FAST NEAREST NEIGHBORS

THOMAS KOLLAR

Abstract. We present a review of the literature on fast nearest neighbors using the basic approach from Karger and Ruhl [4] and a recent technique called cover trees. A small error in **Insert** procedure from the original paper on cover trees is corrected and an examination of how query time actually varies with the size of the problem is shown using a *Python* implementation of the basic cover tree algorithms.

1. Introduction

Nearest neighbor queries are essential for many applications: from machine learning to computer vision to networking. Most fast nearest neighbor algorithms are specific to Euclidean spaces. For general metric spaces, progress has been slow.

More recently, fast algorithms have been developed for use on metric spaces that satisfy extra properties on the *intrinsic* dimensionality of the dataset. This additional assumption that there exists a relatively small quantity called an expansion constant c on the dataset.

Given that a dataset satisfies this expansion constant property for some c, $O(log(n))$ query times can be derived. While there have been a number of solutions that perform nearest neighbor queries in $O(log(n))$ time [4, 5], we focus on the most recent work on a datastructure called a *cover tree*.

The improvements of using such a datastructure are threefold. First, the space used for cover trees is now $O(n)$, regardless of the dimensionality of the dataset. The second is that the reliance of the algorithm on the expansion constant is explicitly stated. Finally, cover trees provides fewer parameters to set at runtime. Cover trees have also been shown to have better query time when compared with other fast-in-practice implementations like $sb(S)$.

2. Background

In this section we will define a *metric space,* a *closed ball* and develop an idea of the kinds of functionality that are required in a nearest neighbor datastructure. Thus, we define a metric space as follows:

Definition 2.1. A *Metric Space* is a 2-tuple (M, d) where M is a nonempty set of points and d is a function from $M \times M$ to $\mathbf{R}$ satisfying the following four properties for all points $x, y, z \in M$:

(1) $d(x, x) = 0$
(2) $d(x, y) > 0$ if $x \neq y$
(3) $d(x, y) = d(y, x)$
(4) $d(x, y) \leq d(x, z) + d(z, y)$

Here we can see that the first is the requirement that the distance from a point to itself must be 0 (reflexivity). The second requirement says that distances are always positive (positivity). The third says that the distance cannot depend on the order of the points (symmetry). Finally the last requirement is that it must satisfy the Triangle inequality such that the direct distance between x and y is less than the distance from x to y through some other point z.

The following are examples of metrics, the proofs are left to the reader.

Example 2.2. For $M = \mathbf{R^n}$ and $d(x, y) = \|x - y\| = [\sum_i (x_i - y_i)^s]^{\frac{1}{s}}$.

For $s = 2$, this becomes the well-known the *Euclidean metric.*

Example 2.3. $M = \mathbf{R^n}$ again, and $d(x, y) = max\{|x_1 - y_1|, \ldots, |x_n - y_n|\}$.

Note that this is simply the metric from Example 2.2, for $s \to \infty$.

Example 2.4. For $d(x,y) = 0$ if $x = y$ and $d(x,y) = 1$ if $x \neq y$.

This is called the *discrete metric.*

Now let us introduce the notion of a *closed ball* for some metric space M.

Definition 2.5. Let (M,d) be a metric space M with metric d. A *closed ball* of radius r around a point $p \in M$ is defined as: $B_p^M(r) = \{x | x \in M, d(x,p) \leq r\}$.

We will drop the superscript M in this paper since from context the metric space we are working with should always be clear.

For the rest of the paper, M is a metric space with metric d. $S \subset M$ will be the dataset drawn from the metric space M. The minimum distance of a point p from any point in a dataset S is defined as follows:

$$d(p,S) = min_{q \in S} d(p,q)$$

Given that we have a metric space that satisfies the properties in 2.1, we want to perform certain types of queries. These include:

- *Range Queries:* Given an element q and a set S, retrieve all elements that are within distance r of q in S.
- *k-Nearest Neighbor Queries:* Given an element q and a set S, retrieve the closest k elements to q in S.

In the general metric space progress has been slow. The current best times for range and nearest neighbor queries when the query point is far away are $\Omega(n^{1-\delta})$ [4]. For our purposes, we will consider only *1-Nearest Neighbor* queries with the knowledge that these algorithms are extensible to the *k-NN* case.

Instead of attempting to solve nearest neighbor queries for a general metric space, we add a fifth property to the above four properties of the metric space to make the problem simpler. Karger and Ruhl call the additional property the *expansion rate* c of the dataset. The runtimes depend on this expansion constant, which roughly corresponds to the minimum speed with which datapoints will come into view as the diameter a ball around a any point in M is doubled.

The alternative to the expansion constant is a quantity called the *doubling dimension* of a metric space. This was proposed by [5] as an alternative to the *expansion rate* of Karger and Ruhl. If we have a covering of the metric space M, then the *doubling dimension* roughly corresponds to the minimum number of balls with half the diameter that are needed to cover the metric space M.

3. The expansion constant c

We note that there is a difference between the *intrinsic* and the *representational* size of the data. A set of points that lie on the plane in a 40 dimensional space with the Euclidean metric would have a representational size of 40, but an intrinsic size of 2. We look to the *expansion constant* and the *doubling constant* to help give measurements of this intrinsic dimension of a dataset, attempting to factor out notions of representational size whenever possible.

3.1. *Expansion constant* of Karger Ruhl[4].

The *expansion constant* of a set $S \subset M$ is defined on a metric space (M,d).

Definition 3.1. For a metric space (M,d) and $r > 0$, we say that $S \subset M$ has expansion (ρ, c) iff $\forall p$

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

The *expansion constant* is defined as the smallest value c such that $|B_p(2r)| \leq c|B_p(r)|$ $\forall p \in M$ and $r > 0$.

Karger and Ruhl proved some properties of the expansion constant which lead to a simple algorithm for finding the nearest neighbor. We present here a simplified version of their algorithm that does not worry about space requirements, giving a flavor of their approach. Thus, we rederive some of the properties associated with the *expansion constant.*

Lemma 3.2. *Let 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 $r \geq d(p,q)$ with $\left|B_q(\frac{r}{2})\right| \geq \rho$, then:*

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

Proof. So we want the number of k points that are within $B_q(\frac{r}{2})$, the ball of half the current distance to the query point q. We leave it as an exercise to show the *sandwich lemma:* if $d(p,q) \leq r$, then $B_q(r) \subseteq B_p(2r) \subseteq B_q(4r)$.

We want $d(p,q) \leq \frac{r}{2}$. So, from the *sandwich lemma* if we found such a point, we would have: $B_q(\frac{r}{2}) \subseteq B_p(r)$. But clearly, since we are sampling from $B_p(2r)$ and $B_q(\frac{r}{2}) \subseteq B_p(r) \subseteq B_p(2r)$, we can conclude that the k points in the ball of half the current radius around q are eligible for sampling.

Now, since under our assumption $d(p,q) \leq r$ we use the *sandwich lemma* to conclude that $|B_p(2r)| \leq |B_q(4r)|$ and expanding using the expansion constant, we have:

$$|B_q(4r)| \leq c \cdot |B_q(2r)| \leq c^2 \cdot |B_q(r)| \leq c^3 \cdot \left|B_q(\frac{r}{2})\right| = c^3 k$$

So we have k points from the ball of half the radius around q and there are at most $c^3 k$ points in the ball from which we are sampling. Thus, in a random sample, we will have the following probability that at least one sample is inside the ball of radius $\frac{r}{2}$ around q:

$$p(\text{sample is inside the ball of radius } \frac{r}{2} \text{ around } q) = \frac{k}{c^3 k} = \frac{1}{c^3}$$

Finally, we want to know the probability of drawing $3c^3$ bad samples. Note that $(1 - \frac{1}{x})^x \leq \frac{1}{e}$, so we can conclude:

$$\begin{aligned} p(\text{ drawing } 3c^3 \text{ bad samples}) &= \left(1 - \frac{1}{c^3}\right)^{3c^3} \\ &\leq (\frac{1}{e})^3 \\ &\leq 0.05 \end{aligned}$$

We thereby succeed in finding a point inside of $B_q(\frac{r}{2})$ with high probability (at least 95%). □

From this theorem, we can deduce a simple algorithm for performing nearest neighbor queries. It is shown in Algorithm 1. In particular, it keeps sampling $3c^3$ points between the current closest point p to the query point q. It then takes the closest such p and iterates. This takes logarithmic time in the ratio of the largest separation of $q, p \in S$ to the smallest.

Algorithm 1 Nearest Neighbor Search Algorithm for Karger Ruhl

```
Nearest Neighbor(query point q):
    p ←arbitrary point in S
    while p is not the nearest neighbor of q:
        X is a random sample of 3c^3 elements of B_p(2d(p,q))
        p is the element of X ∪ {p} of minimal distance to q
    return p
```

Take $\triangle = \frac{max\ d(p,q)}{min\ d(p,q)}$ for all $p, q \in M$. This is the ratio of the maximum distances in the metric space to the minimum distance in the metric space. Note now that clearly $\frac{1}{\triangle} = \frac{min\ d(p,q)}{max\ d(p,q)} < min\ d(p,q)$. We are now ready to prove that this algorithm runs in $O(log\triangle)$ time.

Theorem 3.3. *The algorithm shown completes in expected $O(log\triangle)$ time.*

Proof. Normalize the space such that the furthest distance from any point is 1. We want the number of times that it will take us to reduce the size of the ball to include the nearest neighbor. Note that at most we will have to reduce it to $min\ d(p,q)$. When this happens, only one $p \in S$ will be possible.

Call N the number of times that we have to select $3c^3$ points. Now, call T_i the number of trials to find an element in the ball of size $\frac{r}{2}$, for r the current radius. Thus, we have $N = \sum_{i=1}^{log(\triangle)} T_i$. The sampling lemma states that $p(T_i = k) > \frac{9}{10}(\frac{1}{10})^k$. We notice that this is a geometric distribution with parameter $p = \frac{9}{10}$.

Thereby, we can conclude that the expectation is $E[T_i] = \frac{10}{9}$. So,

$$E[N] = E[\sum_{i=1}^{log(\triangle)} T_i] = \sum_{i=1}^{log(\triangle)} E[T_i] = \sum_{i=1}^{log(\triangle)} \frac{10}{9} = \frac{10}{9} log(\triangle) = O(log(\triangle))$$

Note that we choose the upper bound on the summation to be $log(\triangle)$ since after $log(\triangle)$ successful trials we will be within $\frac{1}{2^{log(\triangle)}} = \frac{1}{\triangle} < min\ d(p,q)$. □

In Algorithm 1, we notice that there is a line that requires $3c^3$ samples drawn from a ball of radius $B_p(2d(p,q))$. Karger and Ruhl decide to sample elements from circles with radii of power 2^i in advance for each of the n elements in the dataset S. Thus, given a query point, select the ball of radius 2^i that has radius just a bit more than $2d(p,q)$ and use these samples. Continue until the nearest neighbor has been computed. The modified algorithm can be seen in Algorithm 2.

Algorithm 2 Nearest Neighbor Search Algorithm for Karger Ruhl

```
Nearest Neighbor(query point q):
    p ←arbitrary point in S
    while p is not the nearest neighbor of q:
        X is a random sample of 3c^3 elements precomputed for a
            circle of radius 2^i just bigger than 2d(p,q)
        p is the element of X ∪ {p} of minimal distance to q
    return p
```

At this point, we leave the discussion of the nearest neighbor approaches by Karger and Ruhl. Let it be noted that they implement this with $O(nlogn)$ space and $O(log(n))$ nearest neighbor query time using what they call a *metric skip list.*

3.2. ***Doubling constant* of [5].** We now move on to the *doubling constant* of Krauthgamer and Lee. It can be shown the *doubling constant* has fewer problems than the expansion constant. In particular, the doubling constant is robust to small changes in the dataset.

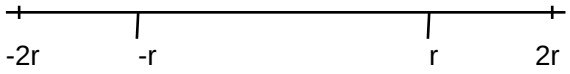
Definition 3.4. The *doubling dimension* of a metric space (M,d) is the minimum value of c_{KL} such that every set in M can covered by $2^{c_{KL}}$ sets of half the diameter.

Lemma 3.5. *Krauthgamer and Lee show that, for a given space X, $c_{KL} \leq 4c_{KR}$.*

However, there is no upper bound on c_{KR} using c_{KL}. Thus, given a c_{KR} for a dataset, one can directly come up with a c_{KL} for that dataset. However, since the converse does not hold c_{KL} would seem to be more general. The Karger-Ruhl dimension also has some strange properties. In particular, for some one dimensional subsets of the real line, the KR dimension is unbounded where the doubling dimension would be finite.

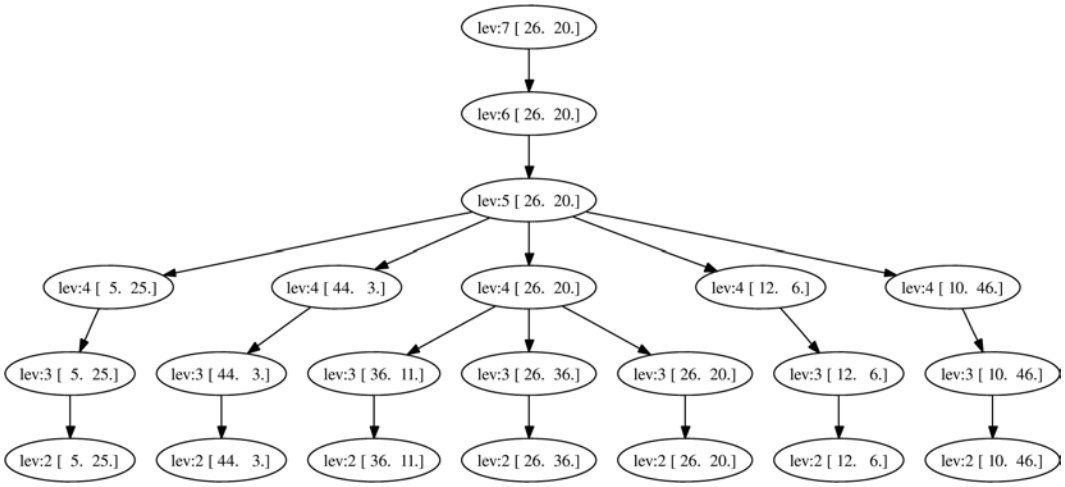
Example 3.6. The addition of a single point can cause the *expansion constant* of a set to grow arbitrarily. Take a discrete annulus $S = \{x \in Z : 2r > |x| > r\}$. We want to show that th addition of the element $\{0\}$ to this annulus will cause the c_{KR} to be approximately r while c_{KR} for the annulus alone is approximately 2. However, it should be directly clear that c_{KL} will increase by at most 1 if we put a ball over 0.

FIGURE 3.1. Annulus Example



(1) S **is the discrete annulus only:** For this case we need only worry points p inside one side of the discrete annulus interacting with elements in the same side or in elements of the other side. If $p < 0$ then the other side is made up of q s.t. $q > 0$ (and vice versa). We will conclude that $c_{KR} = 2$ for this case.

FIGURE 4.1. An example cover tree with seven elements



(a) *Same side interactions:* Clearly if elements are on the same side then $B_p(r')$ has less than or equal to r' points in it: $\left|B_p(r')\right| \approx r'$. So, we conclude that $\left|B_p(2r')\right| \approx 2r' = 2\left|B_p(r')\right|$. Thus, for the one side a constant of 2 works. Now we need to understand how a point on one side of 0 interacts with points on the other.

(b) *Opposite side interactions:* The worst interaction will happen with the element closest to 0 on either size. We take this element and note that its radius is at least $2r$ (for r the radius of the annulus) by the time it reaches any element in the other side. So, this element will already include all of its side $|B_p(2r)| \approx r$. But in the whole annulus there are at most $2r$ points, so we conclude that $|B_p(4r)| \approx 2r = 2\,|B_p(2r)|$.

(2) S **includes the point** $\{0\}$**.** Assume S includes 0 and that $p = 0$. Now, presume also that we have at least one element in $B_p(r')$ (this of course means that we have at 2 points since the annulus is symmetric around 0). So $\left|B_p(r')\right| = 2$. However, $r' \geq r$, so $2r' \geq 2r$. Thus, $B_p(2r')$ includes all the elements of the annulus. Clearly this means that $\left|B_p(2r')\right| = 2r \leq r\left|B_p(r')\right|$. Thus, we conclude that the minimum c_{KR} must be at least r.

4. Cover Trees

Cover trees are a relatively new data structure. Independent of the doubling dimension, the space used is $O(n)$. Further, nearest neighbor queries can be done in $O(c^{12}ln(n))$ time with insertion and removal taking $O(c^6 ln(n))$. An example of a cover tree can be seen in Figure 4.1. Note that each of the points is compared using the Euclidean metric. At first glance, it might appear that once a node appears it is in the tree forever, that at higer levels nodes seem to be relatively far apart and that children are somewhat close to the parents. These are in fact the three properties of a cover tree, as we now state formally.

Definition 4.1. A *cover tree* T on a dataset S has the following three properties:

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

Let us take a closer look at these three properties. In particular, the nesting property states if a point p is in the tree at the i^{th} stage, then it is also in the tree at the $i-1^{st}$ stage. If we view this on a scale from ∞ to $-\infty$, then intuitively there should be only one point at ∞ and the dataset S at $-\infty$. See Figure 4.2.

The covering tree property says that every node at the $i-1^{st}$ stage has exactly one parent that is within distance 2^i of it. In other words, there exists only one path to each node in the tree, with the parents being close to the children (close with respect to the metric d). See Figure 4.3.

FIGURE 4.2. Graphical representation of property 1. Note that each circle is a point.

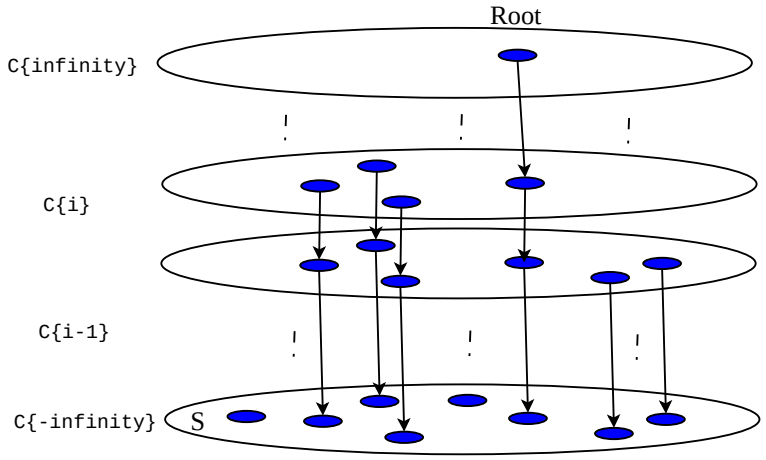


FIGURE 4.3. Graphical representation of property 2.

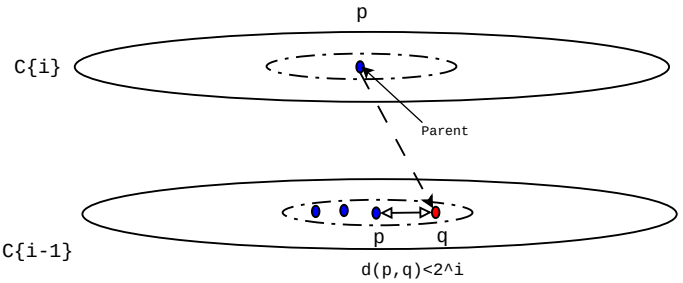
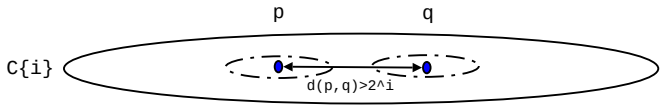


FIGURE 4.4. Graphical representation of property 3.



Finally, the separation property states that every node at the i^{th} stage is separated by at least 2^i. This means that at every level in the tree, the nodes are far away from one another with respect to the metric d. See Figure 4.4.

Before moving into the algorithms we state three theorems without proving them that will be useful later.

Lemma 4.2. *(Width bound) The number of children of any node p is bounded by c^4*

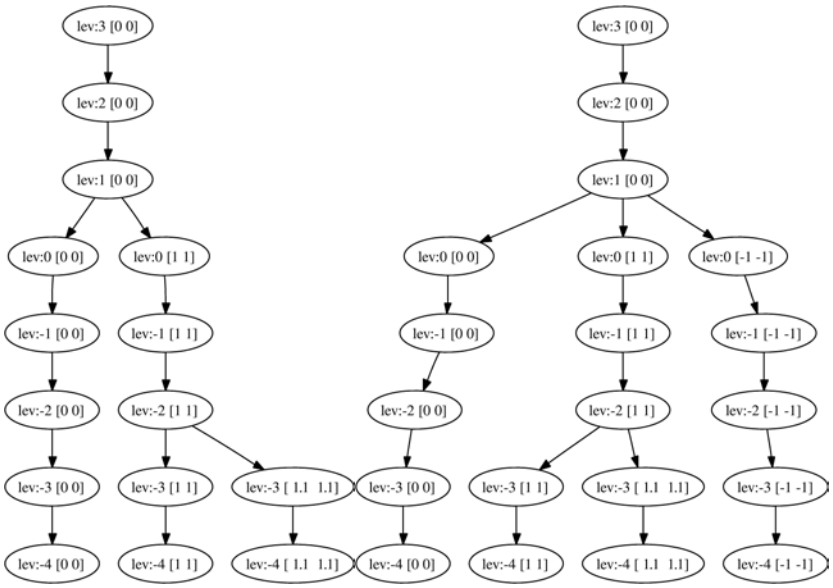
Lemma 4.3. *(Growth bound) For all points $p \in S$ and $r > 0$, if there exists a point $q \in S$ such that $2r < d(p,q) \leq 3r$, then $|B_p(4r)| \geq (1 + \frac{1}{c^2})\,|B_p(r)|$*

Lemma 4.4. *(Depth bound) The maximum depth of any point p in the explicit representation is $O(c^2 log(n))$*

4.1. **Insert.** The insert algorithm starts at the root element such that $Q_\infty = C_\infty$ and it recurses down the tree until it has found a position to put p such that the three properties of a cover tree are satisfied (nesting, covering, and separation). Note that Q_i are the eligible insertion locations (as required by the covering property) at level i and p is the point to insert. We will prove shortly insert runs in $O(log(n))$ time.

The *explicit* representation of a cover tree can be seen in Figure 4.5. The explicit representation would be the representation of the tree structure in Figure 4.5 if each node had to be stored separately. Fortunately, this is not the case and in fact we can use what is called an explicit representation of the tree that stores

FIGURE 4.5. Two stages of insert for element $[-1,-1]$. On the left is the cover tree before insertion and on the left is the cover tree after insertion. The Euclidean metric was used in this example. Note that only the rightmost branch changes.



its data and a pointer to its children at . Because of this, we get trivially that cover trees require only $O(n)$ space.

Example 4.5. In Figure 4.5 we show a cover tree on the left and a cover tree on the right after the insertion of a new element $[-1,-1]$. Note that we are using a Euclidean metric and that *lev:i* means that we are on the level that corresponds to balls of size 2^i.

Intuitively, the process of insertion happens as follows. Say the algorithm is starting at the root level. Examine the children of the root level and if the new point is within 2^i of the root as well as being more than 2^i away from all of the children. If it is, then insert it under the root. Otherwise, recurse on all the children that are within 2^i of the root.

For the particular example of $[-1,-1]$, we start at level 3 and we look at the child of $[0,0]$. Clearly $[-1,-1]$ is within $2^2 = 4$ of the only child of level 3: $[0,0]$. Thus, we violate the *separation* property and have to recurse. The same thing happens at level 2 and we recurse again.

Once the algorithm reaches level 1, it finds that it is more than $2^0 = 1$ away from the points $[1,1]$ and $[0,0]$. Clearly it is within 2^1 of $[0,0]$ and thus it satisfies the covering property. Since each element repeats itself at below where it was inserted, we can be assured that the nesting property holds as well. Thus, we can see that the insertion of $[-1,-1]$ respects all of the properties of the cover tree.

We now want to prove that **Insert** creates a cover tree after p is inserted. Note that the original paper [2] has a slightly broken version of **Insert**. The procedure shown in Algorithm 3 has been modified slightly and fixes small bugs from the original paper.

Remark 4.6. The original description of the insert algorithm in [2] has minor bugs (it is reproduced in Algorithm 4) . If we interpret "no parent found" to be *false* and "parent found" to be *true*, then we can see that this algorithm will fail to maintain the cover tree properties.

In particular, it is easy to see that the covering property will fail to hold. Say that the recursion bottoms out at the first *if* statement and that the recursion depth is n. Then, we recurse up to depth $n-1$ and run the *else* statement, which will go into the second *if* statement. Now, this will return *true* and add a parent of p. Popping up another level of recursion to $n-2$, the last *else* statement is entered and $false$ is returned. One more recursion to $n-3$ causes the algorithm to perhaps enter the second *if* statement again. Thus, we have added two elements to be parents of the point p, which violates the *covering* property of cover trees.

Now we will prove that the modified procedure in Algorithm 3 maintains the cover tree properties.

Algorithm 3 Insert procedure for Cover Trees

```
Insert(point p, cover set Q_i, level i):
    Q = {Children(q) : q ∈ Q_i}
    if d(p, Q) > 2^i:
        return "parent found" - True
    else:
        Q_{i-1} = {q ∈ Q : d(p, q) ≤ 2^i}
        found = Insert(p, Q_{i-1}, i − 1)
        if found and d(p, Q_i) ≤ 2^i
            pick a single q ∈ Q_i such that d(p, q) ≤ 2^i
            insert p into Children(q)
            return "finished" -False
        else:
            return found
```

Algorithm 4 Original faulty **Insert** procedure for Cover Trees

```
Insert(point p, cover set Q_i, level i):
    Q = {Children(q) : q ∈ Q_i}
    if d(p, Q) > 2^i:
        return "no parent found"
    else:
        Q_{i-1} = {q ∈ Q : d(p, q) ≤ 2^i}
        if Insert(p, Q_{i-1}, i − 1)=="parent not found" and d(p, Q_i) ≤ 2^i
            pick a single q ∈ Q_i such that d(p, q) ≤ 2^i
            insert p into Children(q)
            return "parent found"
        else:
            return "no parent found"
```

Theorem 4.7. *If T is a cover tree over a set S, then running **Insert**(p, Q_i, i) from 3 for a new point p preserves the cover tree properties.*

Proof. First, we need to show that this algorithm completes. To do this, all that we need to show is that at some point we enter the first *if* statement. Clearly this will be entered since at each stage we are decreasing i so that the cover size of each point is 2^i at . For some i, clearly p will fall outside the cover for all the points currently in S as long as S is discrete. Thus, the first *if* statement will be invoked. Since the *if* statement holds, there will be some *minimal* level where the point will be inserted.

We now have three things to show given that the algorithm completes: *nesting*, *covering*, and *separation*. Presume that p is inserted at level $i-1$.

1) **Covering:** We know that $d(p, Q_i) \leq 2^i$ by the second *if* statement, so there exists a parent for p and we pick exactly one.

2) **Nesting:** Since when we insert p, we implicitly insert p into all levels below p, then we can clearly see that $C_i \subset C_{i-1}$

3) **Separation:** The first *if* statement ensures that we have added only elements who are also separated.

Thus, we are done. □

Theorem 4.8. ***Insertion*** *takes time at most $O(c^6 log(n))$.*

4.2. **Nearest Neighbor.** The nearest neighbor algorithm starts at the root of the tree and iteratively considers the viable children of each level C_i. When it has reached a level i such that 2^i is less than the minimum distance in the dataset: $min_{p,q \in S}\ d(p, q)$, we conclude that we have found the nearest neighbor. The procedure is shown in Algorithm 5.

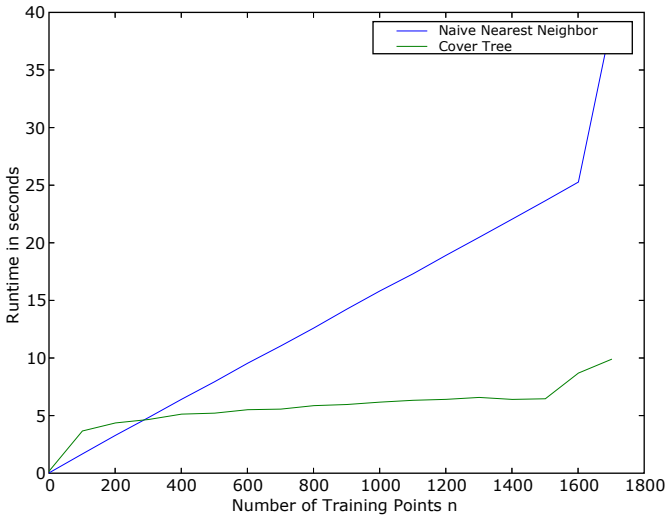
Theorem 4.9. ***NearestNeighbor***(*root, p*) *returns the nearest neighbor of p in S.*

Proof. We start at the root node and we keep getting the children of eligible nodes. A node is eligible if it satisfies $d(p, q) \leq d(p, Q) + 2^i$. Note that these points are the only ones that could possibly do better than

Algorithm 5 Find the nearest neighbor

```
NearestNeighbor(cover tree T, query point p):
    Q_∞ = C_∞
    for i from ∞ down to −∞
        Q = {Children(q) : q ∈ Q_i}
        Q_{i−1} = {q ∈ Q : d(p, q) ≤ d(p, Q) + 2^i}
    return argmin_{q∈Q_{−∞}} d(p, q)
```

FIGURE 5.1. Runtime comparison for naive and cover tree over 1000 queries



q^*, where q^* is the best on the current round. In other words, move out a distance $d(p, Q)$ from the query point. Anywhere on the circle around point p at this distance with radius 2^i could provide a better solution than q^*. Further, these points are the only ones that could do so.

The algorithm ends when the distance between p and q is less than the minimum distance in the dataset. In this case, there is a single point that is the nearest neighbor to p. □

Theorem 4.10. *If the dataset $S \cup \{p\}$ has an expansion constant of c, then the runtime of* ***NearestNeighbor*** *is $O(c^{12} log(n))$.*

4.3. **Extensions.** The approaches for cover trees shown here have been generalized to ϵ-approximate queries, batch queries and batch construction of the data structure. Deletion is similar to insertion. The runtimes of these are shown in Table 1.

5. Discussion

To ensure ourselves that the query time is in fact logarithmic on a practical scale, we performed a simple experiment to compare cover trees to the naive version of nearest neighbors. For this project, the cover tree algorithm was implemented in the *Python* programming language.

The strawman here was the natural linear algorithm that scans through all of the elements and picks the minimum distance element. This takes $O(n)$ time to do. In Figure 5.1 x-axis shows the size of the dataset and the y-axis shows the runtime in seconds. The data is uniformly distributed in the set of $[0, 5000] \times [0, 5000]$ where the metric is the standard Euclidean metric: the L_2 norm (see Example 2.2 with $s = 2$). A thousand queries were performed for each dataset size.

As expected, for a fixed dataset size, the naive algorithm scales linearly and the Cover Tree scales logarithmically. It appears that cover trees become more effective than the naive version of nearest neighbor with as few as 300 training points. Reproducing the table from [2], we can see how the runtimes of the algorithms

TABLE 1. Runtime Comparison

	Cover Tree	Navigation Net	Karger and Ruhl	Naive Implementation
Construction Space	$O(n)$	$O(n)$	$O(nln(n))$	$O(n)$
Construction Time	$O(c^6 nln(n))$	$O(nln(n))$	$O(nln(n))$	$O(1)$
Insertion/Removal	$O(c^6 ln(n))$	$O(ln(n))$	$O(ln(n))$	$O(1)/O(n)$
Query	$O(c^{12} ln(n))$	$O(ln(n))$	$O(ln(n))$	$O(n)$
Batch Query	$O(c^{16} n)$	$O(nln(n))$	$O(nln(n))$	$O(n)$

compare. The advantage of the cover tree, aside from its explicit dependence on c is that its structure is simple and intuitive. Further, it is likely easier to implement than many of the other techniques.

Finally, the work of Karger and Ruhl is not entirely dissimilar from cover trees. The methods of Karger and Ruhl simply have a different way of reducing the distance from the query point q in half. For Karger and Ruhl, they would sample the space of points, for cover trees they would recurse on possible covers. In contrast to KR, cover trees represent the relationship between close and distant points explicitly. While Karger and Ruhl's approach finds an efficient method to perform the needed sampling and to store the samples, in contrast cover trees need no sampling at all and require only a tree that conforms to the cover tree properties.

6. Conclusions

Cover trees have a simple representation and fast query times. The constants at runtime are low enough to make their implementation practical. In [2], nearest neighbor speedups for cover trees were shown for many standard datasets including the NIST handwritten digits dataset and many biological datasets. Here, we have attempted to give a clear, concise and basic understanding of both Karger and Ruhl's work on fast nearest neighbor and on Beygelizimer, Kadade and Langford's work on cover trees. We have shown for a fixed query size and increasing dataset size, that our *Python* implementation of cover trees scales favorably.

References

[1] T. Apostol, **Mathematical Analysis: 2nd Ed.**: Addison-Wesley Publishing Company, 1974.

[2] A. Beygelzimer, S. Kakade, and J. Langford. *Cover Trees for Nearest Neighbor,* 2005.

[3] E. Chavez, G. Navarro, R. Baeza-Yates, and J.L. Marroquin. *Searching in Metric Spaces,* Vol. 33, No. 3, September 2001, p.273-321.

[4] D. Karger and M. Ruhl. Finding nearest Neighbors in Growth Restricted Metrics, *Proceedings STOC,* 2002.

[5] 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.

[6] R. Parwani. **Complexity: A course.** http://staff.science.nus.edu.sg/~parwani/c1/node17.html

[7] J. Munkres. **Topology: 2nd Edition**. Prentice Hall, 2000.