2006

Cover Trees for Nearest Neighbor

Alina Beygelzimer

Sham M. Kakade
University of Pennsylvania

John Langford

Recommended Citation

Beygelzimer, A., Kakade, S. M., & Langford, J. (2006). Cover Trees for Nearest Neighbor. *Proceedings of the 23rd International Conference on Machine Learning*, 97-104. Retrieved from http://repository.upenn.edu/statistics_papers/121

Cover Trees for Nearest Neighbor

Abstract

We present a tree data structure for fast nearest neighbor operations in general n-point metric spaces (where the data set consists of n points). The data structure requires $O(n)$ space *regardless* of the metric's structure yet maintains all performance properties of a navigating net (Krauthgamer & Lee, 2004b). If the point set has a bounded expansion constant c, which is a measure of the intrinsic dimensionality, as defined in (Karger & Ruhl, 2002), the cover tree data structure can be constructed in $O\left(c^6 n \log n\right)$ time. Furthermore, nearest neighbor queries require time only logarithmic in n, in particular $O\left(c^{12} \log n\right)$ time. Our experimental results show speedups over the brute force search varying between one and several orders of magnitude on natural machine learning datasets.

Disciplines
Statistics and Probability

Cover Trees for Nearest Neighbor

Alina Beygelzimer   BEYGEL@US.IBM.COM
IBM Thomas J. Watson Research Center, Hawthorne, NY 10532

Sham Kakade   SHAM@TTI-C.ORG
TTI-Chicago, 1427 E 60th Street, Chicago, IL 60637

John Langford   JL@HUNCH.NET
TTI-Chicago, 1427 E 60th Street, Chicago, IL 60637

Abstract

We present a tree data structure for fast nearest neighbor operations in general n-point metric spaces (where the data set consists of n points). The data structure requires $O(n)$ space *regardless* of the metric's structure yet maintains all performance properties of a navigating net (Krauthgamer & Lee, 2004b). If the point set has a bounded expansion constant c, which is a measure of the intrinsic dimensionality, as defined in (Karger & Ruhl, 2002), the cover tree data structure can be constructed in $O\left(c^6 n \log n\right)$ time. Furthermore, nearest neighbor queries require time only logarithmic in n, in particular $O\left(c^{12} \log n\right)$ time. Our experimental results show speedups over the brute force search varying between one and several orders of magnitude on natural machine learning datasets.

1. Introduction

Problem. Nearest neighbor search is a basic computational tool that is particularly relevant to machine learning, where it is often believed that high-dimensional datasets have low-dimensional intrinsic structure. Here we study how one can exploit potential structure in the dataset to speed up nearest neighbor computations. Such speedups could benefit a number of machine learning algorithms, including dimensionality reduction algorithms (which are inherently based on this belief of low-dimensional

structure) and classification algorithms that rely on nearest neighbor operations (for example, (Laviolette et al., 2005)).

The basic nearest neighbor problem 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 $d(p, q)$.

Context. For general metrics, finding (or even approximating) the nearest neighbor of a point requires $\Omega(n)$ time. The classical example is a uniform metric where every pair of points is near the same distance, so there is no structure to take advantage of. However, the metrics of practical interest typically do have some structure which can be exploited to yield significant computational speedups. Motivated by this observation, several notions of metric structure and algorithms exploiting this structure have been proposed (Clarkson, 1999; Karger & Ruhl, 2002; Krauthgamer & Lee, 2004b).

Denote the closed ball of radius r around p in $S \subset X$ by $B_S(p, r) = \{q \in S : d(p, q) \le r\}$. When clear from the context, we drop the subscript S. Karger and Ruhl (Karger & Ruhl, 2002) considered the following notion of dimension based on point expansion, and described a randomized algorithm for metrics in which this dimension is small. The *expansion constant* of S is defined as the smallest value $c \ge 2$ such that $|B_S(p, 2r)| \le c |B_S(p, r)|$ 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 (also referred to as KR-dimension) as $\dim_{KR}(S) = \log c$. However, as previously observed in (Karger & Ruhl, 2002; Krauthgamer & Lee, 2004b), 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 (though such examples may be pathological in practice).

A more robust alternative is given by the *doubling constant* (Clarkson, 1999; Krauthgamer & Lee, 2004b), which 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 $\dim_{\mathrm{KL}}(S) = \log c$. This notion is strictly more general than the KR-dimension, as shown in (Gupta et al., 2003). 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 aforementioned algorithms have query time guarantees which are only logarithmic in n (while being exponential in their respective notion of intrinsic dimensionality). Unfortunately, in machine learning applications, most of these theoretically appealing algorithms are still not used in practice. When the Euclidean dimension is small, one typical approach is to use KD-trees (see (Friedman et al., 1977)). If the metric is non-Euclidean, or the Euclidean dimension is large, *ball trees* (Uhlmann, 1991; Omohundro, 1987) provide compelling performance in many practical applications (Gray & Moore, 2000). These methods currently have only trivial query time guarantees of $O(n)$, although improved performance may be provable given some form of structure.

The focus of this paper is to make these theoretically appealing algorithms more practically applicable. One significant drawback of these algorithms (based on intrinsic dimensionality notions) is that their space requirements are exponential in the dimension. As we observe experimentally (see Section 5), it is common for the dimension to grow with the dataset size, so space consumption is a reasonable concern. This drawback is precisely what the cover tree addresses.

New Results. We propose a simple data structure, a *cover tree*, for exact and approximate nearest neighbor operations. The data structure improves over other results (Karger & Ruhl, 2002; Krauthgamer & Lee, 2004b; Clarkson, 1999; Har-Peled & Mendel, 2006) by making the space requirement linear in the dataset size, *independent* of any dimensionality assumptions. The cover tree is simple since the data structure being manipulated is a tree; in fact, a cover tree (as a graph) can be viewed as a subgraph

of a navigating net (Krauthgamer & Lee, 2004b). The cover tree throws away most of the edges of the navigating net while maintaining all dimension-dependent guarantees. The algorithms and proofs needed for this structure are inherently different because (for example) a greedy traversal of the tree is not guaranteed to answer a query correctly. We also provide experiments (see Section 5) and public code, suggesting this approach is competitive with current practical approaches.

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, we can state the dependence on c explicitly. In the table below, KL and KR refer to (Krauthgamer & Lee, 2004b) and (Karger & Ruhl, 2002) respectively.

	Cover Tree	KL	KR
Constr. Space	$O(n)$	$c^{O(1)}n$	$c^{O(1)}n \ln n$
Constr. Time	$O(c^6 n \ln n)$	$c^{O(1)}n \ln n$	$c^{O(1)}n \ln n$
Insert/Remove	$O(c^6 \ln n)$	$c^{O(1)} \ln n$	$c^{O(1)} \ln n$
Query	$O(c^{12} \ln n)$	$c^{O(1)} \ln n$	$c^{O(1)} \ln n$

It is important to note that the algorithms here (as in (Krauthgamer & Lee, 2004b) but not in (Karger & Ruhl, 2002)) work without knowledge of the structure; only the analysis is done with respect to the assumptions. Comparison of time complexity in terms of c can be subtle (see the discussion in Section 4). Also, such a comparison is somewhat unfair since past work did not explicitly try to optimize the c dependence.

The algorithms easily extend to approximate nearest neighbor queries for sets with a bounded doubling dimension, as in (Krauthgamer & Lee, 2004b). The algorithm of (Krauthgamer & Lee, 2004b) depends on the aspect ratio Δ defined as the ratio of the largest to the smallest interpoint distance.[1] The query times of our algorithm are the same as those in (Krauthgamer & Lee, 2004b), namely $O(\log \Delta) + (1/\epsilon)^{O(1)}$, where ϵ is the approximation parameter.

In an extended version (Beygelzimer et al., 2005), 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

---

[1] The results in (Clarkson, 1999) also depend on this ratio and rely on some additional stronger assumptions about the distribution of queries. The algorithms in (Krauthgamer & Lee, 2004a) and (Har-Peled & Mendel, 2006) eliminate the dependence on the aspect ratio but do not achieve linear space.

single point insertions), and a batch query (which amortizes the query time over multiple queries).

Organization. The rest of the paper is organized as follows. Sections 2 and 3 specify the algorithms and prove their correctness, with *no* assumptions about any structure present in the data set. Section 4 provides the runtime analysis in terms of dimensionality. Section 5 presents experimental results.

2. 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. Every *node* in the tree is associated with a point in S. Each *point* in S may be associated with multiple nodes in the tree; however, we require that any point appears at most once in every level. Let C_i denote the set of points in S associated with the nodes at level i. The cover tree obeys the following invariants for all i:

1. (Nesting) $C_i \subset C_{i-1}$. This implies that once a point $p \in S$ appears in C_i then *every* lower level in the tree has a node associated with p.

2. (Covering tree) For every $p \in C_{i-1}$, there exists a $q \in C_i$ such that $d(p,q) < 2^i$ and the node in level i associated with q is a parent of the node in level $i - 1$ associated with p.

3. (Separation) For all distinct $p, q \in C_i$, $d(p,q) > 2^i$.

Important Note: With some abuse of terminology, we identify nodes with their associated points, with an understanding of the distinction made above. Since a point can appear in at most one node in the same level, no confusion can occur.

These invariants are essentially the same as used in navigating nets (Krauthgamer & Lee, 2004b), except for (2) where we require only one parent of a node rather than all possible parents. (For every node in level $i - 1$, a navigating net keeps pointers to all nodes in level 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.

It is conceptually easiest to describe the algorithms in terms of an *implicit* representation of the cover tree consisting of an infinite number of levels, with C_∞ containing the point in S associated with the root node and with $C_{-\infty} = S$. However, we must use and

analyze the *explicit* representation, which takes only $O(n)$ space. Recall that if a point $p \in S$ 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 (i.e., the node associated with p is a child of the node associated with p in one level above). 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 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. ∎

3. Single Point Operations

We now present the basic algorithms for cover trees and prove their correctness. The runtime analysis is given in Section 4.

3.1. Finding the Nearest Neighbor

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

Algorithm 1 <u>Find-Nearest</u> **(cover tree** T**, query point** p**)**

---

1. Set $Q_\infty = C_\infty$, where C_∞ is the root level of T.

2. for i from ∞ down to $-\infty$

   (a) Set $Q = \{$ Children$(q) : q \in Q_i\}$.
   (b) Form cover set $Q_{i-1} = \{q \in Q : d(p,q) \leq d(p,Q) + 2^i\}$.

3. return $\arg\min_{q \in Q_{-\infty}} d(p,q)$.

Theorem 2 *If T is a cover tree on S,* **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 $d(q,q') \leq \sum_{j=i-1}^{-\infty} 2^j = 2^i$. 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, so the nearest neighbor is in Q_i. ∎

3.2. Approximating the Nearest Neighbor

The cover tree structure 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 (Krauthgamer & Lee, 2004b). The space bound is now linear (independent of the doubling constant), giving a strict improvement over the results in (Krauthgamer & Lee, 2004b).

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

Proof of correctness: Suppose that the descent terminated in level i. Then either $2^{i+1}(1+1/\epsilon) \leq d(p,Q_i)$ or all nodes 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), and d satisfies the triangle inequality, we have $d(p,Q_i) \leq d(p,S) + 2^{i+1}$. Combining with $2^{i+1}(1+1/\epsilon) \leq d(p,Q_i)$, this gives $2^{i+1}(1+1/\epsilon) \leq d(p,S) + 2^{i+1}$, or $2^{i+1} \leq \epsilon d(p,S)$. Hence, we have $d(p,Q_i) \leq (1+\epsilon)d(p,S)$. ∎

The time complexity follows from inspection of Lemma 2.6 in (Krauthgamer & Lee, 2004b). 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.

3.3. Single Point Insertion

The insertion algorithm (Algorithm 2) is similar to the query algorithm but it is stated recursively. Here Q_i is a subset of the points at level i which may contain the new point p as a descendant. The algorithm starts with the root node, $Q_\infty = C_\infty$. The proof of correctness implies that the structure always exists.

Theorem 3 *Given a cover tree on S with root C_∞,* **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 has scale ∞, there is some minimal scale i between ∞ and the scale where line 2 first holds such that $d(p,Q_i) \leq 2^i$ and so 3b 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 $d(p,q) \geq d(p,q') - \sum_{j=i'-1}^{i} 2^j = d(p,q') - (2^{i'} - 2^i) = 2^{i'} - (2^{i'} - 2^i) = 2^i$, which proves the desired separation $d(p,C_{i-1}) > 2^{i-1}$. Separation at levels below is proved similarly. ∎

3.4. Single Point Removal

The removal (Algorithm 3) is similar to insertion, with some extra complexity due to coping with children of removed nodes.

Algorithm 2 <u>Insert</u>(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 $Q_{i-1} = \{q \in Q : d(p, q) \le 2^i\}$.

        (b) if **Insert**$(p, Q_{i-1}, i-1) =$ "no parent found" and $d(p, Q_i) \le 2^i$

                pick $q \in Q_i$ satisfying $d(p, q) \le 2^i$
                insert p into Children(q)
                return "parent found"

        (c) else return "no parent found"

Algorithm 3 <u>Remove</u>(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 $Q_{i-1} = \{q \in Q : d(p, q) \le 2^i\}$

3. **Remove**$(p, \{Q_{i-1}, Q_i, ..., Q_\infty\}, i-1)$

4. if $d(p, Q) = 0$ then

    (a) remove p from C_{i-1} and from Children(Parent(p))

    (b) for every $q \in$ Children(p)
           set $i' = i - 1$
           while $d(q, Q_{i'}) > 2^{i'}$
               insert q into $C_{i'}$ (and $Q_{i'}$) and increment i'.
           choose $q' \in Q_{i'}$ satisfying $d(q, q') \le 2^{i'}$ and make q' point to q

---

Theorem 4 *Given a cover tree on S, **Remove**$(p, \{C_\infty\}, \infty)$ returns a cover tree on $S - \{p\}$.*

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 $d(q, q') \le 2^i$ 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. ∎

4. The Runtime Analysis

In this section, the distinction between implicit and explicit representation (see Section 2) is important. We start with three lemmas about some structural properties of the cover tree.

Lemma 4.1 (*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 $d(p, q) \le 2^i$, we have $B(p, 2^{i+1}) \subset B(q, 2^{i+2})$ implying $|B(p, 2^{i+1})| \le |B(q, 2^{i+2})| \le c^4 |B(q, 2^{i-2})|$. 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 $d(p, q) \le 2^i$. Then the number of disjoint balls around the children that can be packed into $B(p, 2^{i+1})$ is bounded by

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

which bounds 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 4.2 (*Growth Bound*) *For all points $p \in S$ and $r > 0$, if there exists a point $q \in S$ such that $2r < d(p, q) \le 3r$, then*

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

Proof: Since $B(p, r) \subset B(q, 3r + r)$, we have $|B(p, r)| \le |B(q, 4r)| \le c^2 |B(q, r)|$. And since $B(p, r)$ 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 4.3 (*Depth Bound*) *The maximum depth of any point p 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 5 (*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 4.3 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 $O(k \max_i |Q_i|^2)$.

Also note that in Step 2(a), the number of children encountered is at most $kc^4 \max_i |Q_i|$ using Lemma 4.1. 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

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

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

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

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 4.1, we 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.

Finally we analyze dynamic operations.

Theorem 6 *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 4.3. 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 be implicit for many 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 $|Q_i| \le c^4$ from Lemma 4.1. Multiplying the bounds gives 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. Other steps require work no greater than for insertion. ∎
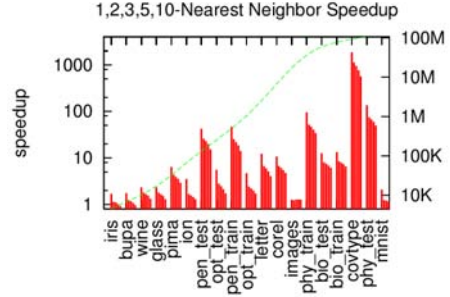

Figure 1. Speedups over the brute force search (logscale) when querying for the nearest $\{1, 2, 3, 5, 10\}$-neighbors of every point in the dataset; datasets are sorted by their byte size in ascending order (shown with a dashed line).

5. Experimental Results

We tested the algorithm on several datasets drawn from the following sources:

– KDD and UCI machine learning archieves:
  `http://kdd.ics.uci.edu/`
  `http://www.ics.uci.edu/~mlearn/`
– KDD 2004 championship dataset:
  `http://kodiak.cs.cornell.edu/kddcup`
– Mnist handwritten digit recognition dataset:
  `http://yann.lecun.com/exdb/mnist/`
– Isomap "Images" dataset:
  `http://isomap.stanford.edu/datasets.html`

For each dataset, we queried for the nearest $\{1, 2, 3, 5, 10\}$-neighbors of each point using the Euclidean metric. The results compared to an optimized brute force algorithm, are summarized in Figure 1. Results for the l_1 metric are similar.

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 2(b) shows two 5000-point datasets with the same worst-case expansion constant but different distributions of expansion across points, and not surprisingly, very different speedups. Figure 2(c) suggests that, for example, the 80th percentile (over datapoints) expansion constant seems to be a better predictor of performance.

Finally, we did experiments comparing cover trees to Clarkson's $sb(S)$ data structure (Clarkson, 2002) developed for the same setting as ours (see also (Clarkson, 1999)). For each dataset, we did exact nearest neighbor queries of every point using the "d" method
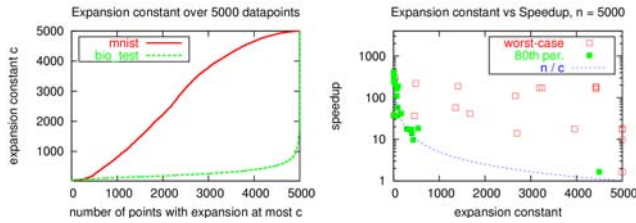
Figure 2.

(b) 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. (c) Speedups versus the worst case and the 80th percentile expansion constants of various 5000-point datasets obtained as prefixes of datasets from Figure 1.
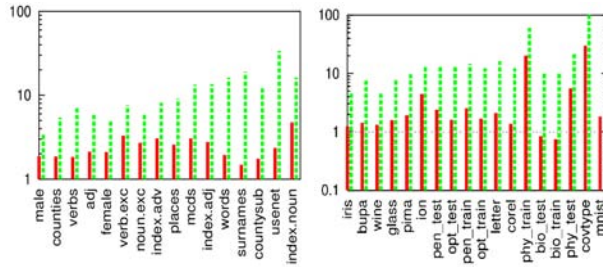

Figure 3. The speedup (logscale) over $sb(S)$ (Clarkson, 2002): (a) NNS of every point in the dataset; points are strings under the edit distance. Dashed spikes show the corresponding speedups in the construction times. (b) (1,2)-NNS (solid and dashed lines respectively). (Mnist is missing due to parsing issues with $sb(S)$.)

in (Clarkson, 2002) that was reported to be uniformly superior to all other methods 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 are shown in Figure 3(b). 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 3(a) shows the speedup of the cover tree over $sb(S)$ for strings under the edit distance.

References

Beygelzimer, A., Kakade, S., & Langford, J. (2005). Cover trees for nearest neighbor. Available at `http://hunch.net/~jl/projects/cover_tree`.

Clarkson, K. (1999). Nearest neighbor queries in metric spaces. *Discrete and Computational Geometry*, *22*, 63–93.

Clarkson, K. (2002). Nearest neighbor searching in metric spaces: Experimental results for sb(s). `http://cm.bell-labs.com/who/clarkson/Msb/readme.html`.

Friedman, J., Bentley, J., & Finkel, R. (1977). An algorithm for finding best matches in logarithmic expected time. *ACM Transactions on Mathematical Software*, *3*, 209–226.

Gray, A., & Moore, A. (2000). N-body problems in statistical learning. *Advances in Neural Information Processing Systems*, *13*, 521–527.

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

Har-Peled, S., & Mendel, M. (2006). Fast constructions of nets in low dimensional metrics and their applications. *SIAM Journal on Computing*, *35*, 1148–1184.

Karger, D., & Ruhl, M. (2002). Finding nearest neighbors in growth restricted metrics. *Proceedings of the 34th Annual ACM Symposium on Theory of Computing* (pp. 741–750).

Krauthgamer, R., & Lee, J. (2004a). The black-box complexity of nearest neighbor search. *Proceedings of the 31st International Colloquium on Automata, Languages and Programming* (pp. 858–869).

Krauthgamer, R., & Lee, J. (2004b). Navigating nets: Simple algorithms for proximity search. *Proceedings of the 15th Annual Symposium on Discrete Algorithms* (pp. 791–801).

Laviolette, F., Marchand, M., & Shah, M. (2005). A PAC-bayes approach to the set covering machine. *Advances in Neural Information Processing Systems*, *18*.

Omohundro, S. (1987). Efficient algorithms with neural network behavior. *Journal of Complex Systems*, *1*, 273–347.

Uhlmann, J. (1991). Satisfying general proximity/similarity queries with metric trees. *Information Processing Letters*, *40*, 175–179.