COVER TREES FOR NEAREST NEIGHBOR

ALINA BEYGELZIMER, SHAM KAKADE, AND JOHN LANGFORD

ABSTRACT. We present a tree data structure for fast nearest neighbor operations in general n-point metric spaces. The data structure requires $O(n)$ space *regardless* of the metric's structure. If the point set has an expansion constant $c > 2$ in the sense of Karger and Ruhl [KR02], the data structure can be constructed in $O\left(c^6 n \log n\right)$ time. Nearest neighbor queries obeying the expansion bound require $O\left(c^{12} \log n\right)$ time. In addition, the nearest neighbor of $O(n)$ points can be queried in $O(c^{16} n)$ time. We experimentally test the algorithm showing speedups over the brute force search varying between 1 and 2000 on natural machine learning datasets.

1. INTRODUCTION

Nearest neighbor search is a fundamental problem with a number of applications in peer-to-peer networks, lossy data compression, vision, dimensionality reduction, computational biology, machine learning, and physical simulation. The standard setting is as follows: Given a set S of n points in some metric space (X, d), the problem is to preprocess S so that given a query point $p \in X$, one can efficiently find a point $q \in S$ which minimizes $d(p, q)$. For general metrics, finding (or even approximating) the nearest neighbor of a point requires $\Omega(n)$ time. However, the metrics and datasets of practical interest typically have some structure which may be exploited to yield significant computational speedups.

Most research has focused on the special case when the metric is Euclidean. Although many strong theoretical guarantees have been made here, this case is quite limiting in a variety of settings when the data do not naturally lie in a Euclidean space or when the data is embedded in a high dimensional space, but has a low dimensional intrinsic structure [TSL00, RS00] (as is common in many machine learning problems).

Several notions of metric structure and algorithms exploiting this structure have been proposed [Cla99, KR02, KL04a]. Karger and Ruhl [KR02] stated a notion of the intrinsic dimensionality of a dataset and an efficient randomized algorithm for metric spaces in which this dimension is small. Krauthgamer and Lee [KL04a] suggested a more robust notion of intrinsic dimensionality (based on the theory of analysis in metric spaces [GKL03]), and presented a simpler deterministic data structure, called a *navigating net*, that is efficient with respect to this notion. A navigating net is a leveled directed acyclic graph where each consequent level "covers" the dataset on a finer scale; adjacent levels are connected by pointers allowing navigation between scales. Both of these algorithms have $O(\log n)$ query time (assuming the Karger-Ruhl abstract dimension is constant).

In machine learning applications, most of these theoretically appealing algorithms are not used in practice (in both the Euclidean and the general case) for a variety of reasons. When the Euclidean dimension is small, one typical approach uses KD-trees (see [FBL77]). If the metric is non-Euclidean (or the Euclidean dimension is large), *ball trees* [Omo87] [Uhl91] provide compelling performance in many practical applications [GM00]. These methods currently have only trivial query time guarantees of $O(n)$, although improved performance may be provable given some form of structure.

We propose a simple data structure, a *cover tree*, for exact and approximate nearest neighbor operations. The data structure improves over the results in [KR02, KL04a] by making the space requirement *linear* in the dataset size. This space bound is independent of any dimensionality assumptions while a navigating net [KL04a] has linear space only when the abstract dimensionality is assumed to be constant with respect to the dataset size. As we observe experimentally (see Section 4), it is common for the Karger-Ruhl dimension to grow with the dataset size, so this latter assumption seems unrealistic in practice. Our algorithms are simple

beygel@us.ibm.com, sham@gatsby.ucl.ac.uk, jl@tti-c.org.

since the data structure being manipulated is a tree — in fact, our data structure (as a graph) is a subgraph of a navigating net [KL04a] (with slightly different parameters). We also provide experiments (and public code [Lan04]) suggesting this approach may be competitive with current practical approaches.

1.1. **Intrinsic Dimensionality.** We consider both notions of intrinsic dimensionality, proposed by Karger and Ruhl [KR02] and Krauthgamer and Lee [KL04a]. As usual, we denote the closed ball of radius r around p in $S \subset X$ by $B_S(p,r) = \{q \in S : d(p,q) \leq r\}$, and when clear from context, we just write $B(p,r)$.

Karger and Ruhl [KR02] consider classes of metrics that satisfy a growth bound. The *expansion constant* of S is defined as the smallest value $c \geq 2$ such that $|B_S(p,2r)| \leq c|B_S(p,r)|$ for every $p \in X$ and $r > 0$. If S is arranged uniformly on some surface of dimension d, then $c \sim 2^d$, which suggests defining the expansion dimension of S as $\dim_{\text{KR}}(S) = \log c$. However, as previously observed in [KR02, KL04a], some metrics that should intuitively be considered low-dimensional turn out to have large growth constants. For example, adding a single point in a Euclidean space may make the KR-dimension grow arbitrarily.

The *doubling constant* (defined in [KL04a] and motivated by the theory of analysis on metric spaces) provides a more robust alternative. The doubling constant is the minimum value c such that every ball in X can be covered by c balls in X of half the radius. The doubling dimension of S is then defined as $\dim_{\text{KL}}(S) = \log c$. This notion of dimensionality is strictly more general than the KR dimension, as shown in [GKL03]. Furthermore, the doubling dimension of k-dimensional Euclidean spaces is $\Theta(k)$. A drawback (so far) of working with the doubling dimension is that only weaker results have been provable, and even those apply only to approximate nearest neighbors. The algorithm in [KL04a] depends on the aspect ratio (the ratio of the largest to the smallest interpoint distance). Although the scheme in [KL04b] eliminates this dependence and makes the space consumption $n^{O(1)}$, independent of the doubling dimension (while the algorithm in [KL04a] is exponential in the dimension), no linear-space algorithm is known even in the case when the dimension is constant.

1.2. **New Algorithmic Results.** We present the cover tree data structure and algorithms for using it. First, let us look at how our algorithm fares when *no* dimensionality assumption is made. For this case, the query time is $\Omega(n)$.

	Cover Tree	Ball Tree	Nav. Net	[KR02]
Construction Space	$O(n)$	$O(n)$	$O(n^2)$	$O(n^2)$
Construction Time	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(n^2)$

In our analysis, we focus primarily on the expansion constant, because this permits results on exact nearest neighbor queries. If c is the expansion constant of S, then we can state the dependence on c explicitly:

	Cover Tree	Nav. Net	[KR02]
Construction Space	$O(n)$	$c^{O(1)}n$	$c^{O(1)}n \ln n$
Construction Time	$O(c^6 n \ln n)$	$c^{O(1)}n \ln n$	$c^{O(1)}n \ln n$
Insertion/Removal	$O(c^6 \ln n)$	$c^{O(1)} \ln n$	$c^{O(1)} \ln n$
Query	$O(c^{12} \ln n)$	$c^{O(1)} \ln n$	$c^{O(1)} \ln n$
Batch Query	$O(c^{16} n)$	$c^{O(1)}n \ln n$	$c^{O(1)}n \ln n$

It is important to note that the algorithms here (as in [KL04a] but not [KR02]) work for arbitrary metrics (with no assumptions on the structure); only the analysis is done with respect to the assumptions.

The main advantage of our data structure is its linear space bound independent of any assumptions made about the dataset. The algorithms are also simple since the structure they manipulate is a tree. Comparison of time complexity in terms c can be subtle (see the discussion at the end of Section 3.1). Also, such a comparison is somewhat unfair since past work did not explicitly try to optimize the c dependence. We make the dependence precise because further progress must optimize it.

The algorithms easily extend to approximate nearest neighbor queries for sets with a bounded doubling dimension, as in [KL04a]. The query times of both algorithms are $O(\log \Delta) + (1/\epsilon)^{O(1)}$, where Δ is the

aspect ratio and ϵ is the approximation parameter. Note that, unlike the navigating net, our space is $O(n)$ independent of the doubling dimension.

Finally, we provide several algorithms of practical interest. These include a lazy construction (which amortizes the construction cost over queries), a batch construction (which is empirically superior to a sequence of single point insertions), and a batch query (which amortizes the query time over multiple queries).

1.3. **Experimental Results.** We experimentally compared our algorithm to an optimized brute force search and to the $sb(S)$ algorithm [Cla02] on a number of benchmark machine learning datasets. This appears to be the first such empirical study looking directly at the viability of algorithms based on intrinsic dimensionality. Figure 4.1(a) shows speedups ranging from a factor of 1 to 2000, depending on the dataset. For fairness, we (1) report every dataset tested, (2) include the time to construct the cover tree. See Section 4 for details.

2. Constructing and Using a Cover Tree

2.1. **The Cover Tree Datastructure.** A *cover tree* T on a data set S is a leveled tree where each level is a "cover" for the level beneath it. Each level is indexed by an integer scale i which decreases as the tree is descended. Let C_i denote the set of nodes at level i.

A cover tree T on a dataset S obeys the following invariants for all i:

(1) (nesting) $C_i \subset C_{i-1}$.
(2) (covering tree) For every $p \in C_{i-1}$, there exists a $q \in C_i$ satisfying $d(p,q) \leq 2^i$, and exactly one such q is a parent of p.
(3) (separation) For all $p, q \in C_i$, $d(p,q) > 2^i$.

These invariants are essentially the same as used in navigating nets [KL04a], except for (2) where we require only one parent of a node rather than all possible parents. (For every node in C_{i-1}, a navigating net keeps all nodes in C_i that are within distance $\gamma 2^i$, where $\gamma \geq 4$ is some constant.) Despite potentially throwing out most of the links in a navigating net, all runtime properties can be maintained.

We make use of an implicit and an explicit representation of the cover tree. The *implicit* representation consists of the (infinitely many) levels C_i and the pointers from every node to its children in the level beneath it. The level C_∞ consists of a single node, called the *root* of the tree. It is simplest to describe the algorithms in terms of this implicit representation. However, we must use and analyze the *explicit* representation, which takes only $O(n)$ space. First, note that if a point p first appears in level i then it is in all levels below i (and, as the following proof shows, p is a child of itself in all of these levels). The explicit representation of the tree coalesces all nodes in which the only child is a self-child. This implies that every explicit node either has a parent other than the self-parent or a child other than the self-child, which immediately gives an $O(n)$ space bound, independent of the growth constant c.

Theorem 2.1. *(Space bound) A cover tree requires space at most $O(n)$.*

Proof. Every *point* has at most one parent other than itself in the explicit tree. To see this, assume $q \neq p$ and $q' \neq p$ are two parents of p. The scale at which q and q' are parents must be different by the covering tree invariant. Nesting implies that p is a sibling of the parent at some lower scale j. If q' is the parent at the lower scale, then separation implies $d(p,q') > 2^j$ which implies that q' can not be a parent at scale j.

Every time a point is a parent of itself, it also has another point as a child. Consequently, there are at most $O(n)$ links and n points implying the space bound. □

2.2. **Single Point Operations.** We now present the basic algorithms for cover trees and prove their correctness. The runtime analysis is given in Section 3.

2.2.1. *Finding the nearest neighbor of a point.* To find the nearest neighbor of a point p in a cover tree, we descend through the tree level by level, keeping track of a subset $Q_i \subset C_i$ of nodes that may contain the nearest neighbor of p as a descendant. The algorithm iteratively constructs Q_{i-1} by expanding Q_i to its children in C_{i-1} then throwing away any child q that cannot lead to the nearest neighbor of p.

Algorithm 1 Find-Nearest (cover tree T, query point p)

(1) set $Q_\infty = C_\infty$.
(2) for i from ∞ down to $-\infty$
  (a) consider the set of children of Q_i:
    $$Q = \{ \text{Children}(q) : q \in Q_i\}.$$
  (b) form next cover set:
    $$Q_{i-1} = \{q \in Q : d(p,q) \le d(p,Q) + 2^i\}$$
(3) return $\arg\min_{q \in Q_{-\infty}} d(p,q)$.

For simplicity of exposition, it is easier to think of the tree as having an infinite number of levels (with C_∞ containing only the root of the tree, and with $C_{-\infty} = S$). In what follows, let Children(p) be the set of children of node p and let $d(p,Q) = \min_{q \in Q} d(p,q)$ be the distance to the nearest point in a set Q.

Note that although the algorithm is stated using an infinite loop over the implicit representation, it only needs to operate on the explicit representation.

Theorem 2.2. *If T is a cover tree on S, then* **Find-Nearest**(T,p) *returns the nearest neighbor of p in S.*

Proof. For any q in C_{i-1} the distance between q and any descendant q' is bounded by $d(q,q') \le \sum_{j=i-1}^{-\infty} 2^j = 2^i$. Consequently, step 2(b) can never throw out a grandparent of the nearest neighbor of p. Eventually, there are no descendants of Q_i not in Q_i, and the nearest neighbor must be in Q_i. □

This algorithm can be used for proximity searches in a more general sense. A function Bound defined on finite sets is said to be *monotonic* if for any sets A and B, Bound$(B) \ge$ Bound(A) whenever $B \subset A$.

Example 2.3. (ϵ-nearest neighbors) Given a point $p \in S$ and $\epsilon > 0$, find the set of all points in $B(p,\epsilon)$. In this case, Bound$(A) = \epsilon$ for all A.

Example 2.4. (k-nearest neighbors) Given a point $p \in S$ and a number k, return the closest k points to p in S. In this case, Bound(A) is the distance to the kth closest point in A.

The algorithm can be modified as follows to work with any monotonic bound:

(1) In Step 3, find the subset of points of interest in Q by brute force.
(2) In Step 2(b), substitute Bound(Q) for $d(p,Q)$.

2.2.2. *Approximating the nearest neighbor of a point.* The Cover Tree can also be used to approximate nearest neighbors. Given a point $p \in X$ and some $\epsilon > 0$, we want to find a point $q \in S$ satisfying $d(p,q) < (1+\epsilon)d(p,S)$. The main idea is to maintain a lower bound as well as an upper bound, stopping when the interval implied by the bounds is sufficiently small. When analyzed with respect to the doubling constant, the proof of the time bound is essentially the same as in [KL04a]. Moreover, the space bound is now linear (independent of the doubling constant), giving a strict improvement over the results in [KL04a].

Algorithm: The only change is in line 2, where instead of descending the tree until no point in Q_i is explicit, we stop as soon as $2^{i+1}(1+1/\epsilon) \le d(p,Q_i)$.

Proof of correctness: Suppose that the descent terminated in level i. Then either $2^{i+1}(1+1/\epsilon) \le d(p,Q_i)$ or all points in Q_i are implicit (in which case we actually return the exact nearest neighbor). Let us consider the former case. Since Q_i is at distance at most 2^{i+1} from the exact nearest neighbor of p (Theorem 2.2), and d satisfies the triangle inequality, we have $d(p,Q_i) \le d(p,S) + 2^{i+1}$. Combining with $2^{i+1}(1+1/\epsilon) \le d(p,Q_i)$, this gives $2^{i+1}(1+1/\epsilon) \le d(p,S) + 2^{i+1}$, or $2^{i+1} \le \epsilon d(p,S)$. Hence, we have $d(p,Q_i) \le (1+\epsilon)d(p,S)$. □

The time complexity follows from inspection of Lemma 2.6 in [KL04a]. In particular, an approximate query takes at most $c^{O(1)} \log \Delta + (1/\epsilon)^{O(\log c)}$, where c is the doubling constant and Δ is the aspect ratio.

2.2.3. *Single Point Insertion.* The insertion algorithm (algorithm 2) is similar to the find nearest neighbor algorithm although the algorithm is stated recursively. Here, Q_i is a subset of the points at level i, which may contain p as a descendant (after insertion). The algorithm starts with $Q_\infty = C_\infty$, the root node of T.

Algorithm 2 Insert(point p, cover set Q_i, level i)

(1) set $Q = \{\text{Children}(q) : q \in Q_i\}$.
(2) if $d(p, Q) > 2^i$ then return "no parent found"
(3) else
  (a) set $Q_{i-1} = \{q \in Q : d(p, q) \leq 2^i\}$.
  (b) if **Insert**(p, Q_{i-1}, $i-1$) = "no parent found" and $d(p, Q_i) \leq 2^i$
    (i) pick $q \in Q_i$ satisfying $d(p, q) \leq 2^i$.
    (ii) insert p into Children(q).
    (iii) return "parent found"
  (c) else return "no parent found"

Algorithm 3 Remove(point p, cover sets $\{Q_i, Q_{i+1}, ..., Q_\infty\}$, level i)

(1) set $Q = \{\text{Children}(q) : q \in Q_i\}$
(2) set $Q_{i-1} = \{q \in Q : d(p, q) \leq 2^i\}$
(3) **Remove**(T, p, $\{Q_{i-1}, Q_i, ..., Q_\infty\}$, $i-1$)
(4) if $d(p, Q) = 0$ then
  (a) remove p from C_{i-1}
  (b) Remove p from Children(Parent(p))
  (c) for every $q \in$ Children(p)
    (i) set $i' = i - 1$.
    (ii) while $d(q, Q_{i'}) > 2^{i'}$
      (A) insert q into $C_{i'}$ (and $Q_{i'}$)
      (B) set $i' = i' + 1$
    (iii) pick $q' \in Q_{i'}$ satisfying $d(q, q') \leq 2^{i'}$
    (iv) make q' point to q

The proof of correctness implicitly shows that the datastructure always exists.

Theorem 2.5. *If T is a cover tree on S, then* **Insert**(p, C_∞, ∞) *returns a cover tree on $S \cup \{p\}$.*

Proof. Let us prove that the algorithm is guaranteed to insert any p not already contained in the cover tree. (If p is in the tree, this can be determined with a single invocation of the search procedure.) The set Q starts non-empty. Since p is not already in the tree, $d(p, S)$ is nonzero, and the condition in line 2 must eventually hold. Since the root is has scale ∞, there will be some minimal scale i between ∞ and the scale where line 2 first holds such that $d(p, Q_i) \leq 2^i$ and so 3.(b) holds.

We now prove that the insertion maintains all the cover tree invariants. If p is inserted in level $i-1$, we know that $d(p, Q_i) \leq 2^i$, and thus we can always find a parent $q \in Q_i$ with $d(p, q) \leq 2^i$, satisfying the covering tree invariant. Once p is inserted in level $i-1$, it is implicitly inserted in every level beneath it (as a child of itself in the previous level), maintaining the nesting invariant. Next we show that doing so does not violate the separation condition in lower levels.

To prove the separation condition in level $i-1$, consider $q \in C_{i-1}$. If $q \in Q$, then $d(p, q) > 2^{i-1}$. If $q \notin Q$, then at some iteration $i' > i$, some parent of q, say $q' \in C_{i'-1}$, was eliminated (in Step 3a), which implies that $d(p, q') > 2^{i'}$. Using the covering tree invariant at level j we have

$$d(p, q) \quad \geq \quad d(p, q') - \sum_{j=i'-1}^{i} 2^j = d(p, q') - (2^{i'} - 2^i) = 2^{i'} - (2^{i'} - 2^i) = 2^i,$$

which proves the desired separation $d(p, C_{i-1}) > 2^{i-1}$. Separation at levels below is proved similarly. □

2.2.4. *Single Point Removal.* The removal (Algorithm 3) is similar to insertion, with extra complexity due to coping with children of removed nodes.

Theorem 2.6. *Given a cover tree T on S,* **Remove**(p, C_∞, ∞) *returns a cover tree on $S - \{p\}$.*

Algorithm 4 Construct(point p, point sets $\langle \text{NEAR}, \text{FAR} \rangle$, level i)

(1) if NEAR $= \emptyset$
(2) then return $\langle p, \text{FAR} \rangle$
(3) else
  (a) $\langle$SELF, NEAR$\rangle$= **Construct** $(p, \textbf{SPLIT}(d(p,\cdot), 2^{i-1}, \textbf{NEAR}), i-1)$
  (b) add SELF to Children(p_i)
  (c) while NEAR $\neq \emptyset$
    (i) pick q in NEAR
    (ii) $\langle$CHILD, UNUSED$\rangle$ = **Construct** $(q, \textbf{SPLIT}(d(q,\cdot), 2^{i-1}, \textbf{NEAR,FAR}), i-1)$
    (iii) add CHILD to Children(p_i)
    (iv) let $\langle$NEW-NEAR, NEW-FAR$\rangle$ =SPLIT$(d(p,\cdot), 2^i,$ UNUSED$)$
    (v) add NEW-FAR to FAR, and NEW-NEAR to NEAR.
  (d) return $\langle p_i, \text{FAR} \rangle$.

Proof. As before, sets Q_i maintain points in level i closest to p, as we descend through the tree decrementing i. The recursion stops when it reaches the level below which p is always implicit.

For each level i explicitly containing p, we remove p from C_i and from the list of children of its parent in C_{i+1}. This does not disturb the nesting and the separation invariants. For each child q of p (by this time p has already been removed from the list of its children), we go up the tree looking for a new parent. More precisely, if there exists a node $q' \in C_i$ such that $d(q,q') \leq 2^i$ we make q' a parent of q; otherwise, we insert q in level C_i and repeat, propagating q up the tree until a parent is found. Insertion does not violate the separation and the nesting constraints, since $d(q, C_i) > 2^i$ (otherwise we would not be inserting q in C_i). This propagation process is guaranteed to terminate since q is covered by the root (at the scale of the root). Hence the covering tree invariant is enforced for all children of p. □

2.3. **Batch and Lazy Variants.** In this section, we present batch and lazy variants of practical interest.

2.3.1. *Batch Construction.* It is common to start with some large dataset (instead of receiving points one by one in an online manner). It is thus natural to try to amortize the construction cost over points. We now present such a batch algorithm. The analysis is given in the next section. Theoretically, the algorithm has the same guarantees as a series of single point insertions; however, preliminary empirical studies show that it is considerably faster (roughly twice as fast, although a controlled comparison has yet to be done).

At each step in the recursive construction, the algorithm has a point p, a set of points NEAR which *must* be inserted beneath p, and a set of points FAR which *might* be inserted beneath p. The algorithm first finds the near and far sets for itself (as a child), and then does the same for the remaining elements of NEAR. It then returns the created node, together with any unused elements of FAR.

To describe the algorithm, we need a helper function, SPLIT$(d(p,\cdot), r, S_1, S_2, ...)$ which splits the elements of $S_1, S_2, ...$ into points satisfying $d(p,q) \leq r$ and points satisfying $2r > d(p,q) > r$. All such points are removed from $S_1, S_2, ...$ The batch construction algorithm (algorithm 4) starts with a call to **Construct**$(p \in S, S - \{p\}, \emptyset)$**.** The proof is removed due to space limitations (see [BKL04]).

Theorem 2.7. (*Batch Correctness*) **Construct** $(p \in S, S - \{p\}, \emptyset)$ *returns a valid cover tree for* S.

Proof. Nesting holds because we explicitly construct the self-child of a point. Covering holds because all children of p are always in the near set and thus at distance at most 2^i. To show separation at level i, note that for any points u, v satisfying $d(u,v) \leq 2^i$, the procedure is first called on either u or v. Assume without loss of generality that it is called on u. Then v is in NEAR (otherwise it would already be in the tree contradicting the assumption that u came first), and the algorithm makes v either a child or a grandchild of u at steps 3(b) or 3(d)(iii). □

2.3.2. *Finding nearest neighbors of many points.* We can modify the previous query algorithm to simultaneously find the nearest neighbors of multiple points (similar to a method used in KD-trees [GM00]). The set of query points is given by a cover tree (which can be done by preprocessing the query set). This allows the descent of the search tree to be amortized over all queries improving the time complexity from

Algorithm 5 Find-All-Nearest (query cover tree p_j, cover set Q_i)

(1) if $i = -\infty$ then for each $a \in L(p_j)$
  return $\arg\min_{b \in Q_{-\infty}} d(a, b)$ as the nearest neighbor of a.
(2) else
  (a) if $j < i$ then
    (i) Set $Q = \{$ Children$(q) : q \in Q_i\}$.
    (ii) Set $Q_{i-1} = \{q \in Q : d(p_j, q) \leq \min_{q \in Q} d(p_j, q) + 2^i + 2^{j+2}\}$.
    (iii) **Find-All-Nearest (p_j, Q_{i-1})**
  (b) else for each $q_{j-1} \in$ Children(p_j)
    Find-All-Nearest (q_{j-1}, Q_i)

$O(c^{12} n \ln n)$ to $O(c^{16} n)$. In Algorithm 5, we abuse notation by letting p denote the subtree of the query tree rooted at node p. Since a point can appear in different levels, a subscript disambiguates as necessary. $L(p)$ denotes the set of leaves of the explicit part of the p subtree. The procedure is recursive, starting with the root of the query cover tree and C_∞. The correctness argument is similar to Theorem 2.2.

2.3.3. *Lazy Construction.* When few queries are done, the overhead of the construction may be greater than the brute force query cost. This drawback disappears with lazy construction. The basic idea is to make the nodes of a cover tree be *queries* modified to hold a list of points. Now every point p is in the list of node q at some level i must satisfy the following invariants:

(1) (closeness) $d(p, q) \leq 2^i$.
(2) (minimal level) There does not exist $q' \neq q$ with $d(p, q') \leq 2^{i-1}$.

Note, in particular, that we do not impose any separation constraint on nodes in the same layer.

The cover tree over completed queries is constructed using a variant of a single point insertion (see Section 2.2.3) modified to satisfy the above invariants. We only need the following modifications:

(1) For every q_i in Q_i, at every level i, compute (and record) the distance to every point attached to q_i.
(2) At the insertion level (say level i) attach every point within distance 2^i from level $i' > i$. Remove all newly attached points from higher level lists.

Note that levels for q which might have been implicit in the cover tree may not necessarily be implicit in the lazy cover tree. This is actually desirable because it organizes points relative to q in a finer manner.

Theorem 2.8. (*Correctness*) *The above modifications preserve the lazy cover tree invariants.*

Proof. Closeness is preserved because we explicitly only attach points within distance 2^i. Minimal level is preserved because we only attach points from higher levels. □

Queries on a lazy cover tree are exactly the same as on a cover tree except that every point in the list of every node is treated as a leaf. Several observations are in order.

(1) If every point is queried, the resulting cover tree is exactly the same as would be constructed using single point insertion.
(2) The set of distance calculations required for insertion is a subset of the set required for a query. This holds because the distance constraint of step 3(a) in **Insert** is stronger than the distance constraint in step 2(b) of **Find-Nearest**. Consequently, the query can fully amortize the insertion.

3. The Runtime Analysis

In this section, the distinction between implicit and explicit representation (see Section 2.1) is important.

3.1. **Query analysis.** We start with three lemmas about some structural properties of the data structure.

Lemma 3.1. *(Width bound) The number of children of any node p is bounded by c^4.*

Proof. Let p be in level i. The number of its children is at most $|B(p, 2^i) \cap C_{i-1}|$, which is certainly bounded by $|B(p, 2^{i+1}) \cap C_{i-1}|$. The idea of the proof is to bound the number of disjoint balls of radius 2^{i-2} that we can pack into $B(p, 2^{i+1})$. Each of these balls can cover at most one point in C_{i-1}, thereby bounding the number of children. For any child q of p, since $d(p, q) \leq 2^i$, we have $B(p, 2^{i+1}) \subset B(q, 2^{i+2})$ implying

$$|B(p, 2^{i+1})| \leq |B(q, 2^{i+2})| \leq c^4 |B(q, 2^{i-2})|.$$

The balls $B(q, 2^{i-2})$ must be disjoint for all $q \in C_{i-1}$, since the points in C_{i-1} are at least 2^{i-1} apart. We also know that each $B(q, 2^{i-2})$ is contained within $B(p, 2^{i+1})$, since $d(p, q) \leq 2^i$. Then the number of disjoint balls around the children that can be packed into $B(p, 2^{i+1})$ is bounded by

$$|B(p, 2^i) \cap C_{i-1}| \leq \frac{|B(p, 2^{i+1})|}{|B(q, 2^{i-2})|} \leq c^4,$$

which gives a bound on the number of children of p. □

The following lemma is useful in bounding the depth of the tree. It says that if there is a point in some annulus centered around p, then the volume growth of a sufficiently large ball around p containing the annulus is non-trivial. In other words, it gives a lower bound on the volume growth in terms of the growth constant c, while the definition of c gives an upper bound.

Lemma 3.2. (*Growth Bound*) *For all points $p \in S$ and $r > 0$, if there exists a point $q \in S$ such that $2r < d(p, q) \leq 3r$, then*

$$|B(p, 4r)| \geq \left(1 + \frac{1}{c^2}\right) |B(p, r)|.$$

Proof. Since $B(p, r) \subset B(q, 3r + r)$, we have

$$|B(p, r)| \leq |B(q, 4r)| \leq c^2 |B(q, r)|.$$

And since the balls $B(p, r)$ and $B(q, r)$ are disjoint and are subsets of $B(p, 4r)$, we have

$$|B(p, 4r)| \geq |B(p, r)| + |B(q, r)|.$$

The result follows by combining these inequalities. □

Using this, we can prove a bound on the *explicit* depth of any point p, defined as the number of explicit grandparent nodes on the path from the root to p in the lowest level in which p is explicit.

Lemma 3.3. (*Depth Bound*) *The maximum depth of any point p in the explicit representation is $O\left(c^2 \log n\right)$.*

Proof. Define $S_i = \{q \in S : 2^{i+1} \leq d(p, q) < 2^{i+2}\}$. First let us show that if point $q \in S_i$ is a grandparent of p, then $q \in C_i$. If $q \in C_j$ for some j, then any of its grandchildren is at most 2^{j+1} away implying $j \geq i$. Nesting says that $q \in C_i$, since $C_j \subset C_i$.

Now let us consider the grandparents of p in levels C_i, C_{i+1}, C_{i+2}, C_{i+3}. There are at most four of these, due to the tree property. In fact, there can be no other unique grandparents above level $i + 3$ in S_i. Recall that if $q \in S_i$, then $d(p, q) < 2^{i+2}$. If q is also in C_{i+3}, the well-separateness constraint implies that there can be no other point in S_i which is also in C_{i+3}. Nesting implies that there are no other grandparents in $j > i + 3$, else these grandparents would also be in C_{i+3}.

Thus any annulus S_i can only contain unique grandparents of p up to level $i + 3$. Now we just need to bound the number of non-empty S_i around p containing all points in S. To do this, apply the growth bound with $r = \frac{d(p,q)}{2}$ where q is the nearest neighbor of p to discover $|B(p, 4r)| \geq \left(1 + \frac{1}{c^2}\right) |B(p, r)| = \left(1 + \frac{1}{c^2}\right)$. Then, find the next nearest point q satisfying $d(p, q) \geq 8r$, and apply the growth bound with $r' = \frac{d(p,q)}{2}$ to discover $|B(p, 4r)| \geq \left(1 + \frac{1}{c^2}\right)^2$ since each application of the growth bound is disjoint (note that this process may significantly undercount points). This process can be repeated at most $\frac{\log n}{\log(1+1/c^2)}$ before the lower bound exceeds the upper bound of n. Upon termination, every point q can be associated with the maximal r satisfying $2r \leq d(p, q)$. The set of points associated with every step in the process lie in at most 4 annuli S_i. Consequently, there are at most $O\left(\frac{\log n}{\log(1+1/c^2)}\right)$ nonempty annuli around any p. This is $O(c^2 \log n)$ since $c \geq 2$. The number of explicit grandparents in S_i is constant, completing the proof. □

We can now state and prove the main theorem.

Theorem 3.4. (*Query Time*) *If the dataset $S \cup \{p\}$ has expansion constant c, the nearest neighbor of p can be found in time $O\left(c^{12} \log n\right)$.*

Proof. Let Q^* be the last explicit Q_i considered by the algorithm. Lemma 3.3 bounds the explicit depth of any point in the tree (and in particular any point in Q^*) by $k = O\left(c^2 \log n\right)$. Consequently, the number of iterations is at most $k|Q^*| \leq k \cdot \max_i |Q_i|$. In each iteration, at most $O(\max_i |Q_i|)$ time is required to determine which elements need explicit descent, implying a bound of $O(k \max_i |Q_i|^2)$ on the query time.

Also note that in Step 2(a), the number of children encountered is at most $kc^4 \max_i |Q_i|$ using Lemma 3.1. Step 2(b) never does more work than Step 2(a). Step 3 requires at most $\max_i |Q_i|$ work. Consequently, the running time is bounded by $O(k \max_i |Q_i|^2 + k \max_i |Q_i| c^4))$ finishing the proof, provided that we can show that $\max_i |Q_i| \leq c^5$.

Consider any Q_{i-1} constructed during the i-th iteration. Recall that $Q = \{\text{ Children}(q) : q \in Q_i\}$, and let $d = d(p, Q)$. We have

$$Q_{i-1} \quad = \quad \{q \in Q : d(p, q) \leq d + 2^i\} = B(p, d + 2^i) \cap Q \subseteq B(p, d + 2^i) \cap C_{i-1},$$

where the first equality follows by definition of Q_{i-1} and the second from $Q \subseteq C_{i-1}$.

First suppose that $d > 2^{i+1}$. Then we have

$$|B(p, d + 2^i)| \quad \leq \quad |B(p, 2d)| \leq c^2 \left| B\left(p, \frac{d}{2}\right) \right|.$$

Now since $d \leq d(p, S) + 2^i$ (as a consequence of $Q \subseteq C_{i-1}$), and $d > 2^{i+1}$(by assumption), we also have $d(p, S) \geq d - 2^i > 2^i$. Hence $B\left(p, \frac{d}{2}\right) = \{p\}$, and $|Q_{i-1}| \leq c^2$.

We are left with the case $d \leq 2^{i+1}$. Consider a point $q \in C_{i-1}$ which is also in $B(p, d + 2^i)$. As in the proof of Lemma 3.1, we will bound the number of disjoint balls of radius 2^{i-2} that can be packed into $B(p, d + 2^i + 2^{i-2})$. Any such ball can contain at most one point in C_{i-1} (due to the separation constraint), implying a bound on $|Q_{i-1}|$. We have

$$|B(p, d + 2^i + 2^{i-2})| \leq |B(q, 2(d + 2^i) + 2^{i-2})| \leq$$

$$|B(q, 2^{i+2} + 2^{i+1} + 2^{i-2})| \leq |B(q, 2^{i+3})| \leq c^5 |B(q, 2^{i-2})|,$$

and thus $|Q_{i-1}| \leq |B(p, d + 2^i) \cap C_{i-1}| \leq c^5$. □

Comparing the time complexity of navigating nets and cover trees in terms of its dependence on the expansion constant is non-trivial. Our data structure does run-time computations which were done in the preprocessing stage of the navigating nets algorithm. Navigating nets can be run in a more greedy (depth first search) mode, while cover trees use a from of a fused depth and breadth first search. The tradeoff is even more subtle because the radius of the balls used to form the covers in the navigating nets is larger than the radius used in the cover tree, implying that a node may have to maintain more children.

3.2. **Insertion and Removal.**

Theorem 3.5. *Any insertion or removal takes time at most $O\left(c^6 \log n\right)$.*

Proof. First we show that all but one node in each cover set are either expanded to their children or removed in the next two cover sets. To see why, note that each Q_i is contained in a ball of radius 2^{i+1} around the point p we are inserting (by definition). Fix i and assume that some node q appears (either explicitly or implicitly) in all of Q_i, Q_{i-1}, Q_{i-2}. Then no other node $q' \in Q_i$ can appear in Q_{i-2}, since the separation constraint in level i says that $d(q, q') > 2^i$ while the maximum distance between $q \in Q_{i-2}$ and any other node in Q_{i-2} can be at most 2^i. Thus q is either removed or expanded to its children, in which case it has to consume one level of its explicit depth.

Let $k = c^2 \log |S|$ be the maximum explicit depth of any point, given by Lemma 3.3. Then the total number of cover sets with explicit nodes is at most $3k + k = 4k$, where the first term follows from the fact that any node that is not removed must be explicit at least once every three iterations, and the additional k accounts for a single point that may stay implicit for more than three iterations.

Thus the total amount of work in Steps 1 and 2 is proportional to $O(k \max_i |Q_i|)$. Step 3 requires work no greater than step 1. For every i, Q_i is a valid set of children for a hypothetical node at level $i+1$, and thus $|Q_i| \leq c^4$ from Lemma 3.1. Multiplying these bounds together we get the result.

To obtain the bound for the removal, we can use a similar argument to show that at most one point can be propagated up more than twice in the search for a parent. Thus Step 5 in Algorithm 3 takes at most $O(k \max_i |Q_i|)$ steps total. Other steps require work no greater than for insertion, completing the proof. □

3.3. **Batch Operations.**

Theorem 3.6. (*Construction Time*) **Construct** $(p \in S, S - \{p\}, \emptyset)$ *requires time at most* $O(c^6 n \log n)$.

Proof. (sketch) The maximum amount of work in level i associated with point p is proportional to the number of siblings within distance $2^{i+2} + 2^{i+1}$ since all elements in level i doing work related to p must have a parent at level $i+1$, for which p is in the FAR set (and thus at distance 2^{i+2}). The maximum number of such siblings is c^4 using an argument similar to the growth bound. The maximum number of explicit levels which nearby points can have is $O(c^2 \log n)$ (similar to the insertion proof). Multiplying max sibling level * max siblings/level * number of points, we get O(c^{6}n\log n) work completing the proof. □

Theorem 3.7. (*Batch Query Time*) **Find-All-Nearest** (T, C_∞) *requires time* $O(c^{16} n)$.

Proof. (sketch) The proof is similar to the query proof, except that the cover set at level i has a radius increased by 2^{i+1} due to the need to the need to cover the nearest neighbor of grandchildren of the query cover tree. This increases the size of the cover set by a factor of c^2 which increases the c-complexity by c^4.

The dependence on n rather than $n \log n$ is due to amortizing the descent of the cover tree over the leaf nodes. In particular, Step 2(a)(i) is executed on each explicit node at most once. All other steps have the same or lower time complexity. □

4. Experimental Results

We tested the algorithm on several datasets drawn from the UCI machine learning and KDD archives [UCI], the KDD 2004 championship [KDDCup], the Mnist handwritten digit recognition dataset [mnist], and the Isomap "Images" dataset [isomap]. For each dataset, we queried for the {1,2,3,5,10}-nearest neighbors of each point using the Euclidean metric (results with an l_1 metric were qualitatively similar). The results, compared to an optimized brute force algorithm, are summarized in Figure 4.1(a).

We used the following optimizations of the basic algorithms to obtain the results in Figure 4.1(a):

(1) Similar to [Moo00], we cache summary information in each node; in particular, we cache the maximum distance to any grandchild, the distance to the parent, and the scale.
(2) Since the upper bound can be tightened as a cover set is expanded, the order of expansion is important. Ideally, we might expand from the closest to furthest on the theory that closer nodes are more likely to have closer children. We used an $O(n)$ lazy sorting which recurses only on the low direction of quicksort.
(3) Our theoretical results use scales that are powers of 2. It turns out that a smaller base provides speedups in practice.
(4) We relax the separation invariant in batch construction.

A natural question is whether the expansion constant is a relevant quantity for analysis. Since it is defined as the worst-case expansion over all points, it may not be the best measure of hardness of NNS. Figure 4.1(c) illustrates this point by showing two datasets on 5000 points each with the same worst-case expansion constant but different distributions of expansion across points, and not surprisingly, very different speedups. Figure 4.1(d) suggests that, for example, the 80th percentile (over datapoints) expansion constant seems to be a better predictor of performance. Figure 4.1(e) shows the speedups from Figure 4.1(a) (as stalactite impulses) against the respective normalized expansion constants and their 80% versions (stalagmite impulses). The 80th percentile (dashed stalagmites) are more predictive of the speedup.

Finally, we did experiments comparing cover trees to Clarkson's $sb(S)$ data structure [Cla02] developed for the same setting as ours (see also [Cla99]). For each dataset, we did exact nearest neighbor queries of every point using the "d" method in [Cla02] that was reported to be uniformly superior to all other methods

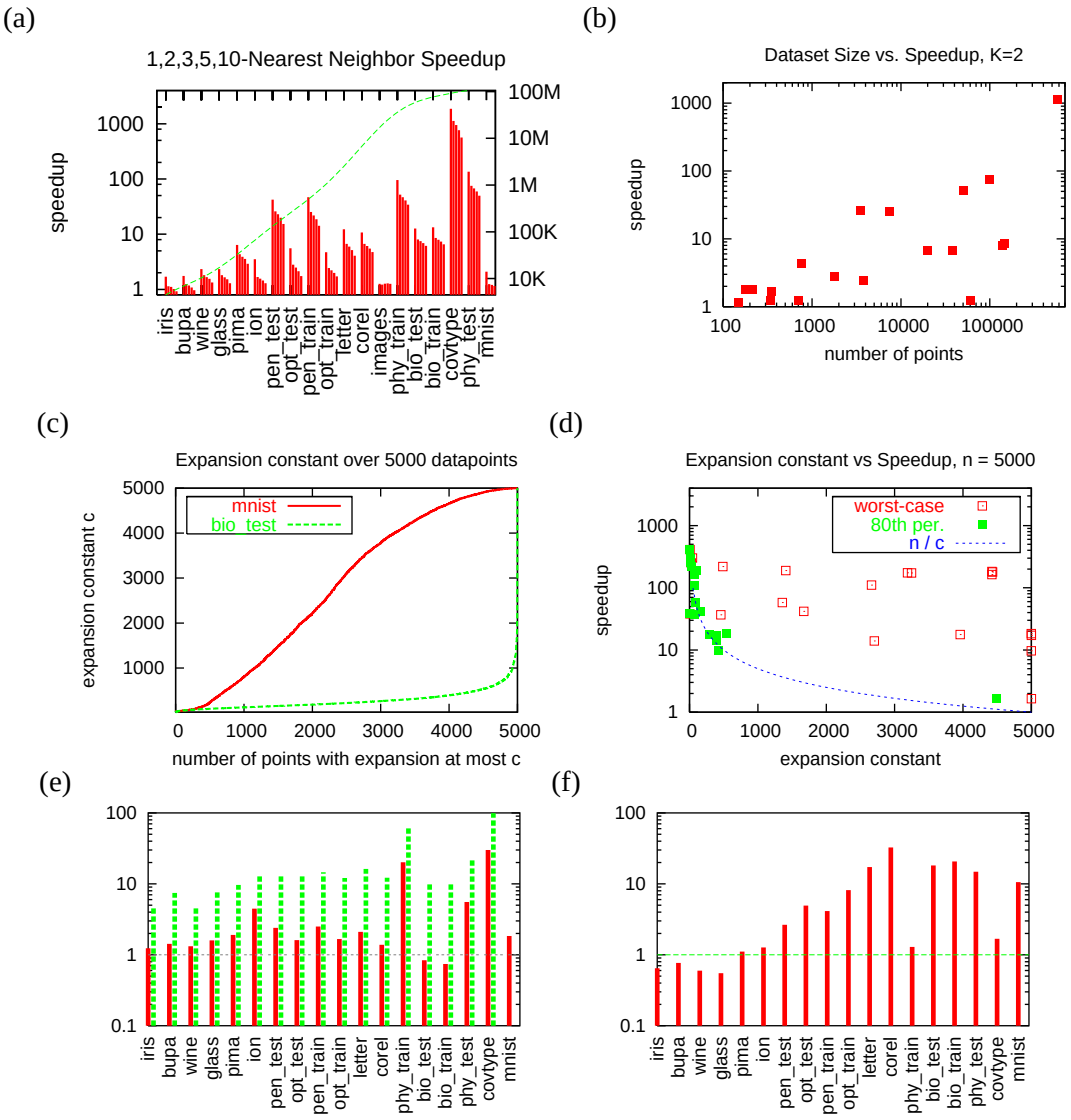

FIGURE 4.1. a) The speedup of the cover tree algorithm over the brute force search when querying for the nearest (1,2,3,5,10) neighbors (left to right in the graph) of every point in the dataset (logscale); 1-nearest neighbor query corresponds to self-search. Datasets are sorted by byte size in ascending order; the size is shown using a dashed line. (b) The speedup versus the number of data points in each dataset. Larger datasets have larger potential speedups. (c) The cumulative distribution of expansion constants across points for two datasets with the same maximum expansion. We achieve very little speedup on the 'mnist' dataset and about a factor of 10 speedup on the bio_test dataset. (d) A graph showing speedup vs the worst case expansion constant and speedup vs the 80th percentile (over datapoints) expansion constant on various 5000 point datasets obtained as prefixes of datasets form [UCI, KDDCup, mnist, isomap]. (e) The speedup (logscale) over the $sb(S)$ data structure [Cla02] for the (1,2) neighbors (solid and dashed lines respectively). One datapoint (images) is missing due to parsing issues with the $sb(S)$ code. (f) Construction times. The speedup over the $sb(S)$ data structure [Cla02]. Notice considerable savings in the construction times on the two datasets (bio_test and bio_train), on which $sb(S)$ outperforms the cover tree.

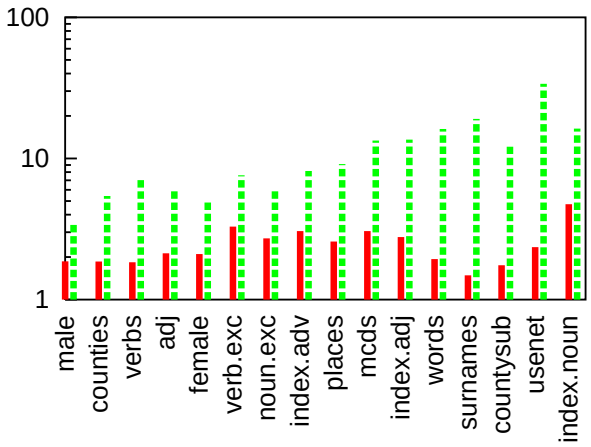

FIGURE 4.2. The speedup of the cover tree algorithm over the $sb(S)$ data structure [Cla02] when querying for the nearest neighbor of very point in the dataset (i.e., self-search); points are strings under edit distance. Dashed spikes show the corresponding speedup in the construction times. Datasets are sorted by byte size in ascending order, from 8K to 1.4M.

available in the $sb(S)$ package. We included the construction time when evaluating both algorithms and used the same timing mechanisms and the same implementation of the distance functions. Our algorithm was significantly faster on almost every dataset tested; the speedups (ranging from 0.8 to 30) are shown in Figure 4.1(e). The respective constuction times are shown in Figure 4.1(f). Notice considerable savings in the construction times of the cover tree on the onlu two datasets (bio_test and bio_train), on which $sb(S)$ is superior (for 1-nearest neighbor). It should be noted, however, that the k-nearest neighbor implementation in $sb(S)$ is via a reduction to fixed-radius queries; a better scheme might be possible, but it is not straightforward.

Figure 4.2 shows the speedup of the cover tree over $sb(S)$ for strings under the edit distance. For more details, the code, and the datasets see [Lan04, BKL04].

Acknowledgement. We would like to thank Piotr Indyk, David Karger, Robert Krauthgamer, James Lee, and Alexander Gray for helpful discussions and comments.

REFERENCES

[AMN+98] S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman and A. Wu. An optimal algorithm for approximate nearest neighbor searching, *Journal of the ACM*, 45(6) : 891–923, 1998.

[AM] S. Arya and D. M. Mount. ANN: Library for Approximate Nearest Neighbor Searching, http://www.cs.umd.edu/~mount/ANN/.

[BKL04] A. Beygelzimer, S. Kakade, J. Langford. Cover trees for nearest neighbor, preprint, 2004. Available at http://hunch.net/~jl/projects/cover_tree/cover_tree.html

[Cla99] K. Clarkson: Nearest Neighbor Queries in Metric Spaces. *Discrete & Computational Geometry*, 22(1): 63-93 (1999)

[Cla02] K. Clarkson: Nearest Neighbor Searching in Metric Space: Experimental Results for sb(S), 2002, http://cm.bell-labs.com/who/clarkson/Msb/readme.html.

[DIIM04] M. Datar, N. Immorlica, P. Indyk, and V. Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions, *Proceedings of the 20th Annual Symposium on Computational Geometry*, 2004.

[FBL77] J. H. Friedman, J. L. Bentley, and R. A. Finkel, "An algorithm for finding best matches in logarithmic expected time", *ACM Transactions on Mathematical Software*, 3(3):209-226, September 1977.

[GM00] A. Gray and A. Moore. N-Body Problems in Statistical Learning, *Proceedings of NIPS*, 2000.

[GKL03] A. Gupta, R. Krauthgamer, and J. R. Lee. Bounded geometries, fractals, and low-distortion embeddings, *Proceedings of the 44th Annual IEEE Symposium on Foundations of Computer Science*, 534–543, 2003.

[isomap] Isomap datasets, http://isomap.stanford.edu/datasets.html

[KR02] D. Karger and M. Ruhl. Finding Nearest Neighbors in Growth Restricted Metrics, *Proceedings of STOC*, 2002.

[KDDCup] The 2004 KDD-cup dataset, http://kodiak.cs.cornell.edu/kddcup/

[KL04b] R. Krauthgamer and J. Lee. The black-box complexity of nearest neighbor search, *Proceedings of the 31st International Colloquium on Automata, Languages and Programming (ICALP)*, 2004.

[KL04a] R. Krauthgamer and J. Lee. Navigating nets: Simple algorithms for proximity search, *Proceedings of the 15th Annual Symposium on Discrete Algorithms (SODA*), 791–801, 2004.

[Lan04] J. Langford, Cover Tree code written in C++, http://hunch.net/~jl/projects/cover_tree/cover_tree.html

[Moo00] A. Moore, Using the Triangle Inequality to Survive High Dimensional Data, Proceedings of the Twelfth Conference on Uncertainty in Artificial Intelligence, 2000.
[mnist] The MNIST database of handwritten digits, http://yann.lecun.com/exdb/mnist/
[Omo87] S. M. Omohundro, Efficient Algorithms with Neural Network Behavior. *J. of Complex Systems*, 1(2):273-347, 1987.
[RS00] Sam Roweis and Lawrence Saul, "Nonlinear dimensionality reduction by locally linear embedding", Science v. 290 no. 5500, Dec. 22, 2000. pp. 2323-2326.
[TSL00] Josh Tenenbaum, Vin de Silva and John Langford. A Global Geometric Framework for Nonlinear Dimensionality Reduction. Science 290, 2319-2323, 2000
[UCI] UCI machine learning repository (http://www.ics.uci.edu/~mlearn/) and KDD Archive (http://kdd.ics.uci.edu/).
[Uhl91] J. K. Uhlmann, Satisfying general proximity/similarity queries with metric trees. *Information Processing Letters*, 40:175-179, 1991.