Faster dual-tree traversal for nearest neighbor search

Ryan R. Curtin

Georgia Institute of Technology, Atlanta GA 30332, USA,
ryan@ratml.org

Abstract. Nearest neighbor search is a nearly ubiquitous problem in computer science. When nearest neighbors are desired for a query set instead of a single query point, dual-tree algorithms often provide the fastest solution, especially in low-to-medium dimensions (i.e. up to a hundred or so), and can give exact results or absolute approximation guarantees, unlike hashing techniques. Using a recent decomposition of dual-tree algorithms into modular pieces, we propose a new piece: an improved traversal strategy; it is applicable to any dual-tree algorithm. Applied to nearest neighbor search using both kd-trees and ball trees, the new strategy demonstrably outperforms the previous fastest approaches. Other problems the traversal may easily be applied to include kernel density estimation and max-kernel search.

1 Introduction

The task of nearest neighbor search arises continually in machine learning, data mining, and related domains. For instance, many computer vision algorithms require forms of similarity search [1]; recommendation systems may use k-nearest-neighbor search internally: BellKor's Netflix prize solution does this [2]. Nearest neighbors are also often used in machine learning applications as simple classifiers [3]; more advanced machine learning techniques may also depend on the calculation of nearest neighbors [4].

To formally describe the problem, take S_r to be the reference set. The nearest neighbor search task is, for a given query point p_q, find $\text{argmin}_{p_r \in S_r} d(p_q, p_r)$ for some metric $d(\cdot, \cdot)$.[1] The most straightforward technique for solving this problem is a linear scan over all points in S_r, but for large S_r—or for situations where answers are desired not just for one query point p_q but instead an entire query set S_q—this approach is computationally infeasible. Given $|S_r| = N$, a result for a single query point p_q takes $O(N)$ time.

Owing to both this computational difficulty and the wide applicability of nearest neighbor search, much ink has been spilled describing fast algorithms to solve the nearest neighbor search problem. The first fast algorithms for nearest-neighbor search were based on tree structures [5] [6], where some type of tree

[1] Extending this to the k-nearest neighbor search task is straightforward: replace argmin with k argmin.

structure is built on the reference set S_r and then, to find the nearest neighbor of a query point p_q, a branch-and-bound algorithm is used. Other popular approaches include the use of nets [7] and also locality-sensitive hashing [8] [9] [10]. In general, nets and hashing give approximate solutions, whereas tree-based approaches can give both approximate and exact solutions.

In the situation where there is a query set S_q and not just a single query point p_q, it often makes sense to build a tree on *both* the reference set S_r and the query set S_q, and simultaneously traverse both the query and reference trees. This type of approach is known as a *dual-tree algorithm* [11] [12], and is generally the fastest known way to perform nearest-neighbor search, for sufficiently large query sets in low-to-medium dimensions (i.e. up to a hundred or so, depending on the type of tree and the properties of the dataset). Further, when cover trees are used, and $S_q \sim O(N)$, search time for *all* points in S_q is worst-case $O(N)$ [13] [14]; though, this bound depends on dataset-dependent quantities.

Dual-tree algorithms exist for problems other than nearest neighbor search; some examples include range search [12], kernel density estimation [15], minimum spanning tree calculation [16], mean shift clustering [17], kernel summations [18], max-kernel search [19], and other problems [20] [21] [22]. Thus, results for any dual-tree algorithm are often readily applied to other dual-tree algorithms.

Curtin et al. recently proposed a generalizing abstraction for all dual-tree algorithms, which allows dual-tree algorithms to be understood as four separate components: a type of tree, a dual-tree traversal, a problem-specific pruning rule, and a problem-specific base case [12]. This convenient, modular abstraction lets us focus on only one component at a time, independent of the other three pieces.

For tree-based nearest neighbor search, whether single-tree or dual-tree, the order that tree nodes are visited makes a noticeable difference in both the quality of the results (for approximate search) and the speed of the results. This is why single-tree algorithms such as the original *kd*-tree nearest neighbor search algorithm [5] first recurse into the nearest node to a query point.

In this paper, we exploit the tree-independent dual-tree algorithm abstraction in order to develop an improved general depth-first dual-tree traversal. By applying this traversal to the problem of nearest-neighbor search, we obtain significant speedup over previous dual-tree traversal strategies, and outperform competing strategies, such as single-tree search and LSH, in both the approximate and exact nearest neighbor search tasks. Because of the traversal's generality, it can be applied to problems other than just nearest neighbor search.

2 Trees

First, we must introduce the concepts underlying dual-tree algorithms more formally, and we must also introduce notation. As in [12] and more recent contributions [14] [23], we will use the tree-independent dual-tree algorithm framework. This means that given some dual-tree algorithm that works on a set of query points S_q and a set of reference points S_r, we may understand this algorithm as the combination of four distinct parts:

- A type of *space tree*.

– A *pruning dual-tree traversal*, which visits combinations of nodes from the query tree and reference tree, and is parameterized by a `BaseCase()` and `Score()` function.
– A `Score()` function, which determines if a combination of two nodes can be pruned.
– A `BaseCase()` function, which defines the action to take on a combination of query point and reference point.

If we have each of these four pieces, then, we may assemble a dual-tree algorithm: using a pruning dual-tree traversal with the given `BaseCase()` and `Score()` functions on two space trees that are built on the query and reference sets will yield a working dual-tree algorithm. A formal definition of each of these components is necessary for complete understanding. These definitions are taken from the original introduction of Curtin et. al. [12].

Definition 1 *A* space tree *on a dataset $S \in \mathcal{R}^{n \times d}$ is a rooted, undirected, connected, acyclic simple graph satisfying the following properties:*

– *Each node (or vertex) holds a number of points (possibly zero) and is connected to one parent node and a number of child nodes (possibly zero).*
– *There is one node in every space tree with no parent; this is the root of the tree.*
– *Each point in S is contained in at least one node of the tree.*
– *Each node $\mathscr{N}$ of the tree corresponds to a convex subset of $\mathcal{R}^d$ that contains each of the points in the node as well as each of the convex subsets corresponding to each child of the node.*

Most tree structures in the literature fall under the umbrella definition of a space tree: *kd*-trees [5], PCA trees, metric trees, cover trees, R trees and variants, and even spill trees [24], where the subsets of child nodes are allowed to overlap. In this document formal script letters will be used to notate trees and corresponding quantities; this is the same notation used in [12]. In specific,

– A node in a tree will be denoted with the letter $\mathscr{N}$.
– For some node $\mathscr{N}_i$, the set of children of $\mathscr{N}_i$ will be denoted $\mathscr{C}_i$.
– For some node $\mathscr{N}_i$, the set of points contained in $\mathscr{N}_i$ will be denoted $\mathscr{P}_i$.
– The convex subset of $\mathcal{R}^d$ corresponding to the node $\mathscr{N}_i$ will be denoted $\mathscr{S}_i$.
– For some node $\mathscr{N}_i$, the set of descendant nodes of $\mathscr{N}_i$ will be denoted $\mathscr{D}_i^n$. This set is defined as $\mathscr{C}(\mathscr{N}_i) \cup \mathscr{C}(\mathscr{C}(\mathscr{N}_i)) \cup \ldots$.
– For some node $\mathscr{N}_i$, the set of descendant points of $\mathscr{N}_i$ will be denoted $\mathscr{D}_i^p$. This set is defined as $\mathscr{P}_i \cup \mathscr{P}(\mathscr{D}_i^p)$.

The utility of trees stems from the ability to quickly place bounds on various distance-related quantities for a single node. Consider two space tree nodes $\mathscr{N}_i$ and $\mathscr{N}_j$, and suppose our task is to find the minimum distance between any two descendant points in the nodes:

$$d_{\min}(\mathscr{N}_i, \mathscr{N}_j) = \min_{p_i \in \mathscr{D}_i^p, p_j \in \mathscr{D}_j^p} d(p_i, p_j). \tag{1}$$

Now, suppose $\mathscr{S}_i$ is a ball centered at some point $\mu_i \in \mathcal{R}^d$ with radius λ_i, and $\mathscr{S}_j$ is a ball centered at some point $\mu_j \in \mathcal{R}^d$ with radius λ_j. Then, we may easily place a lower bound: $d_{\min}(\mathscr{N}_i, \mathscr{N}_j) \geq d(\mu_i, \mu_j) - \lambda_i - \lambda_j$. This bound may be calculated with just one distance calculation, instead of $|\mathscr{D}_i^p||\mathscr{D}_j^p|$ distance calculations. During the traversal, bounds like the one on $d_{\min}(\mathscr{N}_i, \mathscr{N}_j)$ are often used to prune away large amounts of work.

3 Traversals

Next, we formally introduce the notion of a dual-tree traversal, again from [12].

Definition 2 *A pruning dual-tree traversal is a process that, given two space trees $\mathscr{T}_q$ (the query tree, built on the query set S_q) and $\mathscr{T}_r$ (the reference tree, built on the reference set S_r), will visit combinations of nodes $(\mathscr{N}_q, \mathscr{N}_r)$ such that $\mathscr{N}_q \in \mathscr{T}_q$ and $\mathscr{N}_r \in \mathscr{T}_r$ no more than once, and call a function `Score(`$\mathscr{N}_q$`, `$\mathscr{N}_r$`)` to assign a score to that node. If the score is ∞ (or above some bound), the combination is pruned and no combinations $(\mathscr{N}_{qc}, \mathscr{N}_{rc})$ such that $\mathscr{N}_{qc} \in \mathscr{D}_q^n$ and $\mathscr{N}_{rc} \in \mathscr{D}_r^n$ are visited. Otherwise, for every combination of points (p_q, p_r) such that $p_q \in \mathscr{P}_q$ and $p_r \in \mathscr{P}_r$, a function `BaseCase(`p_q`, `p_r`)` is called. If no node combinations are pruned during the traversal, `BaseCase(`p_q`, `p_r`)` is called at least once on each combination of $p_q \in S_q$ and $p_r \in S_r$.*

Although the definition is quite complex, real-world dual-tree traversals tend to be straightforward. The standard depth-first dual-tree traversal is shown in Algorithm 1; this is the same traversal used in most dual-tree algorithms that use the *kd*-tree [11] [16] [18][2] and is often used in practice [25]. Generally, a depth-first traversal is preferred because many space trees in practice only hold points in the leaves; breadth-first traversals do not perform well in these situations.

The traversal is originally called with the root of the query tree $\mathscr{T}_q$ and the root of the reference tree $\mathscr{T}_r$. First, `BaseCase()` is called with every pair of query and reference points (lines 4–6). Then, for recursion, we collect a list of combinations to recurse into, sorted by their score. Any combinations with score ∞ are not recursed into. If both nodes have children, then we recurse into combinations of query children and reference children. If only the reference node has children, we recurse into combinations of the query node and the reference children. If only the query node has children, we recurse into combinations of the query children and the reference node. If neither node has children, there is no need to recurse.

The algorithm first recurses into those node combinations with lowest score. Depending on the task being solved (that is, which `Score()` and `BaseCase()` functions are being used), this prioritized approach to recursion can provide significant speedup over unprioritized recursion. For nearest neighbor search, a prioritized recursion gives significantly faster results.

[2] The algorithms in each of the referenced papers tend to look very different because they are not derived in a tree-independent form, but using the *kd*-tree with the traversal in Algorithm 1 and simplifying will yield the same algorithm.

Algorithm 1 `DualDepthFirstTraversal(`$\mathscr{N}_q$`,` $\mathscr{N}_r$`)`.

Input: query node $\mathscr{N}_q$, reference node $\mathscr{N}_r$
Output: none

{Perform base cases for points in node combination.}
for all $p_q \in \mathscr{P}_q$ **do**
  for all $p_r \in \mathscr{P}_r$ **do**
    `BaseCase(`p_q`,` p_r`)`

{Assemble list of combinations to recurse into.}
$q \leftarrow$ empty priority queue
if $\mathscr{N}_q$ and $\mathscr{N}_r$ both have children **then**
  for all $\mathscr{N}_{qc} \in \mathscr{C}_q$ **do**
    for all $\mathscr{N}_{rc} \in \mathscr{C}_r$ **do**
      $s_i \leftarrow$ `Score`$(\mathscr{N}_{qc}, \mathscr{N}_{rc})$
      if $s_i \neq \infty$ **then** push $(\mathscr{N}_{qc}, \mathscr{N}_{rc})$ into q with priority $1/s_i$
else if $\mathscr{N}_q$ has children but $\mathscr{N}_r$ does not **then**
  for all $\mathscr{N}_{qc} \in \mathscr{C}_q$ **do**
    $s_i \leftarrow$ `Score`$(\mathscr{N}_{qc}, \mathscr{N}_r)$
    if $s_i \neq \infty$ **then** push $(\mathscr{N}_{qc}, \mathscr{N}_r)$ into q with priority $1/s_i$
else if $\mathscr{N}_q$ does not have children but $\mathscr{N}_r$ does **then**
  for all $\mathscr{N}_{rc} \in \mathscr{C}_r$ **do**
    $s_i \leftarrow$ `Score`$(\mathscr{N}_q, \mathscr{N}_{rc})$
    if $s_i \neq \infty$ **then** push $(\mathscr{N}_q, \mathscr{N}_{rc})$ into q with priority $1/s_i$

{Recurse into combinations with highest priority first.}
for all $(\mathscr{N}_{qi}, \mathscr{N}_{ri}) \in q$, highest priority first **do**
  `DualDepthFirstTraversal(`$\mathscr{N}_{qi}$`,` $\mathscr{N}_{ri}$`)`

4 Nearest neighbor search

With the notions of space tree and dual-tree traversal established, we may now introduce the problem-specific `BaseCase()` and `Score()` functions used to perform dual-tree nearest neighbor search. These are the same rules introduced by Curtin et. al. [12] and used in **mlpack** [25]. The rules depend on auxiliary arrays N and D; during the traversal, $N[p_q]$ holds the current nearest neighbor candidate for query point p_q, and $D[p_q]$ holds $d(p_q, N[p_q])$. At the beginning of the traversal, each element in D is initialized to ∞. At the end of the traversal, $N[p_q]$ will hold the nearest neighbor of p_q, and $D[p_q]$ will hold the distance between p_q and its nearest neighbor.

The `BaseCase()` function (Algorithm 2) receives a query point p_q and a reference point p_r as input. The distance between the points is calculated, and if this is better than the current best candidate distance for p_q, $d(p_q, p_r)$ is taken as the new best candidate distance and p_r as the new nearest neighbor candidate.

The `Score()` function is significantly more complex due to the bound function $B_{df}(\mathscr{N}_q)$[3]. Given a query node $\mathscr{N}_q$ and a reference node $\mathscr{N}_r$, we can prune

[3] Our formulation here is specialized for depth-first traversals, unlike some more general formulations [12]. We are only considering depth-first traversals in this work, though, so there is no need to introduce a more complicated bound function.

Algorithm 2 `BaseCase(`p_q`,` p_r`)` for nearest neighbor search.

Input: query point p_q, reference point p_r, candidate point $N[p_q]$, candidate distance $D[p_q]$
Output: none

if $d(p_q, p_r) < D[p_q]$ **then**
    $N[p_q] \leftarrow p_r$
    $D[p_q] \leftarrow d(p_q, p_r)$

Algorithm 3 `Score(`$\mathscr{N}_q$`,` $\mathscr{N}_r$`)` for nearest neighbor search.

Input: query node $\mathscr{N}_q$, reference node $\mathscr{N}_r$
Output: a score for the node combination $(\mathscr{N}_q, \mathscr{N}_r)$, or ∞ if it should be pruned

if $d_{\min}(\mathscr{N}_q, \mathscr{N}_r) > B_{df}(\mathscr{N}_q)$ **then**
    return ∞
return $d_{\min}(\mathscr{N}_q, \mathscr{N}_r)$

if we can determine that no descendant point of $\mathscr{N}_r$ can possibly be the nearest neighbor of any descendant point of $\mathscr{N}_q$. If we had perfect knowledge, this condition is easily expressed; we would prune if

$$d_{\min}(\mathscr{N}_q, \mathscr{N}_r) > \max_{p_q \in \mathscr{D}_q^p} D[p_q]. \tag{2}$$

But of course, because this requires looping over every descendant point in $\mathscr{N}_q$, we cannot calculate this every time `Score()` is called. Instead, we can use caching. Define the depth-first traversal bound function, $B_{df}(\mathscr{N}_q)$, recursively:

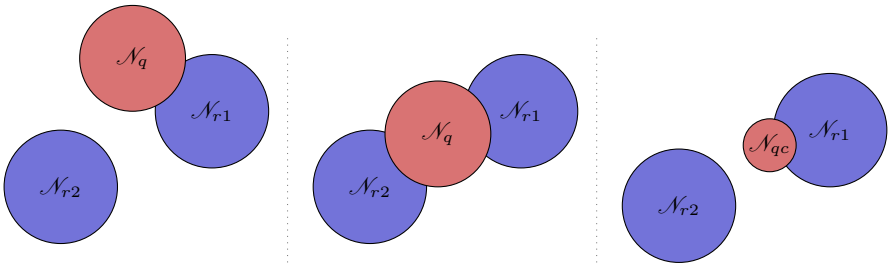
$$B_{df}(\mathscr{N}_q) = \max \left\{ \max_{p_q \in \mathscr{P}_q} D[p_q], \max_{\mathscr{N}_{qc} \in \mathscr{C}_q} B_{df}(\mathscr{N}_{qc}) \right\}. \tag{3}$$

When we visit a node combination $(\mathscr{N}_q, \mathscr{N}_r)$, we may cache the result of the calculation $B_{df}(\mathscr{N}_q)$, for use by subsequent calls to `Score()`. Then, a call to `Score()` takes just one distance calculation ($d_{\min}(\mathscr{N}_q, \mathscr{N}_r)$) and $|\mathscr{P}_q| + |\mathscr{C}_q|$ accesses. Proving the correctness of this algorithm is straightforward.

We may use this to construct a generalized dual-tree algorithm for nearest neighbor search. Any type of space tree can be paired with any type of pruning dual-tree traversal that uses the `BaseCase()` and `Score()` above, and correct nearest-neighbor search results will be obtained. With this algorithm established, we will now turn towards improving the depth-first dual-tree recursion strategy.

5 Delaying reference recursion

Algorithm 1 is the standard depth-first dual-tree traversal that is used in practice, and it prioritizes recursions: node combinations with lower scores (from `Score()`) are recursed into first. So, for instance, consider nearest neighbor search, where the result of `Score()`, if the node combination is not pruned, is $d_{\min}(\mathscr{N}_q, \mathscr{N}_r)$. In the situation depicted in Figure 1(a), combination $(\mathscr{N}_q, \mathscr{N}_{r1})$



(a) $\mathcal{N}_q$ closer to $\mathcal{N}_{r1}$. (b) $\mathcal{N}_q$ equidistant. (c) $\mathcal{N}_{qc}$ is not equidistant.

Fig. 1. Different situations for recursion.

should be visited before combination $(\mathcal{N}_q, \mathcal{N}_{r2})$. It is clear that this is the right choice, because a depth-first traversal of $(\mathcal{N}_q, \mathcal{N}_{r1})$ is more likely to tighten the bound $B_{df}(\mathcal{N}_q)$ such that $(\mathcal{N}_q, \mathcal{N}_{r2})$ can be pruned when it is recursed into.

But, consider a more tricky case, depicted in Figure 1(b). Here, $d_{\min}(\mathcal{N}_q, \mathcal{N}_{r1}) = d_{\min}(\mathcal{N}_q, \mathcal{N}_{r2}) = 0$, so we are unable to tell whether it is better to recurse into $(\mathcal{N}_q, \mathcal{N}_{r1})$ first or into $(\mathcal{N}_q, \mathcal{N}_{r2})$ first. Indeed, Algorithm 1 will select arbitrarily. This situation may occur in Algorithm 1 from lines 11 to 13 if, for a given child query node $\mathcal{N}_{qc}$, two or more reference children $\mathcal{N}_{rc}$ have the same score s_i.

We can do better than arbitrary selection. Consider some child $\mathcal{N}_{qc}$ of $\mathcal{N}_q$. Figure 1(c) shows an example $\mathcal{N}_{qc}$. In this example, the choice is now clear: the combination $(\mathcal{N}_{qc}, \mathcal{N}_{r1})$ should be recursed into before $(\mathcal{N}_{qc}, \mathcal{N}_{r2})$. Thus, the correct answer to the question "should we recurse into $(\mathcal{N}_q, \mathcal{N}_{r1})$ or $(\mathcal{N}_q, \mathcal{N}_{r2})$ first?" is to sidestep the question entirely: we should not recurse in the reference node, but instead in the query node. Then, at the level of the query child, the decision may be clearer.

In essence, the strategy is to delay recursion in the reference nodes until it is clear which reference node should be recursed into first. This improvement, once generalized, is encapsulated in Algorithm 4. Lines 15–20 check if reference recursion should be delayed because the scores of all reference children are identical. If so, the recursion will proceed by recursing only in the queries. If necessary, this reference recursion delay will continue until no longer possible. This delay is not possible when the query node does not have any children. This improved strategy can make a huge difference in the performance of the algorithm; recursing into a suboptimal reference child first can cause the bound $B_{df}(\cdot)$ to be unnecessarily loose, whereas first recursing into the best reference child will tighten $B_{df}(\cdot)$ more quickly and possibly allow other reference children to be pruned entirely.

For trees such as the *kd*-tree where each node has two children only, the extra implementation overhead for this strategy is trivial and simplifies to the addition of a single `if` statement. However, note that there are some situations where the modified traversal will not outperform the original prioritized traversal. For instance, for nearest neighbor search, if the query tree is identical to the reference tree and nodes in the tree cannot overlap, then it is very unlikely that the situation described in Figure 1(a) will be encountered: during the recursion, the query node will only overlap itself and possibly be adjacent to a sibling node.

Algorithm 4 `ImprovedDualDepthFirstTraversal(`$\mathscr{N}_q$`,` $\mathscr{N}_r$`).`

Input: query node $\mathscr{N}_q$, reference node $\mathscr{N}_r$
Output: none

{Perform base cases for points in node combination.}
for all $p_q \in \mathscr{P}_q$ **do**
  for all $p_r \in \mathscr{P}_r$ **do**
    `BaseCase(`p_q`,` p_r`)`

{Assemble list of combinations to recurse into.}
$q \leftarrow$ empty priority queue
if $\mathscr{N}_q$ and $\mathscr{N}_r$ both have children **then**
  for all $\mathscr{N}_{qc} \in \mathscr{C}_q$ **do**
    $q_{qc} \leftarrow \{\}$
    for all $\mathscr{N}_{rc} \in \mathscr{C}_r$ **do**
      $s_i \leftarrow$ `Score(`$\mathscr{N}_{qc}$,$\mathscr{N}_{rc}$`)`
      if $s_i \neq \infty$ **then** push $(\mathscr{N}_{qc}, \mathscr{N}_{rc}, s_i)$ into q_{qc}
    if all elements of q_{qc} have identical score **then**
      $s_i \leftarrow$ `Score(`$\mathscr{N}_{qc}$,$\mathscr{N}_r$`)`
      push $(\mathscr{N}_{qc}, \mathscr{N}_r)$ into q with priority $1/s_i$
    else
      for all $(\mathscr{N}_{qi}, \mathscr{N}_{ri}, s_i) \in q_{qc}$ **do**
        push $(\mathscr{N}_{qi}, \mathscr{N}_{ri})$ into q with priority $1/s_i$
else if $\mathscr{N}_q$ has children but $\mathscr{N}_r$ does not **then**
  for all $\mathscr{N}_{qc} \in \mathscr{C}_q$ **do**
    $s_i \leftarrow$ `Score(`$\mathscr{N}_{qc}$,$\mathscr{N}_r$`)`
    if $s_i \neq \infty$ **then** push $(\mathscr{N}_{qc}, \mathscr{N}_r)$ into q with priority $1/s_i$
else if $\mathscr{N}_q$ does not have children but $\mathscr{N}_r$ does **then**
  for all $\mathscr{N}_{rc} \in \mathscr{C}_r$ **do**
    $s_i \leftarrow$ `Score(`$\mathscr{N}_q$,$\mathscr{N}_{rc}$`)`
    if $s_i \neq \infty$ **then** push $(\mathscr{N}_q, \mathscr{N}_{rc})$ into q with priority $1/s_i$

{Recurse into combinations with highest priority first.}
for all $(\mathscr{N}_{qi}, \mathscr{N}_{ri}) \in q$, highest priority first **do**
  `ImprovedDualDepthFirstTraversal(`$\mathscr{N}_{qi}$`,` $\mathscr{N}_{ri}$`)`

6 Experiments

To test the efficiency of this strategy, we will observe the performance of our recursion strategy on the tasks of exact and approximate nearest neighbor search, with multiple types of trees, and with many different datasets. For approximate search, we compare with LSH (locality-sensitive hashing). The datasets utilized in these experiments are described in Table 1. Each dataset is from the UCI dataset repository [26], with the exception of the birch3 dataset [27], LCDM dataset [28], and SDSS-DR6 dataset [29].

The first test will focus on the task of exact nearest neighbor search: Algorithms 2 and 3 paired with a type of tree and traversal. Using the flexible **mlpack** library [25], we test with the *kd*-tree and the ball tree, using three dual-tree traversal strategies: a depth-first unordered recursion (equivalent to Algorithm 1 where the recursion priority is ignored); the standard depth-first

prioritized recursion (Algorithm 1); and our improved recursion (Algorithm 4). In addition, a single-tree algorithm is used; this is the canonical tree-based nearest neighbor search algorithm [5] with a prioritized recursion, run once for each query point. The dataset is randomly split into 60% reference set and 40% query set, and the algorithm is run ten times. The number of distance evaluations and the total runtime are collected. Table 2 shows the average number of distance calculations for each algorithm and the average runtime for each algorithm.

We can see from the results that our improvement is, in many cases, significant. In the best case, it gives more than 2x speedup over the next fastest strategy. This effect is especially pronounced on larger datasets, which will have deeper trees: a bad recursion decision early on can significantly affect the ability to prune during the algorithm. Ball trees exhibit less pronounced effects. This is because the bounding structure is a ball of fixed radius, whereas the *kd*-tree is adaptive in all dimensions. Therefore, two child nodes of a ball tree node may overlap, causing the improved strategy of delaying reference recursions to not pay off at lower levels. Nonetheless, especially for large datasets, where the dual-tree strategy is faster than the single-tree strategy, the improved traversal is a clear best choice.

Dataset	n	d
cloud	2048	10
winequality	6497	11
birch3	100000	2
miniboone	130064	50
covertype	581012	55
power	2075259	7
lcdm	16777216	3
sdss-dr6	39761242	4

Table 1. Dataset information.

The second task is approximate nearest neighbor search, and in this situation we will also be able to compare with locality-sensitive hashing. Relative-value approximation means that for an approximation parameter ϵ, we are guaranteed for a query point p_q with true nearest neighbor p_r^*, the algorithm will return an approximate nearest neighbor $\hat{p}_r$ such that $d(p_q, \hat{p}_r) \leq (1+\epsilon)d(p_q, p_r^*)$. It is

algorithm	**cloud**	**winequality**	**birch3**	**miniboone**
kd-tree, unordered	0.036s (270k)	0.288s (2.15M)	7.310s (62.2M)	62.481s (214M)
kd-tree, prioritized	0.005s (34.2k)	0.039s (222k)	0.419s (2.90M)	25.081s (78.8M)
kd-tree, improved	0.005s (**27.7k**)	0.021s (**104k**)	0.201s (**1.10M**)	12.643s (34.5M)
single *kd*-tree	0.005s (32.9k)	**0.017s** (112k)	0.262s (1.65M)	**6.637s (19.2M)**
ball tree, unordered	0.011s (356k)	0.104s (3.08M)	1.817s (71.6M)	32.947s (616M)
ball tree, prioritized	0.003s (104k)	0.023s (666k)	0.285s (10.9M)	27.934s (514M)
ball tree, improved	0.003s (86.8k)	0.017s (455k)	**0.160s** (5.65M)	**2.332s** (351M)
single ball tree	**0.002s** (69.6k)	**0.012s** (315k)	0.165s (**5.38M**)	26.357s (**254M**)

algorithm	**covertype**	**power**	**lcdm**	**sdss-dr6**
kd-tree, unordered	302.8s (1.09B)	1163.0s (18.7B)	5628.7s (41.5B)	24717s (156B)
kd-tree, prioritized	15.823s (52.5M)	30.072s (302M)	319.871s (1.87B)	9069s (50.3B)
kd-tree, improved	**4.469s (12.8M)**	**12.714s (200M)**	**71.587s (350M)**	**428.9s (2.14B)**
single *kd*-tree	6.207s (16.3M)	19.684s (232M)	120.6s (476M)	471.4s (2.24B)
ball tree, unordered	163.027s (2.90B)	771.975s (25.3B)	1861.9s (71.1B)	9444s (363B)
ball tree, prioritized	52.487s (902M)	113.437s (3.90B)	386.74s (14.4B)	5202s (192B)
ball tree, improved	**27.251s** (392M)	**83.744s** (2.58B)	**195.175s** (6.46B)	**5150s** (136B)
single ball tree	29.948s (**228M**)	138.422s (**2.49B**)	402.6s (**5.93B**)	7226s (**101B**)

Table 2. Runtime (distance evaluations) for exact nearest neighbor search.

algorithm	cloud	winequality	birch3	miniboone
kd-tree, unordered	0.005s (34.5k) [1.5]	0.025s (148k) [1.44]	0.267s (2.14M) [1.44]	6.831s (22.6M) [1.38]
kd-tree, prioritized	0.003s (17.4k) [1.5]	0.012s (74.5k) [1.5]	0.140s (1.16M) [1.5]	4.863s (15.5M) [1.38]
kd-tree, improved	**0.002s (13.7k)** [1.7]	**0.010s (51.2k)** [1.63]	**0.107s (654k)** [1.63]	3.360s (9.28M) [1.38]
single *kd*-tree	0.003s (23.2k) [2.45]	0.013s (78.0k) [2.33]	0.198s (1.47M) [2.33]	**1.845s (5.75M)** [1.5]
ball tree, unordered	**0.002s** (50.8k) [27.6]	0.007s (186k) [32.3]	0.079s (2.72M) [11.5]	**2.942s** (50.4M) [285]
ball tree, prioritized	**0.002s** (49.2k) [27.6]	**0.006s** (167k) [32.3]	**0.072s** (2.46M) [11.5]	3.266s (54.2M) [249]
ball tree, improved	**0.002s** (45.1k) [27.6]	**0.006s (161k)** [32.3]	**0.072s (2.25M)** [11.5]	3.494s (50.3M) [99]
single ball tree	**0.002s** (43.2k) [999]	**0.006s** (176k) [36.0]	0.111s (3.56M) [10.1]	3.812s (**36.1M**) [99]
multiprobe LSH	0.031s (19.3k) [20/122]	0.011s (472k) [37/33]	1.614s (8.85M) [8/16k]	175.995s (1.77B) [13/328]

algorithm	covertype	power	lcdm	sdss-dr6
kd-tree, unordered	7.796s (27.4M) [1.5]	419.725s (13.0B) [1.27]	75.432s (508M) [1.33]	512.829s (2.89B) [1.27]
kd-tree, prioritized	2.954s (10.6M) [1.5]	**8.392s (189M)** [1.44]	44.187s (306M) [1.38]	380.047s (2.17B) [1.27]
kd-tree, improved	**2.045s (6.25M)** [1.5]	11.044s (191M) [1.56]	**29.069s (160M)** [1.44]	**242.624s (1.11B)** [1.27]
single *kd*-tree	3.869s (11.2M) [1.86]	16.674s (226M) [2.33]	85.821s (397M) [1.78]	329.663s (1.58B) [1.27]
ball tree, unordered	2.187s (33.0M) [99]	415.964s (13.0B) [11.5]	**19.776s** (668M) [19]	**73.638s** (239M) [49]
ball tree, prioritized	**2.183s (32.3M)** [75.9]	**6.753s (233M)** [13.3]	20.158s (**660M**) [19]	75.687s (**237M**) [49]
ball tree, improved	2.539s (33.8M) [49]	8.269s (248M) [15.7]	25.749s (702M) [21.2]	299.8s (451M) [49]
single ball tree	5.496s (40.3M) [27.6]	19.097s (431M) [15.7]	113.299s (1.46B) [21.2]	2054.8s (3.06B) [19]
multiprobe LSH	130.699s (963M) [0.51]	1181.32s (14.0B) [63/9.6]	*timeout* [14/0.968]	*timeout* [7/0.29]

Table 3. Runtime (distance calculations) [ϵ or M/W] for approximate NN search.

easy to modify the given `Score()` function to enforce this condition; replace the equation in line 3 with $d_{\min}(\mathscr{N}_q, \mathscr{N}_r) > (1/(1+\epsilon))B(\mathscr{N}_q)$.

After applying this change, testing is performed in the same way as for exact nearest neighbor search. ϵ for each tree-based approach is selected to give an average per-point relative error of 0.1 (± 0.01) for each dataset. Because our scheme does not allow the error for an individual point to exceed ϵ, the actual relative error for an individual query point is often much lower. Thus, it is often necessary to set ϵ far higher than the target average error of 0.1. For LSH, the LSHKIT package is used, which implements multi-probe LSH and autotunes the hashing parameters [30]. We use the suggested number of hash tables ($L = 10$) and probes ($T = 20$), and then autotune to select the number of hash functions (M) and bin width (W). Autotuning failed for the larger power, lcdm, and sdss-dr6 datasets; in these cases suggestions of the LSHKIT authors are used [31].

The results are given in Table 3. With approximation, the improved dual-tree traversal performs fewer distance calculations on smaller datasets, and is still dominant for the larger datasets with *kd*-trees. But with ball trees, the bounds are looser and thus nodes are more likely to be overlapping. Because only an approximate nearest neighbor is required, finding the absolute best reference child to recurse into is of less importance, and the added overhead of delaying query recursions may not necessarily be helpful. Thus, the benefit of the improved traversal may be related to the type of tree being used and the problem being solved. LSH is not competitive on the larger datasets, and on the largest datasets LSH did not complete within 3 days, but it should be noted that the low-dimensional setting is where trees are most effective.

Overall, for large datasets in low-to-medium dimensions, dual-tree search is faster, and the improved traversal we have proposed is the fastest. These experiments, as well as further investigations (not shown here due to space constraints) seem to show for smaller datasets, single-tree search may be fastest; for sufficiently high dimensions, LSH is faster. This corroborates existing results [32]; as the dimension of data gets higher, pruning rules become less effective. Regardless, in low-to-medium dimensions, the improved dual-tree traversal is dominant.

7 Conclusion

Using the recent abstraction of tree-independent dual-tree algorithms, we have proposed a novel depth-first dual-tree traversal which compares favorably against other techniques for exact and approximate nearest neighbor search. Additionally, because of the generic nature of the traversal, it may be applied to many problems: the traversal simply needs to be paired with a type of space tree and `Score()` and `BaseCase()` functions. Examples of problems with existing `Score()` and `BaseCase()` functions include kernel density estimation [12] and max-kernel search [23]. These problems, as well as nearest neighbor search, all stand to benefit from the improved traversal strategy we have proposed.

Acknowledgements. Thanks to Rich W. Vuduc and Chad D. Kersey for helpful discussions and comments during the preparation of this work. This material is based on work supported by the U.S. National Science Foundation (NSF) Award Number 1339745. Any opinions, findings and conclusions or recommendations expressed in this metrial are those of the author and do not necessarily reflect those of NSF.

References

1. N. Kumar, L. Zhang, and S.K. Nayar. What is a good nearest neighbors algorithm for finding similar patches in images? In *The 10th European Conference on Computer Vision (ECCV)*. October 2008.
2. Y. Koren. The BellKor solution to the Netflix Grand Prize. 2009.
3. P. Cunningham and S.J. Delany. k-nearest neighbour classifiers. Technical Report UCD-CSI-2007-4, University College Dublin, 2007.
4. K.Q. Weinberger, J. Blitzer, and L.K. Saul. Distance metric learning for large margin nearest neighbor classification. In *Advances in Neural Information Processing Systems 18 (NIPS*05)*, pages 1473–1480, 2005.
5. J.L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.
6. K. Fukunaga and P.M. Narendra. A branch and bound algorithm for computing k-nearest neighbors. *IEEE Transactions on Computers*, 100(7):750–753, 1975.
7. K.L. Clarkson. Nearest neighbor queries in metric spaces. *Discrete & Computational Geometry*, 22(1):63–93, 1999.
8. A. Andoni and P. Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. In *Forty-Seventh Annual IEEE Symposium of Foundations of Computer Science (FOCS '06)*, pages 459–468. IEEE, 2006.
9. P. Indyk and R. Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing (STOC '98)*, pages 604–613. ACM, 1998.
10. M. Datar, N. Immorlica, P. Indyk, and V.S. Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *Proceedings of the Twentieth Annual Symposium on Computational Geometry (SoCG '04)*, pages 253–262. ACM, 2004.
11. A.G. Gray and A.W. Moore. 'N-Body' problems in statistical learning. In *Advances in Neural Information Processing Systems 14*, volume 4, pages 521–527, 2001.
12. R.R. Curtin, W.B. March, P. Ram, D.V. Anderson, A.G. Gray, and C.L. Isbell Jr. Tree-independent dual-tree algorithms. In *Proceedings of the 30th International Conference on Machine Learning (ICML '13)*, 2013.

13. P. Ram, D. Lee, W.B. March, and A.G. Gray. Linear-time algorithms for pairwise statistical problems. *Advances in Neural Information Processing Systems*, 22, 2009.
14. R.R. Curtin, D. Lee, W.B. March, and P. Ram. Plug-and-play runtime analysis for dual-tree algorithms. *The Journal of Machine Learning Research*, 2015.
15. A.G. Gray and A.W. Moore. Nonparametric density estimation: Toward computational tractability. In *Proceedings of the 3rd SIAM International Conference on Data Mining (SDM '03)*, pages 203–211, San Francisco, 2003.
16. W.B. March, P. Ram, and A.G. Gray. Fast Euclidean minimum spanning tree: algorithm, analysis, and applications. In *Proceedings of the 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '10)*, pages 603–612, Washington, D.C., 2010.
17. P. Wang, D. Lee, A.G. Gray, and J.M. Rehg. Fast mean shift with accurate and stable convergence. In *Proceedings of the Eleventh International Conference on Artificial Intelligence and Statistics (AISTATS 2007)*, pages 604–611, 2007.
18. D. Lee and A.G. Gray. Faster Gaussian summation: Theory and experiment. In *Proceedings of the 22nd Conference on Uncertainty in Artificial Intelligence*, 2006.
19. R.R. Curtin, P. Ram, and A.G. Gray. Fast exact max-kernel search. In *SIAM International Conference on Data Mining (SDM '13)*, pages 1–9, 2013.
20. M. Klaas, M. Briers, N. De Freitas, A. Doucet, S. Maskell, and D. Lang. Fast particle smoothing: if I had a million particles. In *Proceedings of the 23rd International Conference on Machine Learning (ICML '06)*, pages 25–29, 2006.
21. L. Van Der Maaten. Accelerating t-sne using tree-based algorithms. *The Journal of Machine Learning Research*, 15(1):3221–3245, 2014.
22. D.A. Moore and S.J. Russell. Fast Gaussian process posteriors with product trees. In *Proceedings of the Thirtieth Conference on Uncertainty in Artificial Intelligence (UAI-14)*, Quebec City, July 2014.
23. R.R. Curtin and P. Ram. Dual-tree fast exact max-kernel search. *Statistical Analysis and Data Mining*, 7(4):229–253, 2014.
24. T. Liu, A.W. Moore, K. Yang, and A.G. Gray. An investigation of practical approximate nearest neighbor algorithms. In *Advances in Neural Information Processing Systems 17 (NIPS 2004)*, pages 825–832, 2004.
25. R.R. Curtin, J.R. Cline, N.P. Slagle, W.B. March, P. Ram, N.A. Mehta, and A.G. Gray. mlpack: A scalable C++ machine learning library. *Journal of Machine Learning Research*, 14:801–805, 2013.
26. M. Lichman. UCI machine learning repository, 2013.
27. T. Zhang, R. Ramakrishnan, and M. Livny. BIRCH: A new data clustering algorithm and its applications. *Data Mining and Knowledge Discovery*, 1(2):141–182, 1997.
28. R. Lupton, J.E. Gunn, Z. Ivezic, G.R. Knapp, and S. Kent. The SDSS imaging pipelines. In *Astronomical Data Analysis Software and Systems X*, volume 238, page 269, 2001.
29. J.K. Adelman-McCarthy, M.A. Agüeros, S.S. Allam, C.A. Prieto, K.S.J. Anderson, et al. The sixth data release of the Sloan Digital Sky Survey. *The Astrophysical Journal Supplement Series*, 175(2):297, 2008.
30. W. Dong, Z. Wang, W.K. Josephson, M. Charikar, and K. Li. Modeling LSH for performance tuning. In *Proceedings of the 17th ACM Conference on Information and Knowledge Management (CIKM 2008)*, pages 669–678. ACM, 2008.
31. W. Dong. Personal communication, 2015.
32. A.W. Moore. The Anchors hierarchy: Using the triangle inequality to survive high dimensional data. In *Proceedings of the Sixteenth Conference on Uncertainty in Artificial Intelligence (UAI 2000)*, pages 397–405, 2000.