Proximity Search in the Greedy Tree

Oliver Chubet* Parth Parikh† Donald R. Sheehy‡ Siddharth Sheth§

November 10, 2022

Abstract

Over the last 50 years, there have been many data structures proposed to perform proximity search problems on metric data. Perhaps the simplest of these is the ball tree, which was independently discovered multiple times over the years. However, there is a lack of strong theoretical guarantees for standard ball trees, often leading to more complicated structures when guarantees are required. In this paper, we present the *greedy tree*, a simple ball tree construction for which we can prove strong theoretical guarantees for proximity search queries, matching the state of the art under reasonable assumptions. To our knowledge, this is the first ball tree construction providing such guarantees. Like a standard ball tree, it is a binary tree with the points stored in the leaves. Only a point, a radius, and an integer are stored for each node. The asymptotic running times of search algorithms in the greedy tree match those of more complicated structures regularly used in practice.

1 Introduction.

The ball tree [17] is perhaps the simplest hierarchical data structure for proximity search queries in metric spaces, such as approximate nearest neighbor or range search. Ball trees are defined by recursively partitioning the data set and representing the parts with a binary tree. Each node of the tree stores a metric ball (a center and a radius) that covers the points in its subtree. A search proceeds by a branch and bound process where nodes can be pruned out of the search as soon as the corresponding ball falls entirely outside the search range. Approximate queries can be supported by halting the search when all feasible balls are less than some determined radius or simply being more agressive in pruning the search. This simple pattern works for many different types of proximity search problems.

Unfortunately, strong theoretical guarantees have not been proven for previously studied ball tree constructions. In this work, we present the *greedy tree*: a linear-size structure that supports many approximate and exact proximity search queries with worst-case running times comparable to those of more complicated data structures. For example, if the spread of the input (the ratio of the largest to smallest pairwise distances) is Δ and the doubling dimension (see Section 3) is d, then approximate nearest neighbors can be computed in $2^{O(d)} \log \Delta$ time and approximate range searches can be computed in $2^{O(d)} \log \Delta + O(k)$ time, where k is the number of points returned by the search.

Various ball tree constructions differ primarily in their method of partitioning the points. Early methods emphasized balanced splits with the goal of minimizing the height of the search tree. Thus, it was common to search for a median in some direction [17, 23] in analogy with kd-trees [1]. However, because most proximity searches require searching multiple paths down the tree, the height is not the only factor determining the running time. It is at least as important to bound the number of such paths, or, roughly speaking, the *width* of the search. In the greedy tree, we construct the tree using a farthest point sampling, also known as a greedy permutation, which orders the points so that the i^{th} point has the farthest distance from the first $i-1$ points. For each point in this ordering, an approximate nearest predecessor is used to determine its parent in the tree. The details of the construction are presented in Section 5. The key to bounding the running time is the packing property of greedy permutations that allows us to bound the width of the search.

*North Carolina State University, United States, oachubet@ncsu.edu
†North Carolina State University, United States, pmparikh@ncsu.edu
‡North Carolina State University, United States, don.r.sheehy@gmail.com
§North Carolina State University, United States, ssheth4@ncsu.edu

1.1 Contributions and Summary. In Section 4, we present a basic pattern for proximity search in ball trees as exemplified by range search and approximate nearest neighbor search. We then show how algorithms implementing this pattern can be analyzed in terms of formally defined notions of the width and height of the search. In Section 5, we give the detailed construction of the greedy tree and show how it guarantees constant width searches in doubling metrics, leading to the desired running time bounds in Section 6. As an example of a more challenging query that can be implemented on a greedy tree, we discuss k nearest neighbor queries in Section 7. To our knowledge, this is the first algorithm for general metric spaces that can approximate the distance to the kth nearest neighbor in time independent of k. In Section 8, we discuss an open source implementation as well as some applications and directions for future work.

2 Related Work.

Despite a plethora of proximity search data structures, most designs reflect a tradeoff between structural simplicity and query time complexity. We focus on hierarchical structures that are constructed by recursively decomposing a set into two or more disjoint subsets. Most such structures are generalizations of structures designed for Euclidean space such as quadtrees [9] and kd-trees [1]. These structures include ball trees [17], metric trees [23], vp-trees [24], and M-trees [3]. These data structures are simple to define and query, but few of them have strong theoretical bounds on running times. For example, although the vp-trees were shown to support expected $O(\log n)$-time queries over certain input distributions, no worst-case bounds are known.

Other hierarchical structures have been proposed that admit strong guarantees on the running time of queries. These guarantees are generally expressed in terms of the "dimension" of the input metric (see Clarkson [6] for a survey of different notions of dimension). This approach goes back to the metric skip lists of Karger and Ruhl [14] in so-called growth restricted metrics.

Krauthgamer and Lee's navigating nets [15] compute $(1+\varepsilon)$-approximate nearest neighbors in $2^{O(d)} \log \Delta + \left(\frac{1}{\varepsilon}\right)^{O(d)}$ time for metrics with doubling dimension d. A navigating net is a directed acyclic graph, rather than a tree, as is the *sb* structure of Clarkson [4, 5]. Both of these require space that depends exponentially on the dimension.

Beygelzimer et al. [2] presented the cover tree, which may be viewed as a spanning tree of a navigating net. This was the first data structure in this line of work that had truly linear size independent of the dimension. Har-Peled and Mendel [12] presented the net tree, which had slightly stronger guarantees at the cost of significantly larger space and complexity.

One difficulty of working with net-trees or cover trees is that they use a compression scheme to guarantee linear size. Although not terribly complicated, this compression adds some significant complication to a correct search algorithm [2]. It also led to significant errors in the analysis [8]. In addition to an increased structural complexity, the compression scheme uses radii which are rounded to the next power of 2 (or 5 for net-trees) which reduces the amount of pruning possible within a proximity search. These challenges were addressed in the implementation by Izbicki [13], who introduced a variety of clever heuristics and improvements. However, the greedy tree eliminates the need for this extra work altogether.

As its name suggests, the construction of the greedy tree depends on a greedy permutation, which is determined by a furthest point sampling of the underlying set. Greedy permutations were introduced by Rosenkrantz et al. [18] to approximate the traveling salesman problem. Gonzalez [10], and Dyer and Frieze [7] used them for k-center clustering. Relaxing the greedy permutation, resulting in a sequence of points that is only approximately greedy, is a variation of an idea used in the net-tree construction [12]. This is a critical idea that allows the greedy tree to achieve its theoretical guarantees.

3 Background.

3.1 Doubling Metrics. Let $(X, \mathbf{d})$ be a metric space with distance $\mathbf{d}$. We overload notation and write $\mathbf{d}(x, S)$ to be the minimum distance from x to any point in a set S.

A *metric ball* of radius r centered at $x \in X$ is $\mathsf{B}(x, r) := \{p \in X \mid \mathbf{d}(x, p) \leq r\}$. A collection of sets, $\mathcal{C}$, *covers* a set A if $A \subseteq \bigcup_{S \in \mathcal{C}} S$. The *doubling constant* of a metric space X is the minimum number ρ such that any ball of radius r can be covered by at most ρ balls of radius $\frac{r}{2}$. The *doubling dimension* of X is $\dim(X) := \log_2 \rho$. A metric space with bounded doubling dimension is called a *doubling metric*. A set A is *r-packed* if $d(a, b) \geq r$ for any distinct $a, b \in A$.

The following lemma is a useful fact which follows from a classical packing argument [11].

LEMMA 3.1. *If A is an r-packed set with diameter D in metric space X, then $|A| \leq (\frac{2D}{r})^d$, where d is the doubling dimension of X.*

The *spread* of a set A, denoted Δ, is the ratio of the largest to the smallest pairwise distances of points in A.

3.2 Ball Trees. Let A be a set in a metric space X. A *ball tree* on A is a binary tree defined by recursively partitioning A. Each node of the tree stores a center and a radius that covers the points in its leaves. A ball tree *node* of radius r centered at $x \in A$ is denoted (x, r).

Let (x, r) be a node in a ball tree constructed on set A. If $a \in A$ is contained in a leaf of (x, r), then a is a point of (x, r). The set of all points in (x, r) is denoted by $\text{Pts}(x, r)$. Two nodes (y, r_y) and (z, r_z) are *independent* if $\text{Pts}(y, r_y) \cap \text{Pts}(z, r_z) = \varnothing$.

3.3 Greedy Permutations. Let $P = (p_0, \ldots, p_{n-1})$ be a finite sequence of points in a metric space with distance $\mathbf{d}$. The i^{th}-*prefix* is the set $P_i = \{p_0, \ldots, p_{i-1}\}$ containing the first i points of P.

The sequence P is a *greedy permutation* if for all i,

$$\mathbf{d}(p_i, P_i) = \max_{p \in P} \mathbf{d}(p, P_i).$$

If a sequence P is such that,

$$\mathbf{d}(p_i, P_i) \leq \alpha \max_{p \in P} \mathbf{d}(p, P_i),$$

for all $i > 0$ where $\alpha > 1$ then P is an α-*approximate greedy permutation*. The point $p_0 \in P$ is the *seed* of the greedy permutation. The *predecessor mapping* $T : P \setminus \{p_0\} \to P$ maps each point p_i in P (other than the seed) to the (approximately) closest point in the prefix P_i. The *insertion distance* of p is $\mathbf{d}(p, T(p))$, denoted ε_p. For α-approximate greedy permutations, we require that the predecessor mapping satisfies the condition that

$$\varepsilon_p \leq \frac{1}{\alpha} \varepsilon_{T(p)}.$$

There is a straightforward algorithm to compute such approximate greedy permutations and the corresponding predecessor mapping [20, 19]. A greedy permutation can be computed in $O(n \log \Delta)$ time for low-dimensional data [12, 19]. An $O(n \log n)$-time randomized approximation algorithm exists [12], but, to our knowledge, it has never been implemented.

4 Proximity Search in a Ball Tree.

In this section, we present algorithms for common proximity searches using ball trees. In particular, we focus on range search and nearest neighbor search. The basic pattern presented for these algorithms is easily transferred to other proximity search problems such as approximate range search, range counting, or farthest point queries.

General ball trees may have leaf nodes with non-zero radii. A search through such ball trees can only be accurate up to the largest leaf radius, unless one iterates through the points in leaves. For simplicity, we assume that leaf radii are zero.

Proximity search queries in a ball tree G follow a common branch and bound pattern. If q is a query point then the *search radius* R determines the query region $\mathsf{B}(q, R)$. The search maintains a *viable set* of nodes of G that may intersects the query region. Viability is checked using the triangle inequality by computing the distance from the query region to the center of the node and subtracting the radius. The viable set H is stored as a max-heap ordered by radius.

4.1 Range Search. In a range search, the query is a center q and a radius R. The output should contain all points within distance R of q, i.e., the subset of $\text{Pts}(G)$ within $\mathsf{B}(q, R)$. The algorithm RANGE describes a range search on a ball tree.

Algorithm $\text{RANGE}(q, G, R)$:

1. Initialize H with the root node of G and `output` as $\varnothing$.

2. While H is not empty:
   (a) Remove the node with maximum radius (x, r) from H.
   (b) If $\mathbf{d}(q, x) \leq R - r$, then add $\text{Pts}(x, r)$ to output.
   (c) Else, if (x, r) is not a leaf, then:
      i. Split (x, r) into its children (y, r_y) and (z, r_z).
      ii. If $\mathbf{d}(q, y) \leq R + r_y$, add (y, r_y) to H.
      iii. If $\mathbf{d}(q, z) \leq R + r_z$, add (z, r_z) to H.
3. Return output.

Initially H contains the root of G, so $\text{Pts}(H) = \text{Pts}(G)$. However, H is empty at the end of RANGE. So, for every point $p \in \text{Pts}(G)$, there corresponds a node which is the *last node* containing p in the search. In this last iteration, the node containing p is either added to output, or pruned from the search. Figure 1 shows an iteration of the main loop of RANGE. Figure 2 shows how the search space of RANGE is pruned over time. The following theorem proves correctness of RANGE.

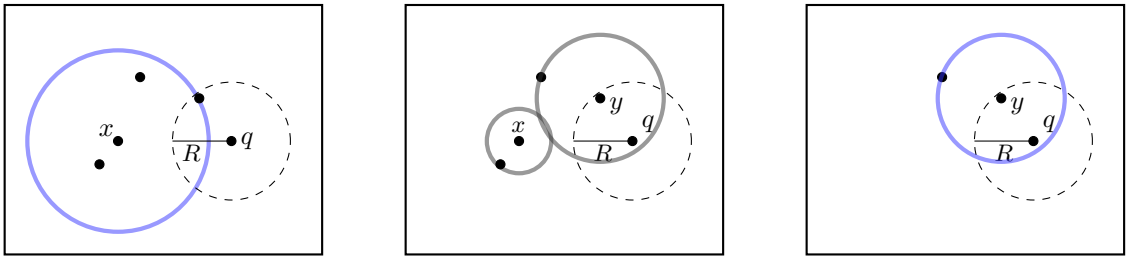


Figure 1: Consider this iteration of RANGE when (x, r_x) is the node with maximum radius in H. So, (x, r_x) is split into its children (y, r_y) and (x, r'_x). As (y, r_y) is viable, it is added to H.

THEOREM 4.1. *If* $\text{RANGE}(q, G, R)$ *returns* output *then* $\text{Pts}(G) \cap \mathsf{B}(q, R) = \mathsf{output}$.

Proof. For some $p \in G$ such that $p \notin \mathsf{output}$, let the last node containing p be (a, r_a). It follows that $(a, r_a) \notin H$. Thus, $\mathbf{d}(q, a) > R + r_a$ and so by the triangle inequality $\mathbf{d}(q, p) > R$. So, if $p \notin \mathsf{output}$ then $p \notin \text{Pts}(G) \cap \mathsf{B}(q, R)$. Thus, $\text{Pts}(G) \cap \mathsf{B}(q, R) \subseteq \mathsf{output}$. By the viability check in Step 2b it is clear that $\mathsf{output} \subseteq \text{Pts}(G) \cap \mathsf{B}(q, R)$. Therefore, $\text{Pts}(G) \cap \mathsf{B}(q, R) = \mathsf{output}$. ∎

4.2 (Approximate) Nearest Neighbor Search. A c-approximate nearest neighbor query returns a point of $\text{Pts}(G)$ whose distance to the query q is at most c times the distance to the nearest point of $\text{Pts}(G)$ of q. Here, R is the distance from q to the current nearest point, and R decreases as points closer to q are discovered. ANN illustrates an (approximate) nearest neighbor search in a ball tree.

Algorithm $\text{ANN}(q, G, c)$:

1. Initialize H with root node (x_0, r_0) of G.
2. Set $\mathsf{nearest} := x_0$ and $R := \mathbf{d}(q, \mathsf{nearest})$.
3. While H is not empty:
   (a) Remove node with maximum radius (x, r) from H.
   (b) If (x, r) is not a leaf:
      i. Split (x, r) into its children (y, r_y) and (z, r_z).
      ii. If $\mathbf{d}(q, y) \leq R$ then $\mathsf{nearest} := y$ and $R := \mathbf{d}(q, y)$.
      iii. If $\mathbf{d}(q, z) \leq R$ then $\mathsf{nearest} := z$ and $R := \mathbf{d}(q, z)$.
      iv. If $\mathbf{d}(q, y) - r_y \leq R/c$, then add (y, r_y) to H.

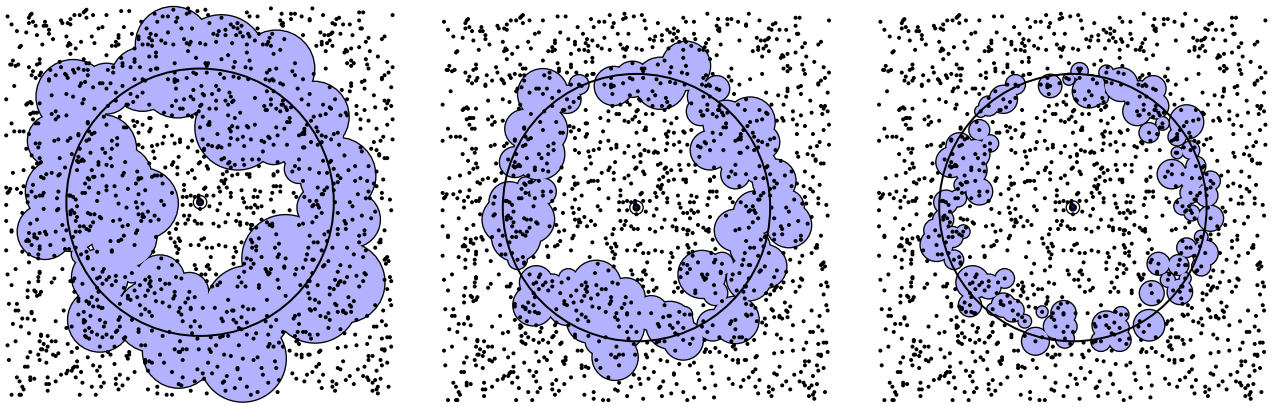


Figure 2: As the RANGE algorithm proceeds, the nodes in the viable heap H get smaller. The figure shows three different moments in the execution. The nodes completely contained by the query region that have already been added to the output are no longer stored in H. There are nodes completely contained within the query region that have not been processed yet because their radii are small.

v. if $\mathbf{d}(q, z) - r_z \leq R/c$, then add (z, r_z) to H.

4. Return nearest.

A $(1+\varepsilon)$-approximate nearest neighbor is returned when $c = 1+\varepsilon$. Figure 3 illustrates the pruning condition of ANN. The proof of correctness of ANN closely resembles that of RANGE. The viable heap contains every point in G initially, but is empty at the end. So corresponding to each point there is a node that was the last node to contain it.

THEOREM 4.2. *If the output of* ANN(q, G, c) *is* p *then* $\mathbf{d}(q, p) \leq c\mathbf{d}(q, p')$ *where* p' *is the nearest neighbor of* q.

Proof. Let the last node containing p' be (a, r_a). Suppose $\mathbf{d}(q, p) > c\mathbf{d}(q, p')$. Then by the triangle inequality, $\mathbf{d}(q, a) \leq \mathbf{d}(q, p') + \mathbf{d}(p', a) \leq R/c + r_a$. Therefore, we have a contradiction because (a, r_a) should have been added to H and so it could not have been the last node to contain p'. ∎

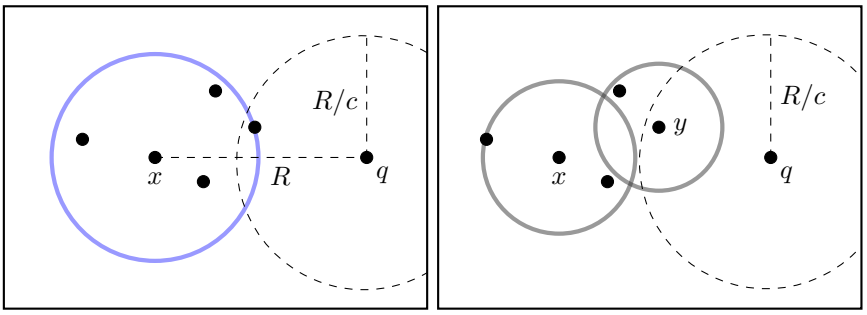


Figure 3: Here we have an iteration of ANN when (x, r_x) is the node with maximum radius in H. The node is split into its children (x, r'_x) and (y, r_y). It can be seen that y will become the new nearest neighbor and (x, r'_x) is no longer viable.

4.3 Analyzing a Search. We can analyze the running time of both range search and nearest neighbor search in a ball tree using a similar analysis. We assume that distance computations require only constant time. In both algorithms the running time is dominated by the main loop, which does a constant number of distance computations and a constant number of heap operations per iteration.

The *width* w of a search is the maximum number of viable nodes at any time. The *height* h of a search is the maximum depth of any node touched in the search.

LEMMA 4.1. *A* RANGE *query in a greedy tree takes* $O(wh \log w + k)$ *time, where* k *is the output size.*

Proof. Let H be the viable heap for the search. There are $O(wh)$ iterations of the main loop, and $|H|$ is $O(w)$. There are a constant number of heap operations per iteration, so the total running time is $O(wh \log w)$ plus the time to add points to the output, which is $O(k)$. ∎

It is hard to get a better bound on the width and height of a search without additional assumptions. However, when one guarantees a constant width, the running time becomes $O(h + k)$.

For a RANGE query, the nodes in the viable heap remain viable. However for an ANN query, as the search radius changes, some nodes in the viable heap, although viable upon insertion, may become too far from the query to be viable at this new scale. A node with radius r has *level* k if $\lceil \log r \rceil = k$. Thus, to ensure heap performance, one may replace the viable heap with a bucket queue [22], where a bucket $lv(k)$ stores the nodes of level k. The the heap operations become constant time. Although now when we pop the last node in a bucket we must traverse the buckets until we find the next non-empty bucket. By construction of a ball tree, the radius of a child is never greater than the radius of its parent. So, it follows that the radius of an inserted node is never greater than the maximum of the current bucket. There are at most $\log \Delta$ buckets, so we pay an additive $\log \Delta$ term to account for the bucket queue traversal. We conclude the following lemma.

LEMMA 4.2. *An* ANN *query in a greedy tree (using a bucket queue) runs in* $O(wh + \log \Delta)$ *time.*

5 Greedy Trees.

Our strategy is to produce a ball tree with packing guarantees that allow us to optimize the height and width of a proximity search. In this section we introduce the greedy tree which is a ball tree that uses farthest point sampling to achieve packing guarantees.

5.1 Constructing a Ball Tree from a Permutation. As the name implies, we construct the *greedy tree* from a greedy permutation. The construction applies more generally to any permutation on a point set P with a predecessor mapping T. Given P and T, the *descendants* of a point p are defined recursively so that q is a descendant of p if $q = p$ or $T(q)$ is a descendant of p.

To build a ball tree, it suffices to describe how to partition a set and pick the centers of the two parts. For a set $S \subseteq P$, let a and b be the first and second points of S with respect to their ordering in P. These will be the centers of the two sets. The set centered at b will contain the descendants of b and the set centered at a will contain the rest of the points. The convention is that the left child is centered at a and the right child is centered at b.

The definition in terms of partitions is simple to state, but there is an equivalent definition that is simpler to implement. One can construct the ball tree G for P and T incrementally as follows. Start with a root node centered at p_0. Iterate through the points b in the permutation (starting at p_1). Let $a = T(b)$ and let x be the unique leaf of G centered at a. Create new nodes centered at a and b and assign them to be the left and right children of x. See Figure 4.

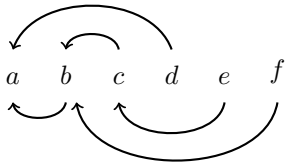
Once the tree is built, the radius of each node can be computed as the distance to its farthest descendant. The number of leaves in the subtree rooted at any node is also stored in the node. This can be used for range counting and k nearest neighbor search.

We say G has a parameter α when P is an α-approximate greedy permutation. As noted previously, for all nodes p_i (other than the root), we construct the greedy permutation so that $\varepsilon_{p_i} \leq \alpha \mathbf{d}(p_i, P_i)$ and $\varepsilon_{T(p_i)} \geq \alpha \varepsilon_{p_i}$.

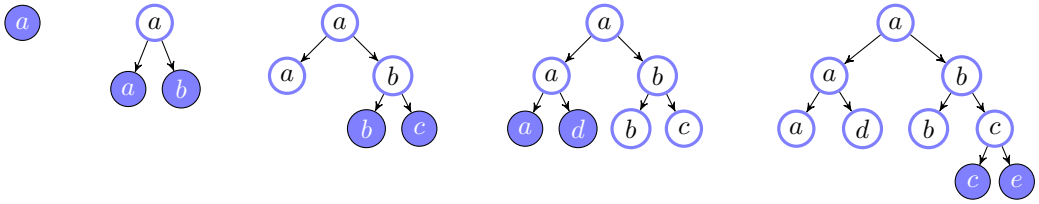
5.2 Structure Theorem. In this section, we prove several structural results for approximate greedy trees. These results are used in Section 6 to bound the running time of proximity searches.

THEOREM 5.1. *Let* G *be a greedy tree with parameter* $\alpha > 1$. *Then the following properties hold:*

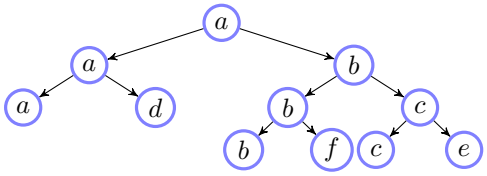
1. *The radius of a node* (p, r_p) *is bounded,* $r_p \leq \frac{\varepsilon_p}{\alpha - 1}$.
2. *Let* X *be a set of pairwise independent nodes from* G. *Then the centers of* X *are* $\frac{(\alpha-1)r}{\alpha}$*-packed, where* r *is the minimum radius of any parent of a node in* X.



(a) The permutation (a, b, c, d, e, f) is depicted with arrows representing a predecessor mapping.



(b) The tree is constructed incrementally. Each new point creates two new nodes.



(c) The completed greedy tree.

Figure 4: A ball tree is computed for a given permutation and predecessor pairing.

3. The height h of G is $2^{O(d)} \log \Delta$.

Proof. 1. Let (p, r_p) be a node in G and $q \in \text{Pts}(p, r_p)$ be such that $r_p = \mathbf{d}(p, q)$. Additionally, let (x, r_x) and (y, r_y) be nodes containing q such that $T(x) = p$ and $T(y) = x$. By the triangle inequality,

$$r_p \leq \mathbf{d}(p, x) + \mathbf{d}(x, q) \leq \varepsilon_x + r_x.$$

Moreover, by construction, $\varepsilon_x \leq \frac{1}{\alpha}\varepsilon_p$, so $r_p \leq \frac{1}{\alpha}\varepsilon_p + r_x$, so by induction on the height,

$$r_p \leq \sum_{i=1}^{h} \varepsilon_p \left(\frac{1}{\alpha}\right)^i \leq \sum_{i=0}^{\infty} \varepsilon_p \left(\frac{1}{\alpha}\right)^{i+1} \leq \frac{\varepsilon_p}{\alpha - 1}.$$

2. Let X be a set of pairwise independent nodes from G, and let (x, r_x) be an arbitrary node in X with a parent (p, r_p). Then, $r_p \leq \varepsilon_x + r_x \leq \varepsilon_x + \frac{\varepsilon_x}{\alpha - 1}$. It follows that $\varepsilon_x \geq \frac{\alpha - 1}{\alpha} r$ and therefore, H is $\frac{(\alpha - 1)r}{\alpha}$-packed, where r is the minimum radius of any parent of a node in X.

3. Let n_h be the leaf of G of maximum depth. Let $(n_0, n_1, \ldots, n_h)$ be the path from the root to n_h. For all indices j, let $n_j = (q_j, r_j)$ and $\varepsilon_j = \mathbf{d}(T(q_j), q_j)$. If $q_j \neq q_{j+1}$, then $q_j = T(q_{j+1})$, and so, by construction, $\varepsilon_{j+1} \leq \frac{1}{\alpha}\varepsilon_j$. This implies there are at most $1/(\log \alpha)$ distinct points q_j per level.

Let X be the set containing the right child of each node of level k that is centered at a point q. Then X is a pairwise independent set, and therefore is $\left(\frac{\alpha - 1}{\alpha}\right) 2^{k-1}$-packed. Furthermore, the center of each node in X is contained within the ball $\mathsf{B}(q, 2^k)$. By the Standard Packing Lemma 3.1, it follows that there can be at most $2^{O(d)}$ nodes in X. So, there are at most $2^{O(d)}$ nodes per level on the path from the root to n_h. There are at most $\log \Delta$ levels, so it follows that the length of the path h is at most $2^{O(d)} \log \Delta$.

∎

6 Proximity Search Analysis.

In this section, we give a framework for analyzing proximity search algorithms on a greedy tree. For many problems, exact algorithms will require near linear time at least in the worst case. However, for approximate queries, such as approximate nearest neighbors, existing data structures can easily achieve $O(\log \Delta)$ running times in doubling metrics. Previously, it was not known how to achieve such bounds for ball trees. In this section, we show how to bound the running time of generic search in a greedy tree.

We build on the ball tree analysis from Section 4 using properties of greedy trees. We can use Theorem 5.1 to bound the width and height of ANN in the greedy tree, which ultimately bounds the running time of the search. One may note that many of the analyses for the running times of proximity searches in the greedy tree follow a similar pattern to that of ANN.

LEMMA 6.1. *The width of a (1+ε)-*ANN *search is* $(1 + \frac{1}{\varepsilon})^{O(d)}$.

Proof. It is sufficient to show that while executing ANN, there are $(1+\frac{1}{\varepsilon})^{O(d)}$ viable nodes in any bucket $lv(k+1)$. Note that there are at most $1/(\log \alpha)$ sets of pairwise independent nodes. Let X be one such set. We know that the minimum radius of any parent of X is $\alpha 2^k$. So by Theorem 5.1, the centers of X are $(\alpha - 1)2^k$-packed. Consider a node $(y, r_y) \in X$. The algorithm requires that $\mathbf{d}(y, q) - r_y \leq \frac{R_y}{1+\varepsilon} \leq \frac{\mathbf{d}(y,q)}{1+\varepsilon}$, where R_y is the search radius when (y, r_y) is inserted. It follows that $R_y \leq \left(1 + \frac{1}{\varepsilon}\right) r_y$. Thus $X \subseteq B(q, R+r)$, where R is the maximum search radius at which a node of X was inserted and r is the maximum radius of the nodes of X. So $X \subseteq \mathsf{B}(q, (\frac{1+2\varepsilon}{\varepsilon})\alpha 2^k))$. Thus by the Standard Packing Lemma 3.1,

$$|X| \leq \Big(\frac{2\alpha(1+2\varepsilon)}{(\alpha-1)\varepsilon}\Big)^d.$$

Therefore the width of the search is $(1 + \frac{1}{\varepsilon})^{O(d)}$. ∎

THEOREM 6.1. *The running time of* ANN$(q, G, 1+\varepsilon)$ *on a greedy tree is* $\left(1 + \frac{1}{\varepsilon}\right)^{O(d)} \log \Delta$.

Proof. In Section 4 we show that the running time of ANN is $O(hw + \log \Delta)$, where w is the width and h is the height. By Theorem 5.1, $h = 2^{O(d)} \log \Delta$ and by Lemma 6.1 $w = (1 + \frac{1}{\varepsilon})^{O(d)}$. Therefore ANN runs in $\left(1 + \frac{1}{\varepsilon}\right)^{O(d)} \log \Delta$ time. ∎

7 KNN Search with the Greedy Tree.

The k nearest neighbor problem (KNN) is another natural problem to solve both exactly and approximately using a greedy tree. In this section, we explain how to solve it efficiently. Let $\mathbf{d}_k(q, P)$ denote the k^{th} largest distance (not necessarily distinct) of q to a point in P. The k nearest neighbors of q in a set P is the set $\mathsf{B}(q, \mathbf{d}_k(q, P))$. The $(1+\varepsilon)$-approximate k nearest neighbors of q in P is a superset of the k nearest neighbors that is contained in $\mathsf{B}(q, (1+\varepsilon)\mathbf{d}_k(q, P))$.

The standard approach to using a ball tree for k nearest neighbors is exemplified by the approach used for cover trees [2]. There, the search proceeds as in a NN search, but instead maintains a collection of the k nearest points discovered so far in analogy to the regular NN algorithm. The clear limitation of this approach is that one cannot use the heuristics that led to speedups for ANN because the k nearest nodes could contain more than k points, leading to an overestimate of the search radius. There is a relatively straightforward fix to this in the greedy tree. It leads to an $(1 + \frac{1}{\varepsilon})^{O(d)} \log \Delta + O(k')$ time algorithm, where k' is the number of approximate k nearest neighbors returned. Again, for polynomial-spread inputs, this matches the state of the art [2, 15]. An added bonus is that this algorithm allows one to $(1+\varepsilon)$-approximate the distance to the kth nearest neighbor in $(1 + \frac{1}{\varepsilon})^{O(d)} \log \Delta$ time. This is notable in that there is no dependence on k.

The heart of the algorithm is a heap N that stores a subset of viable nodes. The number of such points in the union of these nodes is called the weight of N. The weight is always at least k.

The purpose of N is to maintain a good approximation to $\mathbf{d}_k(q, P)$. The priority of a node in N is an upper bound on the distance from the query to any point in the node. The simple upper bound is the distance to the center plus the radius (by the triangle inequality). The priority of a node is set to be the minimum of the simple upper bound and the upper bound of its parent. The largest priority is called the radius of N.

The kNN search proceeds as in the Ann algorithm. A heap H of viable nodes (max-ordered by radius) is maintained and $H.radius$ is the largest radius in H. A node is considered viable if it could intersect $\mathsf{B}(q, N.radius/(1+\varepsilon))$. The nodes in N are always a subset of the nodes of H, so a packing of H implies a packing of N. If a ball in N is removed from H, then we also remove it from N and try to reinsert its children.

Every time a node is inserted into N, nodes are removed (in max order) until removing one more would decrease the weight below k. The heap N maintains the invariant that

$$\mathbf{d}_k(q,P) \leq N.radius \leq \mathbf{d}_k(q,P) + 2H.radius.$$

The lower bound follows because there are at least k points among the nodes of N and $N.radius$ is an upper bound on their distance. The upper bound follows from the triangle inequality (see Figure 5).

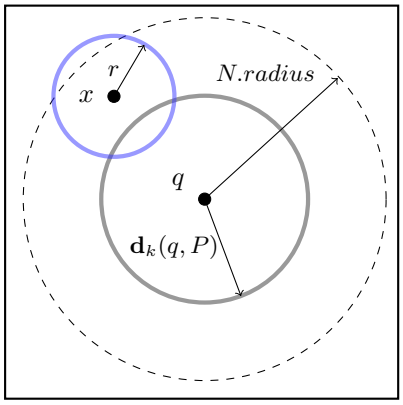


Figure 5: (x, r) is a node viable in H but not in N and $r \leq H.radius$.

The search stops when $H.radius \leq \frac{\varepsilon}{4}$. At that point, H contains a set of nodes whose union is a valid set of $(1+\varepsilon)$-approximate k nearest neighbors. Iterating over the nodes in H that are still viable then yields the desired set of points. The exact answer can be found in a second pass in $O(k')$ time where k' is the number of approximate k nearest neighbors. If instead, only the distance $d_k(q,P)$ is desired, then $N.radius$ gives a $(1+\varepsilon)$-approximation to this value without requiring an explicit enumeration the points.

To analyze the kNN algorithm, we can use the same pattern as for Ann. The only extra work is in maintaining the heap N. Using the same width analysis as Lemma 6.1, the size of N remains constant. Thus, identifying the nodes containing the $(1+\varepsilon)$-approximate k nearest neighbors takes $(1+\frac{1}{\varepsilon})^{O(d)} \log \Delta$ time. The extra $O(k')$ time is only needed to iterate over the output candidates.

8 Conclusion.

The overarching goal of this paper was to show that it is possible to have *both* a simple proximity search data structure *and* strong theoretical guarantees. It was motivated partly by the practical question, *What should one implement for general use?* as well as the pedagogical question, *What should one teach to new students in the field?* We offer the greedy tree for your consideration as a reasonable answer to both questions.

8.1 An Open Source Implementation. A Python implementation of the greedy tree is available as part of the `greedypermutation` Python package. The documentation and source code are available at `https://donsheehy.github.io/greedypermutation/`. The package can be installed with the following command.

```
pip install greedypermutation
```

The greedy tree implementation is in the `greedypermutation.balltree` module.

8.2 Applications. The most common application for nearest neighbor search is for classification in machine learning. For many such tasks, statistics about near neighbors are sufficient rather than returning the points themselves. Moore [16], gives a technique for modifying ball trees to store this information.

Another recent application of these greedy trees is in sketching persistence diagram for topological data analysis [21]. We believe that greedy trees should also allow for efficient metric data analysis, i.e. when the entire point set is considered as one point in a larger data set. In forthcoming work, we will show how to rapidly compute the Hausdorff distance and some of its variants using greedy trees. In other words, the greedy tree is a natural way to preprocess a metric data set into a linear size structure that speeds up approximate Hausdorff distance computations.

References

[1] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, sep 1975. doi:10.1145/361002.361007.

[2] S. Beygelzimer, A. Kakade and Langford J. Cover trees for nearest neighbor. In *Proceedings of the 23rd International Conference on Machine Learning*, pages 97–104, 2006.

[3] Paolo Ciaccia, Marco Patella, and Pavel Zezula. M-tree: An efficient access method for similarity search in metric spaces. In *Proceedings of the 23rd International Conference on Very Large Data Bases*, VLDB '97, pages 426–435, San Francisco, CA, USA, 1997. Morgan Kaufmann Publishers Inc.

[4] Kenneth L. Clarkson. Nearest neighbor queries in metric spaces. *Discrete & Computational Geometry*, 22(1):63–93, 1999.

[5] Kenneth L Clarkson. Nearest neighbor searching in metric spaces: Experimental results for sb (s). See http://cm. bell-labs. com/who/clarkson/Msb/white paper. pdf and http://cm. bell-labs. com/who/clarkson/Msb/readme. html, 2002.

[6] Kenneth L. Clarkson. Nearest-neighbor searching and metric space dimensions. In Gregory Shakhnarovich, Trevor Darrell, and Piotr Indyk, editors, *Nearest-Neighbor Methods for Learning and Vision: Theory and Practice*, pages 15–59. MIT Press, 2006.

[7] M.E Dyer and A.M Frieze. A simple heuristic for the p-centre problem. *Operations Research Letters*, 3(6):285–288, 1985. URL: https://www.sciencedirect.com/science/article/pii/0167637785900021, doi:https://doi.org/10.1016/0167-6377(85)90002-1.

[8] Yury Elkin. New compressed cover tree for k-nearest neighbor search, 2022. URL: https://arxiv.org/abs/2205.10194, doi:10.48550/ARXIV.2205.10194.

[9] R. A. Finkel and J. L. Bentley. Quad trees a data structure for retrieval on composite keys. *Acta Informatica*, 4(1):1–9, 1974. doi:10.1007/BF00288933.

[10] Teofilo F. Gonzalez. Clustering to minimize the maximum intercluster distance. *Theoretical Computer Science*, 38:293–306, 1985. URL: https://www.sciencedirect.com/science/article/pii/0304397585902245, doi:https://doi.org/10.1016/0304-3975(85)90224-5.

[11] A. Gupta, R. Krauthgamer, and J.R Lee. Bounded geometries, fractals, and low-distortion embeddings. In *44th Annual IEEE Symposium on Foundations of Computer Science, 2003. Proceedings*, pages 534–543, 2003.

[12] Sariel Har-Peled and Manor Mendel. Fast construction of nets in low dimensional metrics, and their applications. *SIAM Journal on Computing*, 35(5):1148–1184, 2006.

[13] Mike Izbicki and Christian R. Shelton. Faster cover trees. In *Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37*, ICML'15, pages 1162–1170. JMLR.org, 2015.

[14] David R. Karger and Matthias Ruhl. Finding nearest neighbors in growth-restricted metrics. In *Proceedings of the Thiry-Fourth Annual ACM Symposium on Theory of Computing*, STOC '02, pages 741–750, New York, NY, USA, 2002. Association for Computing Machinery. doi:10.1145/509907.510013.

[15] Robert Krauthgamer and James R. Lee. Navigating nets: Simple algorithms for proximity search. In *Proceedings of the Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '04, pages 798–807, USA, 2004. Society for Industrial and Applied Mathematics.

[16] Andrew W. Moore. The anchors hierachy: Using the triangle inequality to survive high dimensional data. *CoRR*, abs/1301.3877, 2013. URL: http://arxiv.org/abs/1301.3877, arXiv:1301.3877.

[17] Stephen M. Omohundro. Five balltree construction algorithms. Technical Report 562, ICSI Berkeley, 1989.

[18] Daniel J Rosenkrantz, Richard E Stearns, and Philip M Lewis, II. An analysis of several heuristics for the traveling salesman problem. *SIAM journal on computing*, 6(3):563–581, 1977.

[19] Donald R. Sheehy. greedypermutations. https://github.com/donsheehy/greedypermutation, 2020.

[20] Donald R Sheehy. One hop greedy permutations. In *Proceedings of the 32nd Canadian Conference on Computational Geometry*, 2020.

[21] Donald R. Sheehy and Siddharth Sheth. Sketching persistence diagrams, 2020. arXiv:2012.01967.

[22] Steven S. Skiena. *The Algorithm Design Manual*. Springer-Verlag London Ltd, 2008.

[23] Jeffrey K. Uhlmann. Satisfying general proximity / similarity queries with metric trees. *Information Processing Letters*, 40(4):175–179, 1991. URL: https://www.sciencedirect.com/science/article/pii/002001909190074R, doi:https://doi.org/10.1016/0020-0190(91)90074-R.

[24] Peter N. Yianilos. Data structures and algorithms for nearest neighbor search in general metric spaces. In *Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '93, pages 311–321, USA, 1993. Society for Industrial and Applied Mathematics.