Faster Cover Trees

Mike Izbicki MIZBI001@UCR.EDU
Christian R. Shelton CSHELTON@CS.UCR.EDU
University of California Riverside, 900 University Ave, Riverside, CA 92521

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

$$p_{nn} = \underset{q \in X - \{p\}}{\arg\min}\, d(p,q)$$

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* (Omohundro, 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*.

Proceedings of the 32nd International Conference on Machine Learning, Lille, France, 2015. JMLR: W&CP volume 37. Copyright 2015 by the author(s).

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* $\texttt{level}(p)$. *For each child q of p*
$$\texttt{level}(q) = \texttt{level}(p) - 1 \,.$$

2. The covering invariant. *For every node p, define the function* $\texttt{covdist}(p) = 2^{\texttt{level}(p)}$. *For each child q of p*
$$d(p,q) \le \texttt{covdist}(p) \,.$$

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

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

$$\texttt{maxdist}(p) = \operatorname*{arg\,max}_{q \in \texttt{descendants}(p)} d(p,q)$$

In words, this is the greatest distance from p to any of its descendants. This value is upper bounded by $2^{\texttt{level}(p)+1}$, 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: (*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 $O(c^6 \log n)$ for both the original and simplified cover tree.

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 $d(y,x) > d(y,q) - \texttt{maxdist}(q)$ then
5:         $y \leftarrow \texttt{findNearestNeighbor}(q,x,y)$
6: return $y$
```

Algorithm 2 Simplified cover tree insertion

```
function insert(cover tree $p$, data point $x$)
1: if $d(p,x) > \texttt{covdist}(p)$ then
2:     while $d(p,x) > 2\texttt{covdist}(p)$ do
3:         Remove any leaf $q$ from $p$
4:         $p' \leftarrow$ tree with root $q$ and $p$ as only child
5:         $p \leftarrow p'$
6:     return tree with $x$ as root and $p$ as only child
7: return insert_$(p,x)$
function insert_(cover tree $p$, data point $x$)
prerequisites: $d(p,x) \le \texttt{covdist}(p)$
1: for $q \in \texttt{children}(p)$ do
2:     if $d(q,x) \le \texttt{covdist}(q)$ then
3:         $q' \leftarrow \texttt{insert\_}(q,x)$
4:         $p' \leftarrow p$ with child $q$ replaced with $q'$
5:         return $p'$
6: return $p$ with $x$ added as a child
```

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 $\texttt{maxdist}(p) \le 2\texttt{covdist}(p)$, we are guaranteed that $d(p,x) \le \texttt{covdist}(x)$, and so we do not violate the covering constraint. In the second case, the $\texttt{insert\_}$ function recursively descends the tree. On each function call, we search through $\texttt{children}(p)$ 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 $\texttt{children}(p)$ 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

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

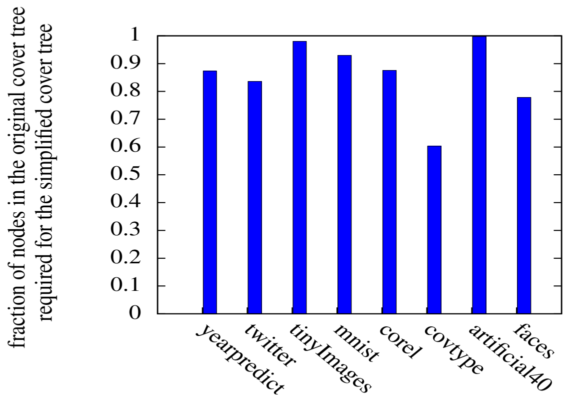

Figure 1. Fraction of nodes required for the simplified cover tree. Fewer nodes means less overhead from traversing the tree and fewer distance comparisons. See Table 1 for information on the datasets.

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 $O(c^6 \log n)$.

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 MLPack 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.

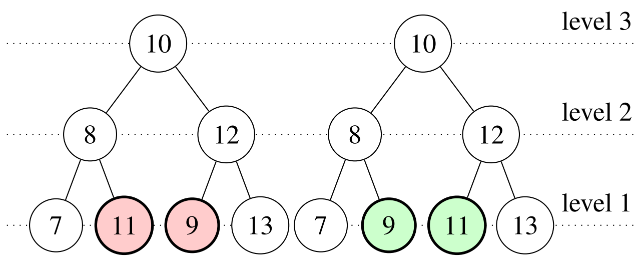

Figure 2. Using the metric $d(a,b) = |a-b|$, both trees are valid simplified cover trees; but only the right tree is a valid nearest ancestor cover tree. Moving the 9 and 11 nodes reduces the value of `maxdist` for their ancestor nodes. This causes pruning to happen more often during nearest neighbor queries.

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; q has a descendent r; and $d(r,x) < d(r,q)$; 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 `rebalance_` to each of q's children; we use the results to update the corresponding child, the *moveset* and the *stayset* variables. Finally, we try to reinsert any nodes in *stayset*. If `rebalance_` was called with q a child of p, then the return value of *stayset* will be empty; any children that could not be directly reinserted must be in the *moveset*.

The `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 `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 *moveset*, except to note that it is usually small in practice. On the datasets in our experiments (see Table 1), the value is usually zero or at worst in the single digits. Figure 3(a) shows that nearest ancestor cover tree construction is not that much slower in practice, and Figure 3(b) shows that this slowdown is overshadowed by the resulting speedup in nearest neighbor queries.

Algorithm 3 Nearest ancestor cover tree insertion

```
function insert_(cover tree $p$, data point $x$)
1:  for $q \in$ children(p) sorted by distance to $x$ do
2:      if $d(q,x) \leq$ covdist(q) then
3:          $q' \leftarrow$ insert_$(q,x)$
4:          $p' \leftarrow p$ with child $q$ replaced with $q'$
5:          return $p'$
6:  return rebalance$(p, x)$

function rebalance(cover trees $p$, data point $x$)
prerequisites: $x$ can be added as a child of $p$ without violating the covering or separating invariants
1:  create tree $x'$ with root node $x$ at level level(p) $- 1$ $x'$ contains no other points
2:  $p' \leftarrow p$
3:  for $q \in$ children(p) do
4:      $(q', moveset, stayset) \leftarrow$ rebalance_$(p,q,x)$
5:      $p' \leftarrow p'$ with child $q$ replaced with $q'$
6:      for $r \in moveset$ do
7:          $x' \leftarrow$ insert$(x', r)$
8:  return $p'$ with $x'$ added as a child

function rebalance_(cover trees $p$ and $q$, point $x$)
prerequisites: $p$ is an ancestor of $q$
1:  if $d(p,q) > d(q,x)$ then
2:      $moveset, stayset \leftarrow \emptyset$
3:      for $r \in$ descendants(q) do
4:          if $d(r,p) > d(r,x)$ then
5:              $moveset \leftarrow moveset \cup \{r\}$
6:          else
7:              $stayset \leftarrow stayset \cup \{r\}$
8:      return (null, $moveset, stayset$)
9:  else
10:     $moveset', stayset' \leftarrow \emptyset$
11:     $q' \leftarrow q$
12:     for $r \in$ children(q) do
13:         $(r', moveset, stayset) \leftarrow$ rebalance_$(p,r,x)$
14:         $moveset' \leftarrow moveset \cup moveset'$
15:         $stayset' \leftarrow stayset \cup stayset'$
16:         if $r' =$ null then
17:             $q' \leftarrow q$ with the subtree $r$ removed
18:         else
19:             $q' \leftarrow q$ with the subtree $r$ replaced by $r'$
20:     for $r \in stayset'$ do
21:         if $d(r,q)' \leq$ covdist(q)$'$ then
22:             $q' \leftarrow$ insert$(q', r)$
23:             $stayset' \leftarrow stayset' - \{r\}$
24:     return $(q', moveset', stayset')$
```

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. Previous work on parallelizing cover trees applied only to the GPU (Kumar & Ramareddy, 2010). Our 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 `merge` function's main purpose is to satisfy the prerequisites for `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 `insert` function.

The `mergeHelper` function returns a partially merged tree and a set of nodes called the *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 *uncovered* set contains all of q's children that would violate the covering invariant if inserted into p. The *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 `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 $d(p,q) \leq$ `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.

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: $(p, leftovers) \leftarrow$ mergeHelper$(p,q)$
 7: for $r \in leftovers$ do
 8:     $p \leftarrow$ insert$(p,r)$
 9: return $p$
function mergeHelper(cover tree $p$, cover tree $q$)
prereqs: level(p) $=$ level(q), $d(p,q) \leq$ covdist(p)
 1: $children' \leftarrow$ children(p)                    ▷ Phase 1
 2: $uncovered, sepcov, leftovers \leftarrow \emptyset$
 3: for $r \in$ children(q) do
 4:     if $d(p,r) <$ covdist(p) then
 5:         $foundmatch \leftarrow$ false
 6:         for $s \in children'$ do
 7:             if $d(s,r) \leq$ sepdist(p) then
 8:                 $(s', leftovers_s) \leftarrow$ mergeHelper$(s,r)$
 9:                 $children' \leftarrow children' \cup \{s'\} - \{s\}$
10:                 $leftovers \leftarrow leftovers \cup leftovers_s$
11:                 $foundmatch \leftarrow$ true
12:                 break from inner loop
13:         if not $foundmatch$ then
14:             $sepcov \leftarrow sepcov \cup \{r\}$
15:     else
16:         $uncovered \leftarrow uncovered \cup \{r\}$
17: $children' \leftarrow children' \cup sepcov$              ▷ Phase 2
18: $p' \leftarrow$ tree rooted at $p$ with children(p')=$children'$
19: $p' \leftarrow$ insert$(p',q)$
20: $leftovers' \leftarrow \emptyset$
21: for $r \in leftovers$ do
22:     if $d(r,p)' \leq$ covdist(p)$'$ then
23:         $p' \leftarrow$ insert$(p',r)$
24:     else
25:         $leftovers' \leftarrow leftovers' \cup \{r\}$
26: return $(p', leftovers' \cup uncovered)$
```

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 Embde 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.

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.

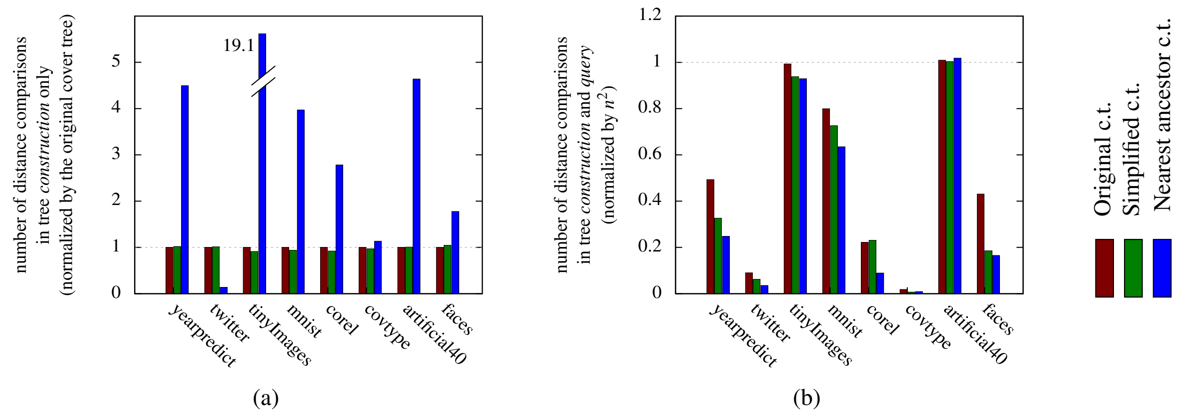

Figure 3. (a) Constructing a nearest ancestor query tree usually takes longer than the original cover tree and the simplified cover tree. (b) Construction *plus* querying is faster in the nearest ancestor cover tree. On most datasets, this faster query time more than offsets the increased construction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

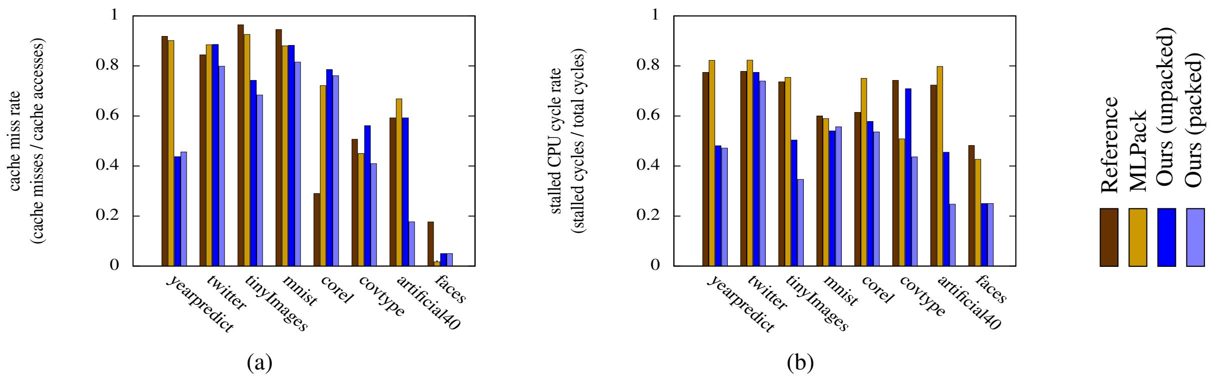

Figure 4. (a) Comparison of our packed nearest ancestor cover tree to our unpacked tree and other implementations, demonstrating better cache performance. (b) A stalled CPU cycle is when the CPU does no work because it must wait for a memory access. Reducing the number of cache misses results in fewer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

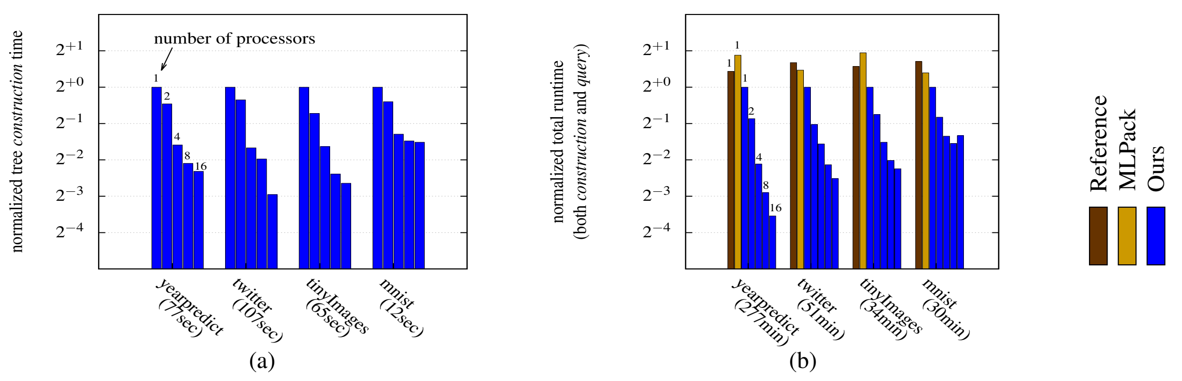

Figure 5. Run times on the "all nearest neighbor" procedure for only those datasets that take more than 5 minutes. (a) Tree construction. A single cover tree merge takes about 1% of the computation time; the main reason for the lack of perfect parallel speedup is the increased number of cache misses 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).

dataset	num data points	num dimensions
yearpredict	515345	90
twitter	583250	78
tinyImages	100000	384
mnist	70000	784
corel	68040	32
covtype	581012	55
artificial40	10000	40
faces	10304	20

Table 1. All MLPack benchmark datasets with at least 20 dimensions and 10000 points, arranged in descending order by runtime of all nearest neighbor search.

6. Experiments

We now validate our improvements empirically. Our first experiments use the Euclidean distance on a standard benchmark suite (described in Table 1). Our last experiments use non-Euclidean metrics on data from bioinformatics and computer vision. In each experiment, we use the "all nearest neighbors" experimental setup. That is, we first construct the cover trees on the dataset. Then, for each point in the dataset, we find its nearest neighbor. This is a standard technique for measuring the efficiency of nearest neighbor algorithms.

6.1. Tree-type comparison

Our first experiment compares the performance of the three types of cover trees: original, simplified, and nearest ancestor. We measure the number of distance comparisons required to build the tree on a dataset in Figure 3(a) and the number of distance comparisons required to find each data point's nearest neighbor in Figure 3(b) using Algorithm 1. Distance comparisons are a good proxy measure of runtime performance because the majority of the algorithm's runtime is spent computing distances, and it ignores the possible unwanted confounding variable of varying optimization efforts. As expected, the simplified tree typically outperforms the original tree, and the nearest ancestor tree typically outperforms the simplified tree. We reiterate that this reduced need for distance comparisons translates over to all other queries provided by the space tree framework (Curtin et al., 2013b).

6.2. Implementation comparison

We next compare our implementation against two good cover tree implementations currently in widespread use: the reference implementation used in the original paper (Beygelzimer et al., 2006) and MLPack's implementation (Curtin et al., 2013a). Both of these programs were written in C++ and compiled using `g++ 4.4.7` with full optimizations. Our implementation was written in Haskell and compiled with `ghc 7.8.4` also with full optimizations.[2] All tests were run on an Amazon Web Services `c3.8x-large` instance with 60 GB of RAM and 32 Intel Xeon E5-2680 CPU cores clocked at 2.80GHz. Half of those cores are hyperthreads, so for simplicity we only parallelize out to 16 cores.

Since the reference implementation and MLPack only come with the Euclidean distance built-in, we only use that metric when comparing the three implementations. Figure 4 shows the cache performance of all three libraries. Figure 5 shows the runtime of all three libraries. Our implementation's cache performance and parallelization speedup is shown on the nearest ancestor cover tree. Neither the original implementation nor MLPack support parallelization.

6.3. Graph kernels and protein function

An important problem in bioinformatics is to predict the function of a protein based on its 3d structure. State of the art solutions model the protein's 3d structure as a graph and use support vector machines (with a graph kernel) for prediction. Computing graph kernels is relatively expensive, however, so research has focused on making the graph kernel computation faster (Vishwanathan et al., 2010; Shervashidze et al., 2011). Such research makes graph kernels scale to *larger graphs*, but does not help in the case where there are *more graphs*. Our contribution is to use cover trees to reduce the number of required kernel computations, letting us scale to more graphs. The largest dataset in previous research contained about 1200 proteins. With our cover tree, we perform nearest neighbor queries on all one hundred thousand proteins currently registered in the Protein Data Bank (Berman et al., 2000).

We use the random walk graph kernel in our experiment. It performs well on protein classification and is conceptually simple. See Vishwanathan *et al.* (2010) for more details. A naive computation of this kernel takes time $O(v^6)$, where v is the number of vertices in the graph. Vishwanathan *et al.* present faster methods that take time only $O(v^3)$. While considerably faster, it is still a relatively expensive distance computation.

The Protein Data Bank (Berman et al., 2000) contains information on the 3d primary structure of approximately one hundred thousand proteins. To perform our experiment, we follow a procedure similar to that used by the PROTEIN dataset used in the experiments in Viswanathan *et al.*. This procedure constructs secondary structure graphs from the primary structures in the Protein Data Bank using the tool VLPG (Schäfer et al., 2012). The Protein Data Bank stores the 3d structure of the atoms in the protein in a PDB file.

[2]Our code can be downloaded at `http://github.com/mikeizbicki/hlearn#covertree`.

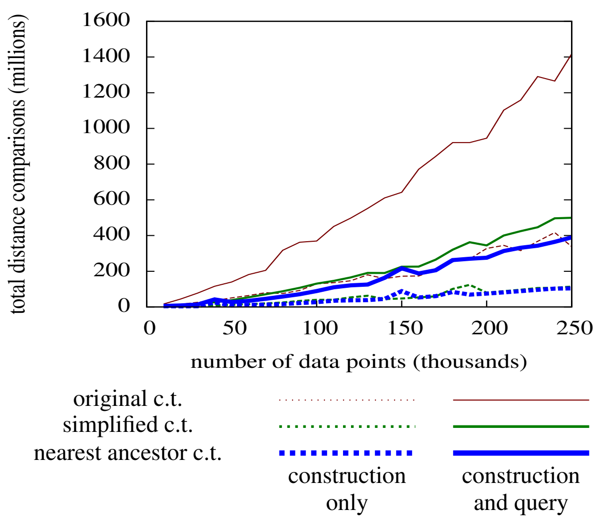

Figure 6. The effect on runtime as we increase the number of data points on the bionformatics data. The relationship is roughly linear, indicating protein graphs have a relatively low intrinsic dimensionality. As expected, the nearest ancestor cover tree performs the best.

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.

number of cores	simplified tree construction time	simplified tree construction speedup	nearest ancestor tree construction time	nearest ancestor tree construction speedup
1	70.7 min	1.0	210.9 min	1.0
2	36.6 min	1.9	94.2 min	2.2
4	18.5 min	3.8	48.5 min	4.3
8	10.2 min	6.9	25.3 min	8.3
16	6.7 min	10.5	12.0 min	17.6

Table 2. Parallel cover tree construction using the earth movers distance. On this large dataset with an expensive metric, we see better parallel speedup than on the datasets with the cheaper L2 metric. The nearest ancestor cover tree gets super-linear parallel speedup because we are merging with Algorithm 4, which does not attempt to rebalance.

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

Berman, Helen M., Westbrook, John, Feng, Zukang, Gilliland, Gary, Bhat, T. N., Weissig, Helge, Shindyalov, Ilya N., and Bourne, Philip E. The Protein Data Bank. *Nucleic Acids Research*, 28(1):235–242, 2000.

Beygelzimer, Alina, Kakade, Sham, and Langford, John. Cover trees for nearest neighbor. In *Proceedings of the 23rd International Conference on Machine Learning*, pp. 97–104, New York, NY, USA, 2006.

Curtin, Ryan R., Cline, James R., Slagle, Neil P., March, William B., Ram, P., Mehta, Nishant A., and Gray, Alexander G. MLPACK: A scalable C++ machine learning library. *Journal of Machine Learning Research*, 14: 801–805, 2013a.

Curtin, Ryan R, March, William B, Ram, Parikshit, Anderson, David V, Gray, Alexander G, and Isbell Jr, Charles L. Tree-independent dual-tree algorithms. In *Proceedings of The 30th International Conference on Machine Learning*, pp. 1435–1443, 2013b.

Friedman, Jerome H, Bentley, Jon Louis, and Finkel, Raphael Ari. An algorithm for finding best matches in logarithmic expected time. *Transactions on Mathematical Software*, 3(3):209–226, 1977.

Frigo, Matteo, Leiserson, Charles E, Prokop, Harald, and Ramachandran, Sridhar. Cache-oblivious algorithms. In *Foundations of Computer Science, 1999. 40th Annual Symposium on*, pp. 285–297, 1999.

Joosten, Robbie P., te Beek, Tim A. H., Krieger, Elmar, Hekkelman, Maarten L., Hooft, Rob W. W., Schneider, Reinhard, Sander, Chris, and Vriend, Gert. A series of PDB related databases for everyday needs., January 2011.

Karger, David R. and Ruhl, Matthias. Finding nearest neighbors in growth-restricted metrics. In *In 34th Annual ACM Symposium on the Theory of Computing*, pp. 741–750, 2002.

Krauthgamer, Robert and Lee, James R. Navigating nets: Simple algorithms for proximity search. In *Proceedings of the Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 798–807, Philadelphia, PA, USA, 2004.

Kumar, R. Deepak and Ramareddy, K. Design and implementation of cover tree algorithm on cuda-compatible gpu. *International Journal of Computer Applications*, 3 (7):24–27, June 2010. Published By Foundation of Computer Science.

Ling, Haibin and Okada, Kazunori. An efficient earth mover's distance algorithm for robust histogram comparison. *PAMI*, 29:853, 2007.

Lisitsyn, Sergey, Widmer, Christian, and Garcia, Fernando J. Iglesias. Tapkee: An efficient dimension reduction library. *Journal of Machine Learning Research*, 14:2355–2359, 2013.

Omohundro, Stephen Malvern. Five balltree construction algorithms. Technical Report 89-063, International Computer Science Institute, December 1989.

Pele, Ofir and Werman, Michael. Fast and robust earth mover's distances. In *ICCV*, 2009.

Ram, Parikshit, Lee, Dongryeol, March, William, and Gray, Alexander G. Linear-time Algorithms for Pairwise Statistical Problems. In *Advances in Neural Information Processing Systems*, 2010.

Rubner, Yossi, Tomasi, Carlo, and Guibas, Leonidas J. A metric for distributions with applications to image databases. In *Sixth International Conference on Computer Vision*, pp. 59–66, 1998.

Schäfer, Tim, May, Patrick, and Koch, Ina. Computation and Visualization of Protein Topology Graphs Including Ligand Information. In *German Conference on Bioinformatics 2012*, volume 26, pp. 108–118, Dagstuhl, Germany, 2012.

Segata, Nicola and Blanzieri, Enrico. Fast and scalable local kernel machines. *Journal of Machine Learning Research*, 11:1883–1926, August 2010.

Shervashidze, Nino, Schweitzer, Pascal, van Leeuwen, Erik Jan, Mehlhorn, Kurt, and Borgwardt, Karsten M. Weisfeiler-lehman graph kernels. *Journal of Machine Learning Research*, 12:2539–2561, November 2011.

Tziortziotis, Nikolaos, Dimitrakakis, Christos, and Blekas, Konstantinos. Cover Tree Bayesian Reinforcement Learning. *Journal of Machine Learning Research*, 15, 2014.

Vishwanathan, S. V. N., Schraudolph, Nicol N., Kondor, Risi, and Borgwardt, Karsten M. Graph kernels. *Journal of Machine Learning Research*, 11:1201–1242, August 2010.