total time taken for INSERT and DELETE.

Thus, we can find the next finger list in the FIND-procedure in constant time if the radius we query is "not too far off" the previous one. But this is true in the case of the FIND algorithm. The query radius never increases more than from r to $2r$, and we can find that finger list in time $O(1)$. From that point, we can seek through decreasing r for the correct radius finger list. Each drop in radius takes $O(1)$ time, but can be "charged" against the improved distance—in the FIND algorithm, we can basically think of this as a random walk step which improves with probability 1 rather than 9/10. Thus the analysis goes through unchanged, showing $O(\log n)$ steps, but now the "steps" bound not only the number of nodes visited but also the work at each node.

A similar change can be made to the range query procedure given in section 3.4, yielding a running time of $O(\log n + k)$.

4.3 Range queries revisited

In section 3.4, we considered the range query of reporting all points within distance r of a point q. Now we consider the related query, where given a number k and a point q, we are asked to output the k closest points to q. This type of query will be useful to us for the INSERT procedure.

Unfortunately, this problem cannot be directly reduced to the one we already solved, as finding the correct r such that $|B_q(r)| = k$ is difficult. It would be considerably easier if in addition to the expansion property we also had a lower bound of the form $|B_q(2r)| \ge c'|B_q(r)|$ because that would relate r and k very closely. However, it is still possible to solve the problem efficiently.

Lemma 11
There is an algorithm that, given a point q and a number k, outputs the k closest neighbors of q in time $O(\log n + k)$. □

Due to its technical nature, we defer the proof to the appendix.

4.4 The INSERT-operation

We have seen in section 3.5 how to construct our data structure in an offline manner, and in particular how to construct the finger lists of any newly inserted element q. The main change for dynamic updates is that now insertions do not occur "at the beginning" of the ordering, but rather at a random position in the middle. This means that the new element q has to appear on other elements' finger lists. The main difficulty of making insertion efficient is to be able to quickly find the elements that should point to q.

How do we find the elements s_i that should include q in their finger lists? An element could have computed different finger lists, had q been present, only if it queried a finger list that should have contained q as well during its construction. This observation is only self-referential at first glance: to construct a finger list $F_r(s_i)$ of radius r, s_i would query only finger lists with radius $2r$ or higher.

Using the query lists we introduced, the elements s_i that have to point to q can be found using a search backwards along query pointers. First, we know that q has to be included in the distance 1 (the maximal distance) finger lists for the $24c^3$ elements preceding it in the ordering. Then, recursively, whenever we add q to a finger list, we check all elements s_i mentioned in the associated query list to determine which of these should change their finger lists, and so on.

The correctness follows, since, as mentioned above, for constructing a finger list of radius r, the INSERT procedure would only query finger lists of radius $2r$ and higher. That means that by fixing all radius 1 finger lists, we will inductively fix all other affected lists.

4.4.1 *The algorithm*

In summary, the INSERT-procedure for a new element q is the following.

1. Insert q at a random position in the ordering.
2. Construct q's finger lists by searching forward in the ordering (cf. section 3.5)
3. Find $3c^2 \log n$ nodes closest to q, compute $R(q)$, $NN(q)$ such that $|NN(q)| = 2c\log n$.
4. Include q in $NN(s_i)$ among the $3c^2 \log n$ nodes s_i closest to q, as necessary.
5. Follow query pointers backwards to find all nodes that see q in the construction of their finger lists. For these elements and the $3c^2 \log n$ closest neighbors:
   (i) Update their finger lists as necessary to include q.
   (ii) Update the query list entries they caused, as necessary.

We will now bound the expected running time of the INSERT-operation, and give more details on the implementation.

4.4.2 *The analysis*

We perform the running time analysis step by step of the algorithm. To do it, we introduce one more definition. The *q-rank* of point p is the number of points closer to q than p is.

Step 2: Constructing q's finger lists
Computing the finger lists can be done by a modified FIND-operation as in section 3.5, using time $O(\log n)$.

Step 3: Finding $3c^2 \log n$ closest neighbors
We can do this using the procedure from Lemma 11 in time $O(\log n)$. In time $O(\log n)$ we can find the element of q-rank $2c\log n$ among these elements, which yields $R(q)$ and $NN(q)$.

Step 4: Include q in $NN(s_i)$
Due to the expansion property, any node s_i for which q is among the $3c\log n$ closest nodes must be among the $3c^2 \log n$ nodes closest to q. So this step of the algorithm suffices to maintain the correct values of the $NN(s_i)$.

Using the list of the $3c^2 \log n$ nearest neighbors s_i computed in step 3, we can check for each of them in $O(1)$ by comparing $d(s_i,q)$ to $R(s_i)$ to determine whether q should be included in $NN(s_i)$.

When this causes an $NN(s_i)$ to grow beyond size $3c\log n$, we can recompute that element's finger lists, $NN(s_i)$ and $R(s_i)$ (such that $|NN(s_i)| = 2c\log n$) in time $O(\log n)$. Since it takes $c\log n$ insertions to cause such a change, the amortized cost per insertion is only $O(1)$.

Thus, the total amortized time of step 4 is $O(\log n)$.

Step 5: Following query pointers backwards
In the next step of the algorithm, we follow query pointers backwards to determine which finger lists have to be updated following the insertion of q.

First, we will show that the number of nodes that we reach using this search is $O(\log n)$ in expectation. Then, we will show that each of these nodes is pointed to by only a constant number of query pointers, so that the query pointer traversal takes total time $O(\log n \log\log n)$, the additional $O(\log\log n)$ factor coming from having to look up the correct finger and query lists.

Lemma 12
The expected number of elements that encounter q in the construction of their finger lists is $O(\log n)$.

Proof: Here and in the following we will use different methods to bound the work spent on the $c\log n$ nodes closest to q and all the other nodes.

In this case, we have to give no particular analysis for the $c\log n$ closest nodes, since even if they all encountered q they would only be $O(\log n)$ nodes.

For the remaining $n - c\log n$ nodes we are going to use a bounding technique that we will also use repeatedly in the remaining runtime analysis. Let s_i be some fixed node in the structure, and $p_{\text{see}}(r)$ be the probability that s_i sees the element of s_i-rank r during its finger list construction. This probability $p_{\text{see}}(r)$ has the property that it is monotonically decreasing in r. This is because if an element of rank r were replaced by an element of rank r' with $r' < r$ in the same ordering, s_i would still see this element in its search, if it saw the element of rank r before. Thus $r' < r \implies p_{\text{see}}(r') > p_{\text{see}}(r)$. Also, since the finger list construction only depends on the ranks of the elements, $p_{\text{see}}(r)$ is actually independent of the element s_i.

Two more observations yield the desired bound. First, since the number of nodes that are encountered in a finger list construction is $O(\log n)$ in expectation, we have $\sum_{r=1}^{n} p_{\text{see}}(r) = O(\log n)$. Second, if an element s_i has q-rank $r \geq c\log n$, then q has a s_i-rank of at least r/c. This follows directly from the expansion property of the metric space. Thus, we have

$$E[\#\text{elements that "see" } q] = \sum_{r=1}^{n} \Pr[\text{element of } q\text{-rank } r \text{ "sees" } q]$$

$$\leq O(\log n) + \sum_{r=c\log n}^{n} p_{\text{see}}(r/c) \leq O(\log n) + c\sum_{r=1}^{n} p_{\text{see}}(r) = O(\log n),$$

proving the lemma. □

Lemma 13
Let s_i be any node. Then the number of finger lists that overlap q's position, that are queried in the construction of s_i's finger lists, is at

most $24c^3 = O(1)$. Thus, s_i can "see" q only a constant number of times during the construction of its finger lists.

Proof: When constructing s_i's finger lists, the first time we access a finger list that overlaps q's position in the ordering, this finger list by definition contains all elements before q that we might include into s_i's finger list. This number is therefore at most $24c^3$. If we move to one of these points, then by the next time we access a finger list overlapping q's position, their number has decreased by one. So after seeing q's position at most $24c^3$ times, we will move past it. □

The previous lemma implies that only $24c^3$ query pointers from finger lists containing q can point to an element s_i that "sees" q during its finger list construction. Thus the running time for the "query pointer search" part of step 5 is $O(\log n \log\log n)$.

Step 5(i): Updating finger lists

The time taken to create the finger lists containing q is at most $O(\log\log n)$ times the number of these finger lists. The $O(\log\log n)$ is taken to find the correct insertion point of the new finger list among the other finger lists. We will now show that q appears in an expected number of $O(\log n)$ finger lists, yielding a total bound for this step of $O(\log n \log\log n)$.

We will first use a coarse bound that we will use for the $c\log n$ elements closest to q.

Lemma 14

Let s_i be any node. Then the expected number of s_i's finger lists that q appears in is $O(1)$.

Proof: Consider the construction of s_i's finger lists. If we imagine s_i being at the beginning of an order of n elements, and inserting q at a random position in that order, then q can only appear on the finger lists that are created "after" its insertion point in the ordering.

Consider some position k in the order. The $24c^3$-rd smallest s_i-rank of the elements before the k-th position is expected to be close to $24c^3 n/k = O(n/k)$. So since the remaining finger lists are constructed from only $O(n/k)$ elements, the proof of Lemma 8 implies that the expected number of finger lists created after this point is $O(\log(n/k))$.

If q gets inserted at a random position, the expected number of finger lists created after that point is therefore on the order of

$$\frac{1}{n}\sum_{k=1}^{n}\log\frac{n}{k} = \log n - \frac{\log n!}{n} = \log n - (\log n - O(1)) = O(1),$$

as claimed. □

Again, this bound is enough for the $c\log n$ elements closest to q, as it yields an expected total number of $O(\log n)$ occurrences among these elements, but we require a different approach for the remaining elements.

If $E_{\text{fl}}(r)$ is the expected number times that an element of s_i-rank r appears in s_i's finger lists, then again we have $\sum_{r=1}^{n} E_{\text{fl}}(r) = O(\log n)$ and $r' < r \implies E_{\text{fl}}(r') > E_{\text{fl}}(r)$, by similar arguments as in the proof of Lemma 12 for $p_{\text{see}}(r)$. Following the same proof technique as above, this shows that the expected total number of finger lists that q appears in is $O(\log n)$ among all elements.

Lemma 15

An element q appears in $O(\log n)$ finger lists in expectation. □

Step 5(ii): Updating query lists

All the nodes that "see" q in the construction of their finger lists might now take potentially different query paths than before the insertion of q. So we have to construct as much of the new query paths as might be influenced by the insertion of q.

For a node s_i that includes q in its finger lists, the queried finger lists might be different as long as q remains among the closest $24c^3$ elements. For a node s_i that does not include q in its finger lists, the queried finger lists might be different up to the next element that s_i *does* include into its finger lists.

Lemma 16

The insertion of q causes $O(\log n)$ query list pointers to change in expectation.

Proof: We will first treat the nodes s_i separately for which q is one of the $c\log n$ closest nodes to s_i (i.e. $q \in NN(s_i)$). For each of these nodes, the expected number of changed query pointers is $O(1)$, for a total of $O(\log n)$ for all these s_i.

This is because only $24c^3$ of the nodes of $NN(s_i)$ appear in s_i's finger lists with radius $\geq R(s_i)$, and only these could result in a change of query pointers. The probability that q is one of these nodes is therefore $O(1/\log n)$. As the worst case number of changed query pointers is $O(\log n)$, the expected number of pointer that have to be changed is $O(1/\log n \cdot \log n) = O(1)$.

For all other elements, we are going to distinguish between the cases that q does appear or does not appear on their finger lists. First, the elements that include q in their finger lists.

Let $E'(r)$ be the expected number of queries performed for the finger list construction of an element s_i, such that q has s_i-rank r, while q is among the elements of the current finger lists. As there are always $24c^3$ elements in the current finger list, we end up charging each query to $24c^3$ elements. Since the expected total number of queries for s_i is $O(\log n)$, we therefore have

$$\sum_{r=1}^{n} E'(r) = 24c^3 \cdot O(\log n) = O(\log n).$$

As an element with a lower rank in the same position in the search will stay as long or longer in the finger list during the search, we have that $r' < r \implies E'(r') \geq E'(r)$. A similar calculation as in Lemma 12 gives that q is expected to influence at most a total number of $O(\log n)$ query pointers.

Second, we have to analyze the number of query pointers influenced by q among the elements that see q during their finger list construction, but do not actually include q among their finger list elements. The changes that q might necessitate are limited to the finger list construction from seeing q until the next finger list element is found. We will bound this in a similar fashion as above: let $E''(r)$ be the expected number of elements seen by s_i after an element of rank r, before the next element of its finger lists is found (we bound this independent of whether q is actually a finger list element itself, we only bound the number of search steps until the next element is found). Again, we have $r' < r \implies E''(r') \geq E''(r)$, because s_i is more likely to see an element of lower rank in its finger list construction.

We have

$$\sum_{r=1}^{n} E''(r) = \sum_{j=1}^{O(\log n)} E[X_j^2/2],$$

where X_j is the number of elements seen between the j-th and the $(j+1)$-st of s_i's finger list elements that we find, searching from s_i. Since the variables X_j are distributed geometrically with a constant expectation, we also have $E[X_j^2] = O(1)$, and therefore $\sum_{r=1}^{n} E''(r) = O(\log n)$. So in this case also, q is expected to influence at most a logarithmic number of query lists. □

Thus, the running time of the INSERT-operation is expected total time $O(\log n \log\log n)$. Note that we did not analyze the cost to remove finger and query list entries now obsolete due to the insertion of q. But since our work will be linear in their size, and the size of the structure is expected to grow upon insertion of an element, the above bounds dominate the insertion time.

4.5 The DELETE-operation

For the DELETE-operation, we use the query lists to determine all finger lists containing q. Essentially, we do the opposite of what we did for INSERT, and rebuild a node's finger lists if $NN(s_i)$ becomes smaller than $c\log n$. Thus, the same time bounds apply, and the operation takes time $O(\log n \log\log n)$.

5. APPLICATION TO PEER-TO-PEER NETWORKS

The nearest neighbor search structure described in this paper can be used with only slight modifications in the Chord Peer-To-Peer network protocol co-developed by the first author [7]. We will now give a brief introduction to the relevant parts of the Chord data management protocol and describe how the nearest neighbor structure can be used in this context.

The Chord protocol allows for distributed data access by storing data items in multiple locations across the network. Every node gets assigned a random identifier, and the nodes can be imagined ordered on a circle based on these IDs. If an data item is stored in the network, it has one primary location, where the original copy of the item is stored. To speed up accesses, an item might be replicated to be stored at other nodes besides the primary location. In Chord, these copies are held on the nodes which immediately precede the primary location in the order of IDs. I.e., the item copies propagate backwards along the ordering, and the nodes that hold the item always form a continuous segment of the ordering of nodes.

In many applications of Peer-To-Peer networks, such as in wireless networks, the cost of accessing an data item grows as the distance to the item increases. Thus, it is advantageous to locate the "closest" copy of a data item to speed up accesses and lower operating costs. We will now show how the data structure developed in the previous sections can be used to this effect.

Observe that our data structure can easily be made "distributed" by simply storing the finger and query lists at the node in the network that they are associated with. This requires only $O(\log n)$ additional storage for each node. As the nodes in Chord already have random IDs and are therefore naturally ordered in a random order, we can make use of the same ordering for our data structure.

The FIND, INSERT and DELETE protocols work as before, although in practice it might not be possible to store pointers into another node's finger lists that allow for $O(1)$ lookups of the correct lists. Thus, a list lookup cost of $O(\log\log n)$ is more realistic, and the operations become more costly by that factor. But since the cost of an operation in a Peer-to-peer network is dominated by the number of node-to-node communications (which stays $O(\log n)$) and not by in-node computations, this does not seem to be a great loss.

Suppose now that we want to find the closest copy to a node q of an data item whose primary locations is node p. The search is similar to the FIND-operation with the difference that we will not search beyond p, since all copies of the data item are stored directly before the node p in the ordering. On a high level, the search algorithm is as follows:

1. If the item is stored at node q, stop and access the item.
2. Find a location p' containing the item, so that there is no closer node with the item between q and p' in the ordering.
3. Perform a FIND from p' for q, keeping track of the closest item containing node seen, and stop as soon as the FIND operation moves beyond p.

Step 1 needs no explanation – if the item is already at our current location there is no need to look for it elsewhere. Step 2 is straightforward by a modification of the Chord lookup protocol—that protocol does a binary search for q, and it can be modified to report an "overshoot" if it encounters any p that contains the item. Step 3 finds the closest copy of an item, assuming that the second step completed successfully, because during the FIND operation we will see all elements that are closer to q than the previously closest element we saw, starting out with p'. So we cannot possibly miss seeing the closest element to q between p' and p, which is the one we are looking for. And obviously, we can terminate the search as soon as we have passed p since the following elements do not contain the item. Both steps only take time $O(\log n)$ (or rather access that number of nodes).

6. CONCLUSION

We have introduced a new data structure for nearest neighbor search in metric spaces with low expansion rates. The structure is simple, efficient, and can easily be use in a distributed environment, for example if points correspond to nodes in a network.

An interesting open problem is to make the data structure fault tolerant, in the following sense. If the data structure is actually distributed on a set of nodes, such as a Peer-To-Peer network, then it is not unlikely that single nodes just "fail" without invoking a deletion procedure. Is it possible to augment the data structure to work even when nodes (and their associated finger lists) just disappear?

Another interesting property of low expansion metrics is that they can be embedded into an $O(\sqrt{\log n})$-dimensional Euclidean space with low distortion (as opposed to the $O(\log n)$-dimensional space required for general metrics). This allows for approximate nearest neighbor searches by reduction to the Euclidean case. It would be interesting to see whether this avenue leads to other approaches to this problem.

7. REFERENCES

[1] J. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.

[2] J. Bentley, B. Weide, and A. Yao. Optimal expected-time algorithms for closest point problems. *ACM Transactions of Mathematical Software*, 6(4):563–580, 1980.

[3] S. Brin. Near neighbor search in large metric spaces. In *Proceedings VLDB*, pages 574–584, 1995.

[4] E. Chávez, G. Navarro, R. Baeza-Yates, and J. L. Marroquín. Searching in metric spaces. *ACM Computing Surveys*, 33(3):273–321, 2001.

[5] K. Clarkson. Nearest neighbor queries in metric spaces. *Discrete Computational Geometry*, 22(1):63–93, 1999.

[6] C. G. Plaxton, R. Rajaraman, and A. W. Richa. Accessing nearby copies of replicated objects in a distributed environment. *Theory of Computing Systems*, 32:241–280, 1999.

[7] I. Stoica, R. Morris, D. Karger, F. Kasshoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings ACM SIGCOMM*, 2001.

[8] J. B. Tenenbaum. Mapping a manifold of perceptual observations. In *Advances in Neural Information Processing Systems*, volume 10. The MIT Press, 1998.

[9] J. B. Tenenbaum, V. de Silva, and J. C. Langford. A global geometric framework for nonlinear dimensionality reduction. *Science*, 290(5500):2319–2323, December 2000. See also `http://isomap.stanford.edu`.

[10] J. Uhlmann. Satisfying general proximity/similarity queries with metric trees. *Information Processing Letters*, 40:175–179, 1991.

[11] P. Yianilos. Data structures and algorithms for nearest neighbor search in general metric spaces. In *Proceedings SODA*, pages 311–321, 1993.

APPENDIX

The Algorithm

In this section we prove Lemma 11: we give an algorithm that reports the k closest neighbors of a node q in time $O(\log n + k)$. The algorithm can be summarized as follows:

K-RANGE-QUERY(q,k) (returns k nearest neighbors of q in S)
let $s_m =$ nearest neighbor of q
let $r = R(s_m)$, $B = B_q(r)$
while $|B| < k$ **do**
  let $r = \text{AUGMENT}(r, B, q)$, $B = B_q(r)$
output k points in B closest to q

Note that the assignments $B = B_q(r)$ here correspond to range queries in the sense of section 3.4.

The sub-routine AUGMENT(r,B,q) that we have to define later can be called with $B = B_q(r)$ and $B \geq \log n$. It returns with high probability in time $O(|B|)$ an $r' > r$ such that

$$\left(1 + \frac{1}{48c^3}\right)|B| \leq |B_q(r')| \leq 7c^3 \cdot |B| \tag{1}$$

In other words, $|B|$ grows by a constant factor each time.

The correctness of the above algorithm is clear. As for the running time, the first two lines take time $O(\log n)$. Every execute of the while loop takes time $O(\log n + |B|)$. Since $|B|$ grows geometrically, the sum of these times collapses to $O(\log n + k)$, as claimed.

The AUGMENT-Procedure

The AUGMENT-procedure overcomes the problem that by there is no bound on by how much we have to increases r to increase the size of $B_q(r)$ by a constant factor. The procedure uses sampling near B to obtain an estimate on the local growth rate of $B_q(r)$.

AUGMENT(r,B,q)
(let $B = \{b_1, b_2, \ldots, b_\ell\}$ be an enumeration of B's elements)
for i = 1 to ℓ **do**
  let $a_i :=$ element closest to b_i in b_i's finger lists with $d(a_i,b_i) > 2r$
  let $r_i := d(a_i,b_i)$
output median of $\{r_i \mid 1 \leq i \leq \ell\}$

The running time of this algorithm is $O(|B|)$ as each iteration of the loop requires a lookup of $F_{2r}(b_i)$, which can be done in constant time as in section 4.2, because we already have pointers to $F_r(b_i)$ after the range search that computed B. Note that if there is no element in b_i's finger lists of distance more than $2r$ to b, then we can set $r_i = \infty$.

For correctness, we need to prove that the inequalities (1) hold (we use r' to refer to the value returned by the AUGMENT-procedure).

The lower bound is the easier of the two. Note that none of the a_i are in B, as their distance to q is at least $d(a_i,q) \geq d(a_i,b_i) - d(b_i,q) > 2r - r = r$. On the other hand, by returning the median of the r_i, we guarantee that the new ball $B_q(r')$ now contains at least half of the a_i. It remains to show that there are not too many repetitions among the a_i.

For this and the following it will be useful to recall that the b_i are arranged on a circle order in the search data structure. For notational simplicity assume that they occur in the order b_1, b_2, …, b_ℓ in the data structure. Notice that we have $a_i \neq a_j$ if $i + 24c^3 < j$. This is because b_i would encounter $b_{i+1}, b_{i+2}, \ldots, b_{i+24c^3}$ before seeing a_j in its finger list construction, and since all the b's are closer to b_i than a_j, a_j would not be included into b_i's finger list. That is, each a_i appears at most $24c^3$ times among the other a's. Thus, $|B_q(r')| \geq |B| + \frac{1}{2}\frac{1}{24c^3}|B|$, which shows the first inequality.

While the previous inequality holds unconditionally, we will prove that the second inequality holds with high probability. Let p be the element of q-rank $7c^2|B|$, and $R := d(p,q) + r$. We will show that with high probability $r' \leq R$. This is enough, because by the expansion property $|B_q(R)| \leq c|B_q(d(p,q))| = 7c^3|B|$, using $R = d(p,q) + r < 2d(p,q)$.

Let X_i be the event that $r_i \leq R$. To show that with high probability at least half of the X_i occur, we will bound them by other events. Let P be the set of elements with q-rank in $\{|B|+1, \ldots, c^2|B|\}$, and Q be the set of element with q rank in $\{c^2|B|+1, \ldots, 7c^2|B|\}$. Note that the elements of Q all are at least distance $4r - r = 3r$ from the elements in B, and thus would be considered as choices for the a_i in the AUGMENT procedure.

In the random order of the data structure the sets P and Q occur interleaved with the points $b_1, b_2, \ldots, b_\ell$. Let us focus on the random sub-ordering consisting just of these $7c^2|B|$ elements. Let Y_i be the event that b_i is directly followed by an element of Q in this ordering. We then have that Y_i implies X_i. This is because if b_i includes the element of Q that directly follows it in its finger list, then that element is a valid candidate for a_i, and since it is closer than R to q, X_i is true. On the other hand, if b_i does not include this element in its finger list, then there must be $24c^3$ elements closer than it to b_i between the two in the ordering. Since Y_i holds, these elements must have q-rank $> 7c^2|B|$, thus cannot be closer than $2r$ to b_i, but also cannot be further than R from b_i, thus X_i is satisfied.

We now proceed to bound the probability that less than half of the Y_i happen. Fix any ordering of the b_i and P. If we imagine inserting the elements of Q at random places in this ordering, then there are $\binom{|B|+|P|+|Q|}{|Q|}$ ways to do so. If, however, more than half of the Y_i are not true, then the elements of Q may not be inserted directly behind the b_i for which Y_i is not true. Thus, there are only $\binom{|B|/2+|P|+|Q|}{|Q|}$ choices for an ordering. By multiplying with the number of possible choices for which Y_i are not true, we obtain an upper bound on the probability of failure by:

$$\begin{aligned}
&\frac{\binom{|B|}{|B|/2}\binom{|B|/2+|P|+|Q|}{|Q|}}{\binom{|B|+|P|+|Q|}{|Q|}} \\
&\leq \Pi_{k=1}^{|B|/2}\left(\frac{|B|/2+|P|+k}{|B|/2+|P|+|Q|+k}\right)\cdot(2e)^{|B|/2} \\
&\leq \left(\frac{|B|+|P|}{|B|+|P|+|Q|}\cdot 2e\right)^{|B|/2} \\
&= \left(\frac{c^2|B|}{7c^2|B|}\cdot 2e\right)^{|B|/2} \leq 0.882^{|B|}
\end{aligned}$$

This is polynomially small since $|B| \geq \log n$.