Abstract

The cover tree data structure speeds up exact nearest neighbor queries over arbitrary metric spaces (Beygelzimer et al., 2006). This paper makes cover trees even faster. In particular, we provide

1. A simpler definition of the cover tree that reduces the number of nodes from $O(n)$ to exactly $n$.
2. An additional invariant that makes queries faster in practice,
3. Algorithms for constructing and querying the tree in parallel on multiprocessor systems, and
4. A more cache efficient memory layout.

On standard benchmark datasets, we reduce the number of distance computations by 10–50%. On a large-scale bioinformatics dataset, we reduce the number of distance computations by 71%. On a large-scale image dataset, our parallel algorithm with 16 cores reduces tree construction time from 3.5 hours to 12 minutes.

1. Introduction

Data structures for fast nearest neighbor queries are most often used to speed up the $k$-nearest neighbor classification algorithm. But many other learning tasks also require neighbor searches, and cover trees can speed up these tasks as well: localized support vector machines (Segata & Blanzieri, 2010), dimensionality reduction (Lisitsyn et al., 2013), and reinforcement learning (Tziortziotis et al., 2014). Making cover trees faster—the main contribution of this paper—also makes these other tasks faster.

Given a space of points $\mathcal{X}$, a dataset $X \subseteq \mathcal{X}$, a data point $p \in \mathcal{X}$, and a distance function $d : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$, the nearest neighbor of $p$ in $X$ is defined as

$$p_{nn} = \arg \min_{q \in X \setminus \{p\}} d(p, q)$$

The naive method for computing $p_{nn}$ involves a linear scan of all the data points and takes time $\Theta(n)$, but many data structures have been created to speed up this process. The $kd$-tree (Friedman et al., 1977) is probably the most famous. It is simple and effective in practice, but it can only be used on Euclidean spaces. We must turn to other data structures when given an arbitrary metric space. The simplest and oldest of these structures is the ball tree (Omo-hundro, 1989). Although attractive for its simplicity, it provides only the trivial runtime guarantee that queries will take time $O(n)$. Subsequent research focused on providing stronger guarantees, producing more complicated data structures like the metric skip list (Karger & Ruhl, 2002) and the navigating net (Krauthgamer & Lee, 2004). Because these data structures were complex and had large constant factors, they were mostly of theoretical interest. The cover tree (Beygelzimer et al., 2006) simplified navigating nets while maintaining good run time guarantees. Further research has strengthened the theoretical runtime bounds provided by the cover tree (Ram et al., 2010). Our contributions make the cover tree faster in practice.

The cover tree as originally introduced is simpler than related data structures, but it is not simple. The original explanation required an implicit tree with an infinite number of nodes; but a smaller, explicit tree with $O(n)$ nodes actually gets implemented. We refer to this presentation as the original cover tree. In the remainder of this paper we introduce the simplified cover tree and the nearest ancestor cover tree; parallel construction and querying algorithms; and a cache-efficient layout suitable for all three cover trees. We conclude with experiments showing: on standard benchmark datasets we outperform both the original implementation (Beygelzimer et al., 2006) and MLPack’s implementation (Curtin et al., 2013a); on a large bioinformatics dataset, our nearest ancestor tree uses 71% fewer distance comparisons than the original cover tree; and on a large image data set, our parallelization algorithm reduces tree construction time from 3.5 hours to 12 minutes.
2. Simplified cover trees

Definition 1. Our simplified cover tree is any tree where: (a) each node p in the tree contains a single data point (also denoted by p); and (b) the following three invariants are maintained.

1. The leveling invariant. Every node p has an associated integer level(p). For each child q of p
   
   \[ \text{level}(q) = \text{level}(p) - 1. \]

2. The covering invariant. For every node p, define the function covdist(p) = \(2^{\text{level}(p)}\). For each child q of p
   
   \[ d(p, q) \leq \text{covdist}(p). \]

3. The separating invariant. For every node p, define the function sepdist(p) = \(2^{\text{level}(p) - 1}\). For all distinct children \(q_1\) and \(q_2\) of p
   
   \[ d(q_1, q_2) > \text{sepdist}(p). \]

Throughout this paper, we will use the functions children(p) and descendants(p) to refer to the set of nodes that are children or descendants of p respectively. We also define the function maxdist as

\[ \text{maxdist}(p) = \arg \max_{q \in \text{descendants}(p)} d(p, q) \]

In words, this is the greatest distance from p to any of its descendants. This value is upper bounded by \(2^{\text{level}(p) + 1}\), and its exact value can be cached within the data structure.\(^1\)

In order to query a cover tree (any variant), we use the generic “space tree” framework developed by Curtin et al. (2013). This framework provides fast algorithms for finding the k-nearest neighbors, points within a specified distance, kernel density estimates, and minimum spanning trees. Each of these algorithms has two variants: a single tree algorithm for querying one point at a time, and a dual tree algorithm for querying many points at once. Our faster cover tree algorithms speed up all of these queries; but for simplicity, in this paper we focus only on the single tree nearest neighbor query. The pseudocode is shown in Algorithm 1.

Analysis of the cover tree’s runtime properties is done using the data-dependent doubling constant \(c\): the minimum value \(c\) such that every ball in the dataset can be covered by \(c\) balls of half the radius. We state without proof two facts:

\(1\)As in the original cover tree, practical performance is improved on most datasets by redefining \(\text{covdist}(p) = 1.3^{\text{level}(p)}\) and \(\text{sepdist}(p) = 1.3^{\text{level}(p) - 1}\). All of our experiments use this modified definition.

(a) any node in the cover tree can have at most \(O(c^4)\) children; (b) the depth of any node in the cover tree is at most \(O(c^2 \log n)\). These were proven for the original cover tree (Beygelzimer et al., 2006) and the proofs for our simplified cover tree are essentially the same. We can use these two facts to show that the runtime of Algorithm 1 is \(O(c^3 \log n)\) for both the original and simplified cover tree.

Algorithm 2 shows how to insert into the simplified cover tree. It is divided into two cases. In the first case, we cannot insert our data point \(x\) into the tree without violating the covering invariant. So we raise the level of the tree \(p\) by taking any leaf node and using that as the new root. Because \(\text{maxdist}(p) \leq 2\text{covdist}(p)\), we are guaranteed that \(d(p, x) \leq \text{covdist}(x)\), and so we do not violate the covering constraint. In the second case, the insert function recursively descends the tree. On each function call, we search through children(p) to find a node we can insert into without violating the covering invariant. If we find such a node, we recurse; otherwise, we know we can add \(x\) to children(p) without violating the separating invariant. In all cases, exactly one node is added per data point, so the resulting tree will have exactly \(n\) nodes. Since every

---

Algorithm 1 Find nearest neighbor

```
function findNearestNeighbor(cover tree p, query point x, nearest neighbor so far y)
1: if \(d(p, x) < d(y, x)\) then
2: \(y \leftarrow p\)
3: for each child q of p sorted by distance to x do
4: if \(d(y, q) > d(y, q) - \text{maxdist}(q)\) then
5: \(y \leftarrow \text{findNearestNeighbor}(q, x, y)\)
6: return y
```

Algorithm 2 Simplified cover tree insertion

```
function insert(cover tree p, data point x) requires: \(d(p, x) \leq \text{covdist}(p)\)
1: for \(q \in \text{children}(p)\) do
2: if \(d(q, x) \leq \text{covdist}(q)\) then
3: \(q' \leftarrow \text{insert}(q, x)\)
4: \(p' \leftarrow p\) with child q replaced with \(q'\)
5: return \(p'\)
6: return \(p\) with \(x\) added as a child
```
node can have at most \( O(c^4) \) children and the depth of the tree is bounded by \( O(c^2 \log n) \), the runtime is \( O(c^8 \log n) \).

The simplified cover tree has the same runtime bounds as the original cover tree, but it has an improved constant factor, as it needs only \( n \) nodes. Fewer nodes reduces both the overhead from traversing the data structure and the number of required distance comparisons. The original cover tree’s nesting invariant dictated these extra nodes. In our definitions, the nesting invariant states that for every node \( p \), if \( p \) has any children, then \( p \) also has itself as a child (this child need not satisfy the leveling invariant). The nesting invariant comes from the presentation of the original cover tree as an infinite data structure, but it does not play a key role in the cover tree’s runtime analysis. Therefore, we can discard it and maintain the runtime guarantees.

Figure 1 shows the reduction in nodes by using the simplified cover tree on benchmark datasets taken from the ML- Pack test suite (Curtin et al., 2013a). Section 6 contains more details on these datasets, and Figure 3 in the same section shows how this reduced node count translates into improved query performance.

3. Nearest ancestor cover trees

In this section, we exploit a similarity between simplified cover trees and binary search trees (BSTs). Insertion into both trees follows the same procedure: Perform a depth first search to find the right location to insert the point. In particular, there is no rebalancing after the insertion. Many alternatives to plain BSTs produce better query times by introducing new invariants. These invariants force the insertion algorithm to rebalance the tree during the insertion step. Our definition of the simplified cover tree makes adding similar invariants easy. We now introduce one possible invariant.

**Definition 2.** A nearest ancestor cover tree is a simplified cover tree where every point \( p \) has the nearest ancestor invariant: If \( q_1 \) is an ancestor of \( p \) and \( q_2 \) is a sibling of \( q_1 \), then

\[
d(p, q_1) \leq d(p, q_2)
\]

In other words, the nearest ancestor cover tree ensures that for every data point, each of its ancestors is the “best possible” ancestor for it at that level. Figure 2 shows a motivating one dimensional example.

Algorithm 3 shows how to insert a point into a nearest ancestor cover tree. It uses the same `insert` function as 2, but the helper function `insert_` is slightly modified in two ways (shown with an underline). First, we sort `children(p)` according to their distance from the data point \( x \). This sorting ensures that our newly inserted point will satisfy the nearest ancestor invariant. But this new point \( x \) may cause other points to violate the nearest ancestor invariant. In particular, if \( x \) has a sibling \( q_1 \) and \( q_2 \) has a descendant \( r \); and \( d(r, x) < d(r, q_2) \); then \( r \) now violates the nearest ancestor invariant. Our second step is to call `rebalance`, which finds all these violating data points and moves them underneath \( x \).

Most of the work of the `rebalance` function happens in the helper `rebalance_`. `rebalance_` returns a valid nearest ancestor cover tree and a set of points that still need to be inserted; `rebalance` just inserts those extra points. `rebalance_` takes two nearest ancestor cover trees \( p \) and \( q \) (where \( p \) is an ancestor of \( q \)) and a point \( x \). Its goal is to “extract” all points from \( q \) that would violate the nearest ancestor invariant if \( x \) became a sibling of \( p \). It returns three values: a modified version of \( q \), a set of points that cannot remain in any point along the path from \( p \) to \( q \) called the `moveset`, and a set of points that need to be reinserted somewhere along the path from \( p \) to \( q \) called the `stayset`. There are two cases. In the first case, the data point at node \( q \) must move. We then filter the descendants of \( q \) into the `moveset` or `stayset` as appropriate and return `null` for our
modified \( q \). In the second case, the data point at node \( q \) must stay. We recursively apply \texttt{rebalance} to each of \( q \)'s children; we use the results to update the corresponding child, the \texttt{moveset} and the \texttt{stayset} variables. Finally, we try to reinsert any nodes in \texttt{stayset}. If \texttt{rebalance} was called with \( q \) a child of \( p \), then the return value of \texttt{stayset} will be empty; any children that could not be directly reinserted must be in the \texttt{moveset}.

The \texttt{rebalance} function loops over all \( O(c^4) \) children of a node, and the maximum depth of the recursion is \( O(c^2 \log n) \). Therefore the overall runtime is \( O(c^6 \log n) \). It may be called up to \( O(c^4) \) times within \texttt{rebalance}, so the body of the for loop on line 12 executes \( O(c^{10} \log n) \) times. Unfortunately, we have no bound on the size of \texttt{moveset}, except to note that it is usually small in practice.

On the datasets in our experiments (see Table 1), the value of \( \log(c) \) is quite small; except to note that it is usually small in practice. In the second case, the data point at node \( q \) must stay. We recursively apply \texttt{rebalance} to nearest ancestor cover trees.

4. Parallel cover tree

In this section we discuss parallelism on shared-memory, multiprocessor machines. Querying in parallel is easy. Since the results of neighbor queries for a data point do not depend on other data points, we can: divide the points among the processors; then each processor traverses the tree independently. More difficult is constructing the tree in parallel. Our strategy is to split the data, create one cover tree on each processor, then merge these trees together. This approach is suitable for any shared-memory multiprocessor machine. We give a detailed description for merging simplified cover trees and discuss at a high level how to extend this procedure to nearest ancestor cover trees.

Algorithm 4 shows the merging procedure. The \texttt{mergeHelper} function’s main purpose is to satisfy the prerequisites for \texttt{mergeHelper}, which has two phases. First, we find all the subtrees of \( q \) that can be inserted directly into \( p \) without violating any invariants, and we insert them. Second, we insert the remaining nodes from \( q \) into \( p \) directly via the \texttt{insert} function.

The \texttt{mergeHelper} function returns a partially merged tree and a set of nodes called the \texttt{leftovers} that still need to be inserted into the tree. The first phase uses the for loop starting on line 3 to categorize the children of \( q \) into three disjoint sets. The \texttt{uncovered} set contains all of \( q \)'s children that would violate the covering invariant if inserted into \( p \). The \texttt{sepcov} set contains all of \( q \)'s children that would not violate the separating or covering invariants when inserted into \( p \). Both of these sets are unused in the second phase of \texttt{mergeHelper}. Every child of \( q \) that is not inserted into
the *uncovered* or *sepcov* sets gets merged with a suitable node in *children*(*p*). This is done by recursively calling the *mergeHelper* function. Any points that could not be inserted into the results of *mergeHelper* get added to the *leftovers* set.

In the second phase of *mergeHelper*, we insert as many nodes as possible into our merged tree *p′*. First update the children with the subtrees in *sepcov*. Then insert the root of *q*. We know that *d*(*p*, *q*) ≤ *covdist*(*p*), so this insertion is guaranteed not to change the level of *p′*. Finally, we loop over the elements in *leftovers* and insert them into *p′* only if it would not change the level of *p′*. Any elements of *leftovers* that cannot be inserted into *p′* get inserted into *leftovers*′ and returned. It is important to do this insertion of *leftovers* at the lowest level possible (rather than wait until the recursion ends and have the insertion performed in *merge*) to avoid unnecessary distance computations.

The *merge* function does not maintain the nearest ancestor invariant. A modified version of *merge* that calls the *rebalance* function appropriately could. But for space reasons, we do not provide this modified algorithm. In our experiments below, we use the provided *merge* function in Algorithm 4 to parallelize both simplified and nearest ancestor tree construction. In practice, this retains the benefits of the nearest ancestor cover tree because the nearest ancestor invariant is violated in only a few places.

Providing explicit bounds on the runtime of *merge* is difficult. But in practice it is fast. When parallelizing on two processors, approximately 1% of the distance calculations occur within *merge*. So this is not our bottleneck. Instead, the main bottleneck of parallelism is cache performance. On modern desktop computers, last level cache is shared between all cores on a CPU. Cover tree construction results in many cache misses, and this effect is exaggerated when the tree is constructed in parallel.

5. Cache efficiency

One of the biggest sources of overhead in the cover tree is cache misses. Our last improvement is to make cover trees more cache efficient. A simple way to reduce cache misses for tree data structures is to use the van Emde Boas tree layout (Frigo et al., 1999). This layout arranges nodes in memory according to a depth first traversal of the tree. This arrangement creates a *cache oblivious* data structure. That is, the programmer does not need any special knowledge about the cache sizes to obtain optimal speedup—the van Emde Boas tree layout works efficiently on any cache architecture. This layout has been known for a long time in the data structures community, but it seems unused in machine learning libraries. Frigo (1999) provides a detailed tutorial.

```plaintext
Algorithm 4 Merging cover trees
function merge(cover tree p, cover tree q)
1: if level(*q*) > level(*p*) then
2: swap *p* and *q*
3: while level(*q*) < level(*p*) do
4: move a node from the leaf of *q* to the root;
5: this raises the level of *q* by 1
6: (*p*, *leftovers*) ← mergeHelper(*p*, *q*)
7: for *r* ∈ *leftovers* do
8: *p* ← insert(*p*, *r*)
9: return *p*

function mergeHelper(cover tree p, cover tree q)
prereqs: level(*p*) = level(*q*), *d*(*p*, *q*) ≤ *covdist*(*p*)
1: *children*′ ← *children*(*p*) > Phase 1
2: *uncovered*, *sepcov*, *leftovers* ← ∅
3: for *r* ∈ *children*(*q*) do
4: if *d*(*p*, *r*) < *covdist*(*p*) then
5: *foundmatch* ← false
6: for *s* ∈ *children*′ do
7: if *d*(*s*, *r*) ≤ *sepdist*(*p*) then
8: (*s′, *leftovers*) ← mergeHelper(*s*, *r*)
9: *children*′ ← *children*′ ∪ (*s′*) − {*s*}
10: *leftovers* ← *leftovers* ∪ *leftovers*
11: *foundmatch* ← true
12: break from inner loop
13: if *not* *foundmatch* then
14: *sepcov* ← *sepcov* ∪ {*r*}
15: else
16: *uncovered* ← *uncovered* ∪ {*r*}
17: *children*′ ← *children*′ ∪ *sepcov* > Phase 2
18: *p′* ← tree rooted at *p* with *children*(*p′*)= *children*′
19: *p′* ← insert(*p′*, *q*)
20: *leftovers*′ ← ∅
21: for *r* ∈ *leftovers* do
22: if *d*(*r*, *p′*) ≤ *covdist*(*p′*) then
23: *p′* ← insert(*p′*, *r*)
24: else
25: *leftovers*′ ← *leftovers*′ ∪ {*r*}
26: return (*p′*, *leftovers*′ ∪ *uncovered*)
```

Our implementation of the cache oblivious cover tree is static. That is, we first construct the cover tree, then we call a function *pack* that rearranges the tree in memory. This means we do not get the reduced cache misses while constructing the tree, but only while querying the tree. The *pack* function is essentially free to run because it requires only a single traversal through the dataset. Figure 4 shows that the van Emde Boas tree layout reduces cache misses by 5 to 20 percent. This results in a reduction of stalled CPU cycles by 2 to 15 percent.
**Faster Cover Trees**

![Figure 3](image1.png)

**Figure 3.** (a) Constructing a nearest ancestor query tree usually takes longer than the original cover tree and the simplified cover tree. (b) Construction plus querying is faster in the nearest ancestor cover tree. On most datasets, this faster query time more than offsets the increased construction cost, giving an overall speedup.

![Figure 4](image2.png)

**Figure 4.** (a) Comparison of our packed nearest ancestor cover tree to our unpacked tree and other implementations, demonstrating better cache performance. (b) A stalled CPU cycle is when the CPU does no work because it must wait for a memory access. Reducing the number of cache misses results in fewer stalled cycles, and so faster run times. We used the Linux `perf stat` utility to measure the `cache-references`, `cache-misses`, `cycles`, and `stalled-cycles-fronte` hardware counters. `perf stat` uses a sampling strategy with negligible affect on program performance.

![Figure 5](image3.png)

**Figure 5.** Run times on the “all nearest neighbor” procedure for only those datasets that take more than 5 minutes. (a) Tree construction. A single cover tree merge takes about 1% of the computation time; the main reason for the lack of perfect parallel speedup is the increased number of cache misses caused by inserting into multiple trees simultaneously. (b) Comparison on total performance to reference and MLPack implementations. Runtimes in both figures are divided by that of our single processor implementation (shown in parenthesis).
Our code can be downloaded at http://github.com/mikeizbicki/hlearn#covertree.

2Our code can be downloaded at http://github.com/mikeizbicki/hlearn#covertree.
Figure 6. The effect on runtime as we increase the number of data points on the bioinformatics data. The relationship is roughly linear, indicating protein graphs have a relatively low intrinsic dimensionality. As expected, the nearest ancestor cover tree performs the best.

From this PDB file, we calculate the protein’s secondary structure using the DSSP tool (Joosten et al., 2011). Then, the tool VLPG (Schäfer et al., 2012) generates graphs from the resulting secondary structure. Some PDB files contain information for multiple graphs, and some do not contain enough information to construct a graph. In total, our dataset consists of 250,000 graphs, and a typical graph has between 5-120 nodes and 0.1-3 edges per node. Figure 6 shows the scaling behavior of all three cover trees on this dataset. On all of the data, the total construction and query cost are 29% that of the original cover tree.

6.4. Earth mover’s distance

The Earth Mover’s Distance (EMD) is a distance metric between histograms designed for image classification (Rubner et al., 1998). In our tests, we convert images into three dimensional histograms of the pixel values in LabCIE color space. LabCIE is a color space represents colors in three dimensions. It is similar to the more familiar RGB and CMYK color spaces, but the distances between colors more accurately match what humans perceive color distances to be. We construct the histogram such that each dimension has 8 equally spaced intervals, for a total of 512 bins. We then create a “signature” of the histogram by recording only the 20 largest of the 512 bins.

Previous research on speeding up EMD focused on computing EMD distances faster. The EMD takes a base distance as a parameter. For an arbitrary base distance, EMD requires $O(b^3 \log b)$ time where $b$ is the size of the histogram signature. Faster algorithms exist for specific base metrics. For example, with an $L_1$ base metric the EMD can be computed in time $O(b^2)$ (Ling & Okada, 2007); and if the base metric is a so-called “thresholded metric,” we can get an order of magnitude constant factor speed up (Pele & Werman, 2009). We specifically chose the LabCIE color space because there is no known faster EMD algorithm. It will stress-test our cover tree implementation.

In this experiment, we use the Yahoo! Flickr Creative Commons dataset. The dataset contains 1.5 million images in its training set, and we construct simplified and nearest ancestor cover trees in parallel on this data. Construction times are shown in Table 2. Using the cheap L2 distance with smaller datasets, tree construction happens quickly and so parallelization is less important. But with an expensive distance on this larger dataset, parallel construction makes a big difference.

7. Conclusion

We’ve simplified the definition of the cover tree, and introduced the new nearest ancestor invariant that speeds up the cover tree in practice. It is possible that other invariants exist that will balance the tree better, providing even more speed improvements.

Acknowledgments

Shelton was supported by DARPA award FA8750-12-2-0010.

References


