Cover Trees Revisited: Exploiting Unused Distance and Direction Information

Zhi-Jie Wang, *Member, IEEE,* Mengdie Nie, Kaiqi Zhao, Zhe Quan, and Bin Yao

Abstract—The cover tree (CT) and its improved version are hierarchical data structures that simplified navigating nets while maintaining good runtime guarantees. They can perform nearest neighbor search in logarithmic time and provide efficient computation in practice. In this paper, we revisit cover trees for nearest neighbor search, and propose a more competitive method. The central idea of our method is to fully exploit the unused distance and direction information. More specially, our method introduces three novel concepts/techniques: (i) range list, (ii) quadrant information, and (iii) vectorial angle cosine. These techniques are seamlessly integrated into our suggested data structure and search algorithms. As an extra bonus, we explore approximate nearest neighbor and k nearest neighbor based on the proposed techniques, and present algorithms for handling updates. Extensive experimental results, based on both real and synthetic datasets, consistently demonstrate that our method is attractive and competitive, compared against existing cover tree structures for nearest neighbor search and its variants.

Index Terms—Cover tree, data mining, machine learing, data structure and algorithms

1 Introduction

Nearest neighbor search is a basic computational tool that can be applied to many domains including machine learning [15], data mining [31], and databases [39]. Data structure plays a key role in nearest neighbor search. It can be used to speed up the k-nearest neighbor classification algorithm, and many other tasks including clustering [4], [26], [5], localized support vector machines [28], dimensionality reduction [23], reinforcement learning [33], and image search [29], [30].

The basic nearest neighbor problem is as follows: Given a set $\mathcal{S}$ of n points in some metric space $(\boldsymbol{X}, d)$, the problem is to preprocess $\mathcal{S}$ so that given a query point $q \in \boldsymbol{X}$, one can find efficiently a point $p^* \in \mathcal{S}$ which minimizes $d(q, p^*)$, i.e., the distance between q and p^*. The naive method for computing nearest neighbor search problem involves a linear scan of all the data points and takes time $O(n)$. This method, however, is obviously expensive. So far, many data structures have been created to speed up this process. When the Euclidean dimension is low, one typical approach is to use the famous structure kd-trees [10]. When the Euclidean dimension is high, or the metric is non-Euclidean, the ball tree [34] is often used, since it is simple and competitive in many practical applications [13]. Although the ball tree is attractive for its simplicity, it provides only the trivial runtime guarantee that queries will take time $O(n)$. To this end, subsequent researches focused on obtaining stronger runtime guarantees. Thereby, more complicated data structures like the metric skip list [19] and the navigating net [20] were developed. Because these data structures were complex and had large constant factors, they were mostly of theoretical interest. Later, Beygelzimer *et al.* [1] proposed the cover tree (CT) —– a leveled tree where each level is a cover for the level beneath it. It is a hierarchical data structure that simplifies navigating nets while it maintains good runtime guarantees. It only consumes linear space and logarithmic time. Recently, Izbicki and Shelton [18] further developed the simplified nearest ancestor cover tree (SNACT), which provides a simpler definition, reducing the number of nodes from $O(n)$ to exactly n. Moreover, SNACT introduces an "additional" invariant, i.e., nearest ancestor invariant, that makes queries faster in practice.

Although the cover tree and its improved version achieved significant improvement for nearest neighbor search, several major insights motivate us to revisit cover tree structures. (i) Existing cover tree structures have integrated some distance information to improve the search performance, but some other intuitive and useful distance information seems to be ignored in previous works. For example, CT [1] employed the upper bound of $d(p, p')$ to prune unqualified nodes and/or subtrees, where p' denotes any descendant node of p; SNACT [18] used the $\texttt{maxdist}(p)$ to achieve this mission, where $\texttt{maxdist}(p)$ refers to the actual maximum value from p to any descendant node, and it is usually much tighter than the upper bound of $d(p, p')$. Nevertheless, the distance information from p to its children, children's children and so on, remains unexploited fully. (ii) Given two points p and p' in some metric space $(\boldsymbol{X}, d)$, traditionally, users easily associate them with the distance between them. Essentially, if one imagines point p as the original point of a coordinate system, then point p' must be located in some quadrant (or at some axis) of the coordinate system. The quadrant information is also useful for users to prune unqualified nodes. Yet, such a concept seems to be thoroughly novel, compared against existing cover tree structures. These ideas yield a novel data structure, dubbed as CT++ for short.

Based on the above data structure, we develop a much

- *Zhi-Jie Wang is with the College of Computer Science and the Ministry of Education Key Laboratory of Dependable Service Computing in Cyber Physical Society, Chongqing University, Chongqing 400044, China (e-mail: cszjwang@cqu.edu.cn).*
- *Mengdie Nie is with the Pinduoduo Company. E-mail: niemd@qq.com.*
- *Kaiqi Zhao is with the department of Computer Science, the University of Auckland, New Zealand. E-mail: kaiqi.zhao@auckland.ac.nz.*
- *Zhe Quan is with College of Computer Science, the Hunan University, Changsha, China . E-mail: quanzhe@hnu.edu.cn.*
- *Bin Yao is with the Department of Computer Science, the Shanghai Jiao Tong University, Shanghai, China . E-mail: yaobin@cs.sjtu.edu.cn.*

more efficient algorithm for nearest neighbor search. Our search algorithm not only takes full use of the proposed data structure, but also introduces a concept, called the vectorial angle cosine, to further speed up the search performance. The rationale behind this idea is surprisingly simple, yet it is useful and has not been exploited in existing cover trees based nearest neighbor search algorithms. In brief, given a line segment or vector $\overline{pq}$, one can image that there exists a hyperplane, which is vertical to $\overline{pq}$ and passes through endpoint p; then the hyperplane divides the data space into two *parts*. It is easy to see that, if p is currently the nearest neighbor of q, then any point in *the other part*, namely the part that does not contain p, is definitely not to be the nearest neighbor of q. (See Section 5.1 for a visual example.) Besides the above contributions, we also extend our techniques to approximate nearest neighbor search and k nearest neighbor search. Moreover, we present algorithms for handling updates, which are a little bit complicated than existing cover tree structures, since more attributes are needed to maintain. Nevertheless, our updating operations have only a little more cost in both theory and practice. To summarize, the main contributions of this paper are as follows.

- We present a new structure that is an improved version of SNACT. Our structure integrates unused distance and direction information that are not exploited in existing cover tree structures.
- We present a new algorithm for nearest neighbor search that takes full use of the proposed structure. Meanwhile, our algorithm integrates a technique called vectorial angle cosine that further speeds up the performance of our method.
- We extend our method to approximate nearest neighbor search and k nearest neighbor search. Besides, we also present the algorithms for handling updates including insertion and removal (i.e., deletion) operations.
- We conduct extensive experiments to demonstrate the efficiency and effectiveness of our proposed method.

The rest of the paper is organized as follows. Section 2 provides some preliminaries necessary for understanding the rest of the paper. Section 3 presents an overview of our solution. Section 4 presents the details of our data structure. Section 5 presents our method for nearest neighbor search. Sections 6 and 7 address how to achieve approximate and k nearest neighbor search, respectively. Section 8 handles the updates of our structure. Section 9 discusses and analyzes the experimental results, and finally Section 10 concludes the paper.

2 Preliminaries

This section first describes some notations and formulates the problem, and then revisits cover tree structures including CT [1] and SNACT [18], and finally reviews other relevant work.

Notations. Following prior works, we use T to denote a tree structure, and for a node p in T, we use children(p) and descendants(p) to denote its children and descendants, respectively. We also use desc(p) for short, when the context is clear. In addition, we use $maxdist(p)$ to denote $\mathrm{argmax}_{p' \in descendants(p)} d(p, p')$, as same as that in [18]. Note that our paper also slightly abuses the notations max and argmax, but its meaning should be clear from the context. Given a set $\mathcal{S}$ of n points in $(\boldsymbol{X}, d)$ where $\mathcal{S} \subset \boldsymbol{X}$, we denote the closed ball of radius r around p in $\mathcal{S}$ by $B_{\mathcal{S}}(p,r) = \{q \in \mathcal{S} : d(p,q) \le r\}$. The expansion constant of $\mathcal{S}$ is defined as the smallest value $c \ge 2$ such that $|B_{\mathcal{S}}(p,2r)| \le c \cdot |B_{\mathcal{S}}(p,r)|$ for every $p \in \boldsymbol{X}$ and $r > 0$. Meanwhile, the doubling constant, as an alternative of expansion constant, refers to the minimum value c such that every ball in $\boldsymbol{X}$ can be covered by c balls in $\boldsymbol{X}$ of half the radius [18], [7], [20]. Following [18], throughout this paper, if the constant c is mentioned, we refer to it as the doubling constant, unless stated otherwise. In addition, when $|\cdot|$ is used, it denotes the cardinality of the corresponding item. For example, $|\mathcal{S}| = n$.

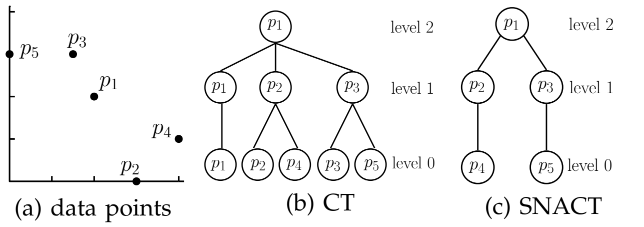


Fig. 1: Illustrations of the cover tree (CT) and the simplified nearest ancestor cover tree (SNACT).

Problem Statement. Let $\mathcal{S}$ be a set of n points in some metric space $(\boldsymbol{X}, d)$, where $\mathcal{S} \subset \boldsymbol{X}$. Given any two points p_1 and p_2 in $\boldsymbol{X}$, denote by $d(p_1, p_2)$ the distance in the metric space $(\boldsymbol{X}, d)$. Given a point p and a set Q of points in $\boldsymbol{X}$, we use $d(p, Q)$ to denote $\mathrm{argmin}_{p' \in Q} d(p, p')$. The basic nearest neighbor (NN) problem is as follows: Given $\mathcal{S} \subset \boldsymbol{X}$ and a query point $q \in \boldsymbol{X}$, it finds a point $p^* \in \mathcal{S}$ such that $d(q, p^*) = d(q, \mathcal{S})$.

Cover Tree. A cover tree T on a data set $\mathcal{S}$ containing n points is a "leveled" tree where each level is a "cover" for the level beneath it [1]. Each level is indexed by an integer scale i which decreases as the tree is descended. Every node in the tree is associated with a point in $\mathcal{S}$. Each point in $\mathcal{S}$ may be associated with multiple nodes in the tree; however, it requires that any point appears at most once in every level. Let C_i denote the set of points in $\mathcal{S}$ associated with the nodes at level i. The cover tree follows three invariants below for all i:

- (i) Nesting invariant, namely, $C_i \subset C_{i-1}$; this implies that once a point $p \in \mathcal{S}$ appears in C_i then every lower level in the tree has a node associated with p.
- (ii) Covering invariant, namely, for every $p \in C_{i-1}$, there exists a $q \in C_i$ such that $d(p,q) < 2^i$ and the node in level i associated with q is a parent of the node in level $i-1$ associated with p.
- (iii) Separation invariant, namely, for all distinct $p, q \in C_i$, $d(p,q) > 2^i$.

These invariants above are essentially the same as that used in navigating nets [20], except for (ii) where the cover tree requires only one parent of a node rather than all possible parents, significantly reducing the space to $O(n)$. Note that, for every node in level $i-1$, a navigating net keeps pointers to all nodes in level i that are within distance $\gamma \cdot 2^i$, where $\gamma \ge 4$ is some constant. Despite potentially throwing out most of the links in a navigating net, all runtime properties can be maintained.

To find the nearest neighbor of a query point q, it descends through the tree level by level, keeping track of a subset $Q_i \in C_i$ of nodes that may contain the nearest neighbor of q as a descendant. The algorithm iteratively constructs Q_{i-1} by expanding Q_i to its children in C_{i-1}, while it throws (or prunes) any child that cannot lead to the nearest neighbor of q. The major heuristic employed in the pruning step is based on an upper bound of $d(p, p')$. That is, given a node p in C_i, for any $p' \in \texttt{descendants}(p)$, $d(p, p') \leq 2^{i+1}$. Finally, the algorithm finds a point p (from the final candidate set) that has the minimum distance to q, and assigns it to p^*. For simplicity, it is easier to think of the tree as having an infinite number of levels (with C_∞ containing only the root, and with $C_{-\infty} = \mathcal{S}$). Algorithm 1 shows the framework of nearest neighbor search based on CT. Meanwhile, for the sake of intuition, Fig. 1(b) shows a running example of the cover tree (CT) indexing 5 two-dimensional data points shown in Fig. 1(a).

Simplified Nearest Ancestor Cover Tree. It is a variant of the cover tree. It introduces two simple yet useful ideas:

- (i) It provides a simpler definition. Specifically, it removes the nesting invariant in CT [1], and defines another invariant called the leveling invariant.
- (ii) It also introduces an *additional invariant*, named the nearest ancestor invariant. Specifically, it ensures that for every node p in the tree, if p' is p's ancestor, then p' must be the nearest ancestor of p. It defines that for any sibling node p^* of p', satisfying $d(p', p) \leq d(p^*, p)$.

The above two invariants make their method yield the following benefits: (i) The simpler definition (i.e., replacing nesting invariant with leveling invariant) reduces the number of nodes from $O(n)$ to exactly n. In other words, it makes every data point in $\mathcal{S}$ corresponding to exactly one tree node. (ii) The nearest ancestor invariant reduces the upper bound of $\texttt{maxdist}(\cdot)$, and so it makes queries faster in practice.

Algorithm 2 shows the nearest neighbor search algorithm based on SNACT, in which it calls (see Line 3) a function SubFindNN($\cdot$), detailed in Algorithm 3. In the algorithm, the major heuristic employed in the pruning step (Line 4) is based on $maxdist(\cdot)$, instead of the upper bound used in Algorithm 1. As pointed out in [18], the search algorithm takes $O(c^6 \log n)$ time, since any node in the tree can have at most $O(c^4)$ children, and the depth of any node in the tree is at most $O(c^2 \log n)$. To further understand the SNACT, Fig. 1(c) illustrates an example of SNACT that manages five data points shown in Fig. 1(a).

Other Related Work. In addition to the works mentioned previously (e.g., in Section 1), other relevant work can be generally classified into two categories: (i) cover trees' applications; and (ii) other approaches for nearest neighbor search and its variants such as approximate and k nearest neighbor queries. One of major applications is in the field of data exploration, where the cover trees are used for *query result diversification* [9], [24], [40], [11]. Another interesting application is for Bayesian reinforcement learning [33]. Moreover, cover trees are also used for *fingerprint recovery* [12] and *fast Sampling* [37]. As for category (ii), in the past decades various methods (e.g., ML-Index [8] were proposed, and there are no less than 1,000 papers that discussed nearest neighbor search [38], [16], [27], [21], [6], [32] and/or its variants [36], [17], [14], [3]. Our work is complementary to these works and is obviously different from them. A complete survey of all those papers goes beyond the scope of this paper. We refer the interested readers to recent works [25], [2], [38] and/or surveys [22], [35] for entry points into the literature. For example, a survey [22] experimentally compares the performance of various approximate nearest neighbour search algorithms (e.g., cover tree, KGraph, SRS, VP-tree). In this paper, we focus our attetion on investigating cover tree based solutions.

Algorithm 1 NNS_CT

```
Input: cover tree $T$, and query point $q$
Output: nearest neighbor $p^*$
1: Set $Q_\infty = C_\infty$, where $C_\infty$ is the root level of $T$;
2: for $i$ from $\infty$ down to $-\infty$ do
3:     Set $Q$ ={children($p$): $p \in Q_i$ };
4:     Set $Q_{i-1}$ ={ $p \in Q$: $d(q,p) \leq d(q,Q) + 2^i$ };
5: Set $p^*$ to be a point $p' \in Q_{-\infty}$ such that $d(q,p') = d(q, Q_{-\infty})$;
6: return $p^*$;
```

Algorithm 2 NNS_SNACT

```
Input: root node $r$, and query point $q$
Output: nearest neighbor $p^*$
1: $p^* = r$;
2: if children($r$) $\neq \emptyset$ then
3:     $p^*$ = SubFindNN($r, q, p^*$); // see Algorithm 3
4: return $p^*$;
```

Algorithm 3 SubFindNN

```
Input: node $p$, query point $q$, and nearest neighbor so far $p^*$
Output: updated $p^*$
1: if $d(p,q) < d(p^*,q)$ then
2:     $p^* = p$;
3: for each child $p' \in$children($p$) sorted by distance to $q$ do
4:     if $d(p^*,q) > d(p^*,p') - maxdist(p')$ then
5:         $p^*$ = SubFindNN($p', q, p^*$);
6: return $p^*$;
```

3 Solution Overview

This section describes our solution at a high level. We first briefly introduce the construction of our data structure, and then address the nearest neighbor search, based on our data structure.

In a nutshell, our construction algorithm involves two major steps. The first step is to construct a SNACT, which can be achieved by directly using the algorithms in [18]. The second step is to precompute the range list and quadrant information for nodes in the SNACT, and store them as the additional attribute information in corresponding nodes. Therefore, as same as the SNACT, our structure also maintains several invariants (i.e., leveling, covering, separating, and nearest ancestor invariants), requires only exactly n nodes, and any node in the tree has at most $O(c^4)$ children, and the depth of any node in the tree is at most $O(c^2 \log n)$.

To perform the nearest neighbor search, our method takes full use of the additional information mentioned earlier. Besides, we also leverage another important concept, vectorial angle cosine, that is computed on-the-fly. All these ideas are collaboratively used to prune unqualified sub-trees and/or nodes. Generally speaking, our search algorithm involves several main steps. First, we choose a point p as the current nearest neighbor, and then traverse the tree recursively. Initially, one can choose the root r as the current nearest neighbor. Then, in the process of traversing, at each level we exploit the heuristics (including the ideas of range list, quadrant, vectorial angle cosine) to prune some unqualified subtrees and nodes; and for the remainder nodes, we compute the distances to query point q, and update the current nearest neighbor. We continue the above steps, until the recursion reaches the leaf level of the tree. The current nearest neighbor in the last iteration is the nearest neighbor p^*. As same as the search algorithm in [18], the runtime of our search algorithm is also bounded by $O(c^6 \log n)$, since every node can have at most $O(c^4)$ children and the depth of the tree is bounded by $O(c^2 \log n)$. Therefore, our search algorithm has the same theoretical guarantee with that in [18], while it makes queries significantly faster than the competitors in practice.

4 Construction of Data Structure

Since the first step of our construction algorithm is essentially to construct SNACT, this section focuses more of our attention on the second step, i.e., computing the additional attributes and attaching them to the corresponding nodes. Generally, in our data structure we introduce two novel concepts: range list and quadrant information. In what follows, we first expatiate these two concepts, and then present the implementation for computing them, and finally analyze the runtime of constructing our data structure.

4.1 Range List

This idea is similar in spirit to the concept of $maxdist(p)$ mentioned in [18], where it pre-computes and stores the maximum distance between node p and all its descendants, instead of storing only $maxdist(p)$. The rationale behind our method is that, for every non-leaf node p, we exploit a list to store the range information for each level beneath the reference node p. That is, we store the maximum distance from p to its children, and then from p to its children's children, and so on. Although the essence of our idea is to use some "unused" distance information, this idea is especially useful for pruning some unqualified nodes that cannot be pruned by only $maxdist(p)$. By doing so, it shall directly speed up the search process, since we do not need to compute much more distance information on-the-fly.

Formally, we use $\texttt{level}(p)$ to denote the corresponding level of a given node p, and denote by rl the range list. Then, the jth ($|rl| - 1 \geq j \geq 0$) element in rl is computed as

$$rl[j] = \underset{\substack{q \in descendants(p) \\ level(p)-1-j \leq level(q) \leq level(p)-1}}{\operatorname{argmax}} d(p,q)$$

The equation above implies that $rl[j]$ stores the maximum distance from p to its descendants whose levels are in the corresponding range. Let rl_{final} denote the final element in rl, i.e., $rl_{final} = rl[|rl|-1]$. The following lemma shows the characteristic of our range list.

Lemma 1: *Given a range list rl which consists of $rl[0]$, $rl[1]$, $rl[2]$, ..., $rl[i]$, $rl[i+1]$, ..., rl_{final}, the values in rl are monotonously increasing.*

Proof. It follows directly from the fact that for any $j \geq 0$, $rl[j]$ stores the maximum value from p to its descendants ranging from $\texttt{level}(p)-1$ to $\texttt{level}(p)-1-j$, while $rl[j+1]$ stores the maximum value from p to its descendants ranging from $\texttt{level}(p)-1$ to $\texttt{level}(p)-2-j$. □

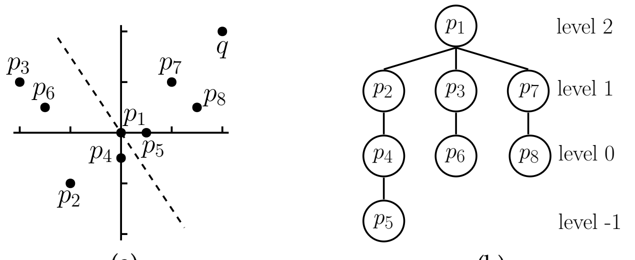


Fig. 2: Illustration of our strategies. Here q is the query point, p_1 is viewed as the original point, and the dashed line divides the data space into two parts.

For any point p', and assume p is the current NN (found so far) of q, our range list has two advantages, shown in the following lemmas.

Lemma 2: *When $d(p', q) - rl_{final} \geq d(p, q)$, the whole subtree rooted at p' can be pruned safely.*

Proof. Consider a sphere or hypersphere $\oplus$ centered at point p' with radius rl_{final}. Then, the minimal distance between $\oplus$ and query point q is $d(p', q) - rl_{final}$. By Lemma 1, we have that $\oplus$ covers p' and its descendants. In addition, $d(p', q) - rl_{final} \geq d(p, q)$ implies that, for any node p° contained in $\oplus$, $d(p^\circ, q) \geq d(p', q) - rl_{final}] \geq d(p, q)$, and so p° cannot be nearer than p to q. Pulling all together, this completes the proof. □

Lemma 3: *When $d(p', q) - rl_{final} < d(p, q)$, if there exists an appropriate j such that $d(p', q) - rl[j] \geq d(p, q) > d(p', q) - rl[j+1]$, then all nodes (in the subtree) whose levels are in $[\texttt{level}(p') - 1 - j,\ \texttt{level}(p')]$ can be pruned safely.*

Proof. The proof is similar to that of Lemma 2; omitted for saving space. □

Example 1: Fig. 2 shows a running example of range list. It can be seen that, for node p_3, its range list contains one element, i.e., $\{rl[0] = d(p_3, p_6)\}$, where $rl[0] = rl_{final}$. In this case, $d(p_6, q) - rl_{final} > d(p_1, q)$. Thus, the whole subtree rooted at p_6 can be pruned. In addition, consider node p_2 as another example. Its range list contains two elements, i.e., $\{rl[0] = d(p_2, p_4), rl[1] = d(p_2, p_5)\}$, where $rl_{final} = rl[1]$. Although rl_{final} in this case cannot prune the whole subtree rooted at p_2, we can find that, for $j = 0$, we have $d(p_2, q) - rl[0] \geq d(p_1, q) > d(p_2, q) - rl[1]$. This means that we can prune nodes (in the subtree) whose levels are in the range of $[\texttt{level}(p_2) - 1 - 0,\ \texttt{level}(p_2)]$. As a result, p_2 and p_4 can be pruned safely. □

4.2 Quadrant Information

As for quadrant information, the first idea we used is the opposite quadrant. The intuition behind this idea is that, for any node p (except the root node), one can view its corresponding parent node as the original point of a Cartesian coordinate system, while the vertical and horizontal axes of the coordinate system divides naturally the data space into many disjointed parts (or quadrants). It is easily understood that for any part $P^\perp$, there exists another part $P^\top$ that is "complementary" to $P^\perp$. Vividly speaking, for two points

p and p', if $p \in P^{\perp}$ and $p' \in P^{\top}$, then they are in the opposite quadrant. It is obvious that for any node p (except the root node), we can pre-compute and store its quadrant information by using its parent node as the original point. This way, in the search phase we can leverage this type of information to quickly prune nodes that are in the opposite quadrant of some reference point.

Formally, let ψ denote the dimension of data points, the horizontal and vertical axes of a ψ-dimensional Cartesian coordinate system divides the data space into 2^{ψ} parts, numbered as 0 to $2^{\psi}-1$. For a node p, denote by parent(p) its parent node. We can view parent(p) as the original point (of the coordinate system), and then define p's quadrant information below.

$$p_{\dagger} = \sum_{i=0}^{\psi-1} 2^i, \ \ s.t. \ p[i] < 0$$

The equation above essentially accumulates all 2^i such that $p[i] < 0$, where $p[i]$ refers to the ith dimension value of point p. Given any other point p', we say p and p' are in the opposite quadrant if and only if $p_{\dagger} + p'_{\dagger} = 2^{\psi} - 1$. This concept is helpful for us to prune unqualified nodes, since one can replace p' with the query point q. The following lemma establishes its usefulness.

Lemma 4: *Given a query point q, and a node p, if $p_{\dagger} + q_{\dagger} = 2^{\psi} - 1$, then p can be safely pruned.*

Proof. To prove this lemma, it suffices to show $d(q,p) > d(q,$ parent (p)). The central idea is to convert $p_{\dagger}$ and $q_{\dagger}$ into ψ-bit binary numbers based on the definition of quadrant information. Denote by $p[i]$ (resp., $q[i]$) the ith coordinate value of p (resp., q). Specifically, when $p[i]$ (resp., $q[i]$) is negative, we view the ith number of $p_{\dagger}$ (resp., $q_{\dagger}$) as 1; otherwise, it is viewed as 0. Then, $p_{\dagger} + q_{\dagger} = 2^{\psi} - 1$ implies that, for any $i \in [0, \psi - 1]$, the ith number of $p_{\dagger}$ is complementary to the ith number of $q_{\dagger}$ (e.g., 1010 vs. 0101). By the definition of quadrant information, we can conclude that the signs of $p[i]$ and $q[i]$ are opposite. In addition, our definition views parent(p) as the original point of the coordinate system. Thus, we have:

$$d(q,p) = \sqrt{\sum_{i=0}^{\psi-1}(q[i]-p[i])^2} > \sqrt{\sum_{i=0}^{\psi-1} q[i]^2} = d(q, parent(p))$$

. □

Example 2: See Fig. 2 again, one can first view p_1 as the original point, and then we can obtain the quadrant information of q and p_2. Here $q_{\dagger} = 0$, and $p_{\dagger} = 2^0 + 2^1 = 3$. In this case, $q_{\dagger} + p_{\dagger} = 2^2 - 1$, and so q and p_2 are in the opposite quadrant w.r.t. the original point p_1. Thus, p_2 can be pruned safely. □

Besides, we also apply another idea to prune points in some quadrants. The rationale behind this idea is that, assume p is the current NN (found so far) of query point q, for any i ($i \in [0, \psi - 1]$), we check whether $d(p,q) < \|q[i]\|$, and if so we perform the pruning operation by lerageving the hyperplane $\mathbb{P}$ that is vertical to the ith axis, where $q[i]$ refers to the ith dimension value of point q, and $\|\cdot\|$ refers to the absolute value. In other words, for any i, if $d(p,q) < \|q[i]\|$, points in $2^{\psi-1}$ quadrants can be pruned. For example, consider the example of $q = (1, -3)$, $d(p,q) = 2$, as shown in Fig. 3(a). In this example, when $i = 1$, $q[i] = -3$;

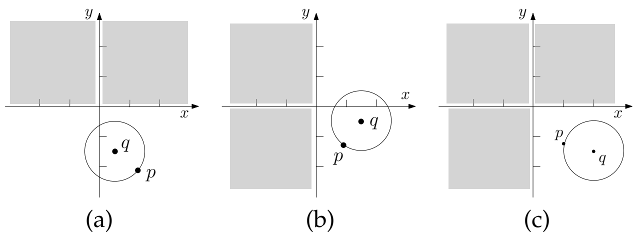


Fig. 3: Illustrations of quadrant pruning.

clearly, $\|q[1]\| = 3 > d(p,q) = 2$. Hence, we can perform the pruning operation by leveraging the hyperplane $\mathbb{P}$ that is vertical to y-axis (remark: here the hyperplane $\mathbb{P}$ is degraded into a line, i.e., x-axis). We can see that all points in the above two quadrants (cf., the grey parts) can be pruned. Similarly, Fig. 3(b) shows the case when $\|q[0]\| > d(p,q)$, where any point in the left two quadrants can be pruned. Note that, for each i ($i \in [0, \psi-1]$), if $d(p,q) < \|q[i]\|$ (in the best case), then points in $2^{\psi} - 1$ quadrants can be pruned; see e.g., Fig. 3(c). Essentially, the core observation of the above idea is that, $\|q[i]\|$ is the lower bound of the closest distance between q and those quadrants, given $d(p,q) < \|q[i]\|$. Formally, we have:

Lemma 5: *Given p is the current NN (found so far) of query point q, for any i ($i \in [0, \psi - 1]$), if $d(p,q) < \|q[i]\|$, then any point p' can be safely pruned if $p'[i] \times q[i] < 0$.*

Proof. By analytic geometry, it is trivial to show $d(q,p') > d(q,p)$; omitted for saving space. □

Algorithm 4 CompAttrs

```
Input: root $r$ of tree $T$,
Output: updated tree $T$
1: Set $Q_\infty = C_\infty$, where $C_\infty$ is the root level of $T$;
2: SubCompAttrs($r$); // see Algorithm 5
3: for $i$ from $\infty$ down to $-\infty$ do
4:     Set $Q$ ={children($p$): $p \in Q_i$ };
5:     for each $p' \in Q$ do
6:         SubCompAttrs($p'$); // see Algorithm 5
7: return $r$;
```

4.3 Implementation

The second step of our construction method is to compute the attributes (including range list and quadrant information), detailed in Algorithm 4. This algorithm first handles the root node r (Line 2), and then handles the rest of nodes (Lines 3-6). In this algorithm, the function SubCompAttrs($\cdot$) performs the specific calculation operations, detailed in Algorithm 5. This algorithm first performs initialization for the integer i, point set C, and a distance value $maxdist_i$ (Lines 1-2), and then computes range list rl (Lines 3-9). Note that, descendants(p)[i] denotes node p's descendants at level i (see Lines 2 and 9). Finally, our algorithm computes the quadrant information (Lines 10-16). In our implementation we check whether $p^*[j] - p[j] > 0$ (Line 15). This is essentially equal to the idea of viewing p^* as the original point.

Theorem 1: *The runtime for constructing our data structure is bounded by $O(c^{12} n \log n)$.*

Proof. Our construction algorithm includes two major steps. The first step is to build a SNACT with n nodes. It takes

Algorithm 5 SubCompAttrs

```
Input: node $p$ of tree $T$
Output: updated node $p$
 1: Set $i =$level$(p) - 1$;
 2: Set $Q =$descendants$(p)[i]$, $maxdist_i = 0$;
 3: while $Q \neq \emptyset$ do
 4:     if $i =$level$(p) - 1$ then
 5:         Set $maxdist_i = d(p, Q)$;
 6:     else
 7:         Set $maxdist_i = \text{argmax}(maxdist_i, d(p, Q))$
 8:     Set $rl[\text{level}(p) - 1 - i] = maxdist_i$, $i = i - 1$;
 9:     Set $Q =$descendants$(p)[i]$;
10: if $p$ is the root node then
11:     Set $p_\dagger = -1$; // a special value for root's quadrant
12: else
13:     Let $p^* =$parent$(p)$;
14:     for $j$ from 0 to $\psi - 1$ do
15:         if $p^*[j] - p[j] > 0$ then
16:             Set $p_\dagger = p_\dagger + 2^j$;
17: return $p$;
```

$O(c^{12} n \log n)$ time [18]. The second step shown in Algorithms 4 and 5 takes $O(c^6 n \log n)$ time, since the dominant step of Algorithm 5 is to compute rl that takes at most $O(c^6 \log n)$ time for any node. Combining all these results, this completes the proof. □

5 Nearest Neighbor Search

As mentioned in Section 3, our nearest neighbor search method fully exploits the information integrated in our data structure, and meanwhile it introduces another novel concept called vectorial angle cosine. This section first expatiates this idea and then presents the implementation for nearest neighbor (NN) search.

5.1 Vectorial Angle Cosine

The rationale of this idea is as follows. Assume that p is the current NN (found so far) of query point q, one can imagine that there exists a hyperplane $\mathbb{P}$ that is vertical to segment $\overline{pq}$ and passes through p. This implies that $\mathbb{P}$ divides the data space into two parts. Clearly, for any point p' located in the part that does not contain point q, it is never to be the NN of q. Here the vectorial angle cosine is just used to determine such a relation by exploiting $\overline{pq}$ and $\overline{pp'}$. Formally, denote by vac the vectorial angle cosine, we have:

Lemma 6: *Given the current NN p of query point q, and any other point p', if $\overline{pq} \cdot \overline{pp'} < 0$, then p' can be pruned safely.*

Proof. First, we can know that $vac = cos\theta = \frac{\overline{pq} \cdot \overline{pp'}}{|\overline{pq}| \times |\overline{pp'}|}$. Then, when $\overline{pq} \cdot \overline{pp'} < 0$, it implies that $vac < 0$, since $|\overline{pq}| \times |\overline{pp'}|$ is never smaller than 0. Meanwhile, $vac < 0$ implies that θ should be larger than 90°. By analytic geometry, hence the lemma holds. □

Example 3: Let's take p_3 in Fig. 2 as an example. Clearly, one can calculate the vectorial angle cosine between $\overline{p_3 p_1}$ and $\overline{q p_1}$. Here $vac = \frac{(-2\times 1.5 + 1 \times 1)}{\sqrt{(-2)^2+1} + \sqrt{1.5^2+1}} < 0$. Thus, p_3 can be pruned. □

Remarks: On one hand, users can easily understand from Section 4.2 that the opposite quadrant can be used to prune some nodes. On the other hand, users can also notice that, nodes in the quadrant same to query point q, may have more chances to be the nearest neighbor of q, although the nearest neighbor of q is not definitely in such a quadrant. □

Algorithm 6 NNS_OUR

```
Input: root node $r$, and query point $q$
Output: nearest neighbor $p^*$
 1: Set $p^* = r$;
 2: if children$(r) \neq \emptyset$ then
 3:     Set $RN = \emptyset$;
 4:     $p^* = \text{FuncFindNN}(r, q, p^*, RN)$; // see Algorithm 7
 5: return $p^*$;
```

5.2 Implementation

To perform nearest neighbor search, our method is detailed in Algorithm 6. In this algorithm, RN is used to store the remainder nodes at the corresponding level (Line 3). More specifically, $RN[k]$ stores the remainder nodes at level k. Here remainder nodes refer to those nodes that are not be pruned. For example, when we process nodes at level k, some node p might be pruned while children(p) may be not pruned currently. In this case, one can store them in $RN[k-1]$, and process them in the rest of steps. The function FuncFindNN$(\cdot)$ is invoked when the root node has children (Line 4).

Algorithm 7 covers the detailed implementation of this function, which is executed recursively. Firstly, it updates the current nearest neighbor if point p (to q) is nearer than the current nearest neighbor p^* (Lines 1-2). It then sets the integer j (which is used to handle or remember the level of some node), and the node set C (which is to be handled in this time of recursion); see Line 3. After that, it chooses a point p°, from the set C° of points whose quadrants equal to that of q, such that $d(p^\circ, q) = d(q, C^\circ)$; and it uses p° as the new nearest neighbor of q, if $d(q, p^\circ) < d(p^*, q)$ (Lines 4-7). The reasons we attempt to find such a point p° as the current nearest neighbor are twofold: (i) it is consistent to the observation mentioned in Section 5.1, namely, points in the quadrant same to q are more likely to be the nearest neighbor of q; and (ii) such a point may have a smaller distance to q, which in turn benefits to the range list based pruning operation executed in Lines 9-14, since a smaller $d(p^*, q)$ (cf., Lines 9 and 11) is more likely to prune a subtree rooted at some node p'.

If nodes or subtrees cannot be pruned by the range list based pruning operation, our algorithm shall execute the quadrant based pruning operation (Lines 15-20). Generally, as for the quadrant based pruning operation, it mainly contain two steps. Firstly, we need to remove the node p' from C, only if p' is in the *opposite* or *same* quadrant of q (Lines 15-17). This step is to avoid repetitive comparisons and calculations in the rest of steps. The reader could be curious why we here need to remove p' when it is in the same quadrant of q. The underlying reason is that, we have attempted to find the nearest neighbor of q in such a quadrant (recall Lines 5-7), other points (when we found such a p°), or all points in C° (when no such a point p° exists), must be not the nearest neighbor at the level j. Secondly, we then consider the case when point p' is in other quadrants (Lines 18-20). The goal of this step is to prune point p' satisfying the conditions mentioned in Lemma 5.

Algorithm 7 FuncFindNN

```
Input: node $p$, query point $q$, and nearest neighbor so far $p^*$, $RN$
Output: updated $p^*$
 1: if $d(p,q) < d(p^*,q)$ then
 2:     Set $p^* = p$;
 3: Let $j =$level$(p) - 1$, $C =$children$(p) \bigcup RN[j]$;
 4: Compute $q$'s quadrant $q_\dagger$ using $p$ as reference point;
 5: Let $C^\circ \subset C$ be set of points whose quadrants equal to $q_\dagger$;
 6: if $d(p^*,q) > d(q,C^\circ)$ then
 7:     Let $p^* = p^\circ$ // where $p^\circ \in C^\circ$ such that $d(p^\circ,q) = d(q,C^\circ)$
 8: for each $p' \in C$ do
 9:     if $d(p',q) - rl[0] \geq d(p^*,q)$ then
10:         for $i$ from $|rl| - 1$ to 0 do
11:             if $d(p',q) - rl[i] \geq d(p^*,q)$ then
12:                 if $i < |rl| - 1$ then
13:                     Put desc$(p')[j-i-1]$ into $RN[j-i-1]$;
14:                 Set $C = C - p'$;
15:     else if $p'$ is in the opposite or same quadrant of $q$ then
16:         Put desc$(p')[j-1]$ into $RN[j-1]$;
17:         Set $C = C - p'$;
18:     else if $d(p^*,q) < \|q[i]\|$ && $p'[i] \times q[i] < 0$ $(i \in [0,\psi])$ then
19:         Put desc$(p')[j-1]$ into $RN[j-1]$;
20:         Set $C = C - p'$;
21:     else if $\overline{p^*q} \cdot \overline{p^*p'} \leq 0$ then
22:         Put desc$(p')[j-1]$ into $RN[j-1]$;
23:         Set $C = C - p'$;
24:     else
25:         $p^* =$ FuncFindNN$(p',q,p^*,RN)$;
26: return $p^*$;
```

Correspondingly, when the above two pruning operations are not performed, our algorithm attempts to invoke the vectorial angle based pruning operation (Lines 21-23). It is worth noting that, although both the quadrant based pruning operation and the vectorial angle based pruning operation use the direction information, the latter needs to compute the vectorial information on-the-fly. This is why we use it after the quadrant based pruning operation, which needs less on-the-fly computation. Finally, when all the above operations cannot prune the corresponding node, our algorithm invokes the recursive function, FuncFindNN($\cdot$), to further explore the data points at the next level (Lines 24-25). A flowchart of NN search can refer to Appendix A.

Theorem 2: *Our method can perform nearest neighbor search at most $O(c^6 logn)$ time.*

Proof. It follows from two facts that (i) the depth of our data structure is at most $O(c^2 \log n)$ and the number of children of any node is at most $O(c^4)$, which are the same as that of SNACT; and (ii) the dominant step of Algorithm 6 is the recursive function, whose runtime is bounded by the depth of our data structure and the number of children of any node. □

6 Approximate Nearest Neighbor

Approximate nearest neighbor (ANN) search is useful when users care about more on the response speed (i.e., query latency) and allow a little deviation in the query accuracy. The approximate nearest neighbor search can be formulated as follows. Given a query point $q \in \boldsymbol{X}$, and the query accuracy $\epsilon > 0$, the approximate nearest neighbor search is to find a point $p \in \mathcal{S}$ satisfying $d(q,p) < (1+\epsilon)d(q,\mathcal{S})$. Note that, ϵ is a user-defined parameter, whose size can be adjusted according to specific application requirements.

Algorithm 8 kNN

```
Input: root node $r$, and query point $q$, $k$
Output: priority queue $\mathbb{Q}$ //where $\mathbb{Q}$ stores $k$ nearest neighbors
 1: Set $\mathbb{Q} = \emptyset$;
 2: if children$(r) \neq \emptyset$ then
 3:     Set $RN = \emptyset$; // $RN$ is the same as that in Algo. 6
 4:     $\mathbb{Q} =$ FuncFind$k$NN$(r,q,\mathbb{Q},RN)$; // see Algo. 9
 5: else
 6:     Enqueue $r$ into $\mathbb{Q}$; // only 1 node is returned in this special case
 7: return $\mathbb{Q}$;
```

To perform ANN search, the basic idea of our method is to use the upper bound for $d(q,C)$ and lower bound for the level j to stop the iterations when the intervals implied by the bounds are sufficiently small. The following lemma shows that we can terminate the iteration (or recursion) in the early stage.

Lemma 7: *Given the query accuracy ϵ, we can stop the iteration when $2^{j+1}(1+1/\epsilon) \leq d(q,C)$, where $j =$level$(p) - 1$ and $C =$children$(p) \bigcup RN[j]$.*

Proof. We prove it by showing $d(q,C) \leq (1+\epsilon)d(q,\mathcal{S})$, where $\mathcal{S}$ is the set of all n data points. Assume, without loss of generality, that the nodes in the j level is Q_j. One can easily know that Q_j is at distance at most 2^{j+1} from the exact nearest neighbor of q (i.e., the upper bound mentioned in Section 2). In addition, one can verify that $d(q,C)$ essentially equals to $d(q,Q_j)$. This is because the remainder nodes at the level j, namely $RN[j]$, are all possible nearest nodes at this level; other nodes that have been pruned, namely nodes in $Q_j - C$, are definitely not to be the nearest neighbor of q. Therefore, we have that $d(q,C) = d(q,Q_j) \leq d(q,\mathcal{S}) + 2^{j+1}$. Combining with $2^{j+1}(1+1/\epsilon) \leq d(q,C)$, this yields:

$$\begin{aligned} 2^{j+1}(1+1/\epsilon) \leq & d(q,Q_j) \\ \leq & d(q,\mathcal{S}) + 2^{j+1} \end{aligned}$$

The above formulation can be further rewritten as $2^{j+1} \leq \epsilon \cdot d(q,\mathcal{S})$. Therefore, putting all the above results together, we have $d(q,C) \leq d(q,\mathcal{S}) + 2^{j+1} \leq d(q,\mathcal{S}) + \epsilon d(q,\mathcal{S}) = (1+\epsilon)d(q,\mathcal{S})$. This completes the proof. □

Implementation. With the above concept in mind, it is not hard to achieve the approximate nearest neighbor search by revising the pseudocodes shown in Algorithm 7. Specifically, one can add a clause, "if $2^{j+1}(1+1/\epsilon) > d(q,C)$", as a condition to execute Lines 4-25. Naturally, when this condition does not hold, the algorithm shall not further execute the iterations (or recursion).

Complexity. Let d_{max} and d_{min} be the maximum and minimum interpoint distance in $\mathcal{S}$. The time complexity follows from the inspection of Lemma 2.6 in [20]. It takes at most $c^{O(1)\log\Delta + (1/\epsilon)^{O(\log c)}}$ time for an approximate nearest neighbor query, where Δ is the aspect ratio defined as $\Delta = d_{max}/d_{min}$. This result can be further written as $O(\log\Delta) + (1/\epsilon)^{O(1)}$. Therefore, the query time of our approximate nearest neighbor algorithm essentially is the same as those in [18], [20].

7 k Nearest Neighbors

The k nearest neighbor (kNN) search problem can be formulated as follows. Given a set $\mathcal{S}$ of n data points, and a query point $q \in \boldsymbol{X}$ where $\mathcal{S} \subset \boldsymbol{X}$, it aims to find k nearest points of q. To perform kNN search based on our data structure, the basic idea is to employ a priority queue $\mathcal{Q}$ to store k nodes at first, and then updates the priority queue $\mathbb{Q}$ when better candidates are found. Formally, let $p^{\nabla} \in \mathbb{Q}$ be the kth nearest neighbor of q currently. One can dequeue node p^{∇} from $\mathbb{Q}$ and enqueue node p' into $\mathbb{Q}$, if p' has the nearer distance than $d(q, p^{\nabla})$. Remark that, when $|\mathbb{Q}| < k$, one can view p^{∇} as the $|\mathbb{Q}|$th nearest neighbor of q currently. In this case we do not need to dequeue p^{∇} from $\mathbb{Q}$. In addition, the general pruning rules developed for the nearest neighbor search can also work, except that we here need to use p^{∇} in $\mathbb{Q}$ to build the pruning rules. More specifically, by Lemmas 2 and 3, we have an immediate corollary below.

Corollary 1: *Given the priority queue $\mathbb{Q}$ and any node p', the whole subtree rooted at p' can be pruned safely if $d(p\prime, q) - rl_{final} \geq d(p^{\nabla}, q)$. Otherwise, if there exists an appropriate j such that $d(p', q) - rl[j] \geq d(p^{\nabla}, q) > d(p', q) - rl[j+1]$, then all nodes (in the subtree) whose levels are in [`level`$(p') - 1 - j$, `level`(p')] can be pruned safely.* □

With the similar argument, by Lemma 6 one can obtain the following corollary.

Corollary 2: *Given the priority queue $\mathbb{Q}$, query point q, and any other point p', if $\overline{p^{\nabla}q} \cdot \overline{p^{\nabla}p'} < 0$, then p' can be pruned safely.* □

Implementation. The general framework of our method for kNN search is shown in Algorithm 8. In this algorithm, the function FuncFind$k$NN$(\cdot)$ is invoked when `children`(r) is not empty (Line 4). The details of the FuncFind$k$NN$(\cdot)$ are shown in Algorithm 9. The notations RN, p°, C, C°, and $q_\dagger$ used here have the same meaning with that in Section 5. This algorithm puts node p into $\mathbb{Q}$ directly only if $|\mathbb{Q}| < k$ (Lines 1-2). Otherwise, it checks whether p is nearer than $p^{\nabla} \in \mathbb{Q}$ to q, and updates $\mathbb{Q}$ if the above condition holds (Lines 3-4). After that, it attempts to find the nearer point in the quadrant same to $q_\dagger$ and updates $\mathbb{Q}$ using the newly found point p° (Lines 5-9), where $p^\circ \in C^\circ$ is such a point that $d(p^\circ, q) = d(q, C^\circ)$. Next, it employs pruning rules to remove unqualified nodes and/or subtrees (Lines 11-25), and yet it further invokes the recursive function for the remainder candidate node (Lines 26-27).

Complexity. Compared to algorithms in Section 5, kNN search algorithm here needs to use an extra $O(\log k)$ time for adjusting the queue $\mathbb{Q}$ in each iteration. Combining Theorem 2, it is not hard to verify that the overall runtime is bounded by $O(c^6 \log n(1 + \log k))$.

8 Handling Updates

Updates mainly include inserting new data point(s) and removing (a.k.a., deleting) node(s) from our data structure. These two operations are somewhat intricate, since inserting and/or removing node(s) may incur range list and quadrant information changed; besides, it may violate some invariants (e.g., nearest ancestor invariant). All of these are needed to be handled carefully. In this section, we address them in detail.

Algorithm 9 FuncFind$k$NN

```
Input: node $p$, query point $q$, and $\mathbb{Q}$, $RN$
Output: updated $\mathbb{Q}$
 1: if $|\mathbb{Q}| < k$ then
 2:     Enqueue $p$ into $\mathbb{Q}$;
 3: else if $d(p,q) < d(p^{\nabla},q)$ then
 4:     Dequeue $p^{\nabla}$ from $\mathbb{Q}$ and Enqueue $p$ into $\mathbb{Q}$;
 5: Let $j$ =level$(p) - 1$, $C$ =children$(p) \bigcup RN[j]$;
 6: Compute $q$'s quadrant $q_\dagger$ using $p$ as reference point;
 7: Let $C^\circ \subset C$ be set of points whose quadrants equal to $q_\dagger$;
 8: if $d(p^{\nabla},q) > d(q,C^\circ)$ then
 9:     Dequeue $p^{\nabla}$ and Enqueue $p^\circ$;
10: for each $p' \in C$ do
11:     if $d(p',q) - rl[0] \geq d(p^{\nabla},q)$ then
12:         for $i$ from $|rl| - 1$ to 0 do
13:             if $d(p',q) - rl[i] \geq d(p^{\nabla},q)$ then
14:                 if $i < |rl| - 1$ then
15:                     Put desc$(p')[j-i-1]$ into $RN[j-i-1]$;
16:                 Set $C = C - p'$;
17:     else if $p'$ is in the opposite or same quadrant of $q$ then
18:         Put desc$(p')[j-1]$ into $RN[j-1]$;
19:         Set $C = C - p'$;
20:     else if $d(p^*,q) < \|q[i]\|$ && $p'[i] \times q[i] < 0$ ($i \in [0,\psi]$)
        then
21:         Put desc$(p')[j-1]$ into $RN[j-1]$;
22:         Set $C = C - p'$;
23:     else if $\overline{p^{\nabla}q} \cdot \overline{p^{\nabla}p'} \leq 0$ then
24:         Put desc$(p')[j-1]$ into $RN[j-1]$;
25:         Set $C = C - p'$;
26:     else
27:         $\mathbb{Q}$ = FuncFind$k$NN$(p',q,\mathbb{Q},RN)$;
28: return $\mathbb{Q}$;
```

8.1 Insertion

Given the root node r of our tree structure T, a new data point $q \in \boldsymbol{X}$, the insertion operation aims to insert q into T at the corresponding location, while the new tree can still maintain several invariants and also the correct attribute information (e.g., range list and quadrant information).

The basic idea of our method is to use separating, covering and/or nearest ancestor invariants to determine the rough location to be inserted, and meanwhile our method maintains two sets, S_r and S_q, to store the nodes whose range list and quadrant information might be changed, respectively. After finishing the initial insertion, although point q can maintain several invariants, the descendants of its sibling nodes might violate some invariant (specifically, the nearest ancestor invariant). Our method readjust (or rebalance) them by using the approach in [18]. This approach

Algorithm 10 Insert

```
Input: root $r$ of tree $T$, data point $q$ to be inserted,
Output: root $r$ of updated tree $T$
 1: $S_r = \emptyset$, $S_q = \emptyset$;
 2: $(r, S_r, S_q) \leftarrow$ SubInsert$(r, q, S_r, S_q)$; // see Algo. 11
 3: for each node $p \in$siblings$(q)$ do
 4:     for each node $p' \in$children$(p)$ do
 5:         ($\mathbb{M}$, $\mathbb{S}$, $p'$) $\leftarrow$ Readjust($q$, $p$ ,$p'$);
 6:         for each $m \in \mathbb{M}$ do
 7:             $(q, S_r, S_q) \leftarrow$ SubInsert($q$,$m$,$S_r$, $S_q$); // see Algo. 11
 8: Set $S_r = S_r \bigcup$ancestor$(q)$, $S_q = S_q \bigcup q$;
 9: Compute $rl$ for nodes in $S_r$ and quadrants for nodes in $S_q$;
10: return $r$;
```

Algorithm 11 SubInsert

```
Input: root node $r$ of tree or subtree $T$, data point $q$ to be
    inserted, $S_r$ and $S_q$
Output: root $r$ of updated tree or subtree $T$
 1: Set $i =$level $(r)$;
 2: if $d(r,q) > 2^i$ then
 3:   if $d(r,q) \geq 2^{i+1}$ then
 4:     $p \leftarrow$ Choose a leaf node from $T$;
 5:     Set $S_r = S_r \bigcup$ancestor$(p)$;
 6:     Set point $p$ as the parent of $r$, $S_q = S_q \bigcup \{p, r\}$;
 7:   else
 8:     Set $q$ as the parent of $r$, $S_r = S_r \bigcup \{q\}$, $S_q = S_q \bigcup \{q, r\}$;
 9: else
10:   Let $C =$children$(r)$, $i =$level$(r)$-1;
11:   Find a point $c^* \in C$ such that $d(q, c^*) = d(q, C)$
12:   if $d(c^*, q) < 2^i$ then
13:     $(c^*, S_r, S_q)\leftarrow$ SubInsert$(c^*, q, S_r, S_q)$;
14:   else
15:     Set $q$ as the child of $r$;
16:     Set $S_r = S_r \bigcup$ancestor$(q)$, $S_q = S_q \bigcup \{q\}$;
17: return $\{r, S_r, S_q\}$;
```

gets two sets of points, say $\mathbb{S}$ and $\mathbb{M}$. The points in $\mathbb{S}$ need to be reinserted along some specific path, which is handled by a function called Readjust($\cdot$). Yet, the points in $\mathbb{M}$ cannot remain at some specific path. For these points, we need to insert them into the subtree rooted at q. After we insert the new node and readjust some nodes, the range list and quadrant information of some nodes might be changed, therefore we finally recompute their attribute information.

Implementation. Let siblings(p) denote a node p's sibling nodes. Algorithm 10 shows the framework of our method. This algorithm first inserts q at some location (Line 2), by using a function named SubInsert($\cdot$). Then, it handles nodes, specifically siblings(q), whose descendants might violate the nearest ancestor invariant (Lines 3-8). Note that, the function, Readjust($\cdot$) shown in Line 5, is the same as the Rebalance($\cdot$) function in [18]. It is a recursive function, in which points in $\mathbb{S}$ shall be reinserted along some path; yet, points in $\mathbb{M}$ are to be inserted into the subtree rooted at q (Lines 6-7). Finally, it collects all nodes whose attribute information might be changed, and computes attributes for nodes in S_r and S_q (Lines 8-9). Note that, to understand Algorithm 11, several places are key of points: (i) exploiting the distance bound of covering invariant (Lines 2 and 9); (ii) using the distance bound of descendants (Lines 3 and 7); (iii) leveraging the property of the nearest ancestor (Line 11); and (iv) utilizing the distance bound of separating invariant (Line 13).

Theorem 3: *The runtime for insertion operation is bounded by* $O(c^{20} \log n)$.

Proof. It stems from the facts that (i) the dominant step of our insertion algorithm is the Readjust operation, which takes at most $O(c^{12} \log n)$ [18]; (ii) the Readjust function is to be called at most $O(c^4) \times O(c^4)$ times, since each loop (Line 3 or 4) is bounded by $O(c^4)$ (i.e., the width of the tree [1], [18]). Pulling all together, this completes the proof. □

8.2 Removal

The removal operation aims to delete some target point q from the tree T, while the new tree still maintains these invariants and also the correct attribute information. The central idea of our method is to attach subtrees rooted at nodes to some other nodes. Given a subtree rooted at node p, in what follows, we use leaf(p) to denote all leaf nodes of node p.

Implementation. Algorithm 12 covers the detailed implementation of our method. When q is just the root node or leaf node, it is trivial as shown in Lines 1-6. For other cases, it is somewhat complicated. The basic idea of our method is to find a sibling node of q that can attach the subtree rooted at q's child(ren), as shown in Lines 9-22. It is possible that subtrees rooted at some nodes cannot be attached to any

Algorithm 12 Remove

```
Input: root node $r$ of tree $T$, to be removed point $q$
Output: root $r$ of updated tree $T$
 1: Set $S_r = \emptyset$, $S_q = \emptyset$;
 2: if $q$ is the root node $r$ then
 3:   $p \leftarrow$Choose a leaf node, and replace root $q$ with point $p$;
 4:   Set $S_r = S_r \bigcup \{p\}$, $S_q = S_q \bigcup$children$(p)$;
 5: else if $q$ is the leaf node then
 6:   Set $S_r = S_r \bigcup$ancestor$(q)$, and delete $q$;
 7: else
 8:   Set $S =$siblings$(q)$;
 9:   if $|S| > 0$ then
10:     Set $C =$children$(q)$, $S_r = S_r \bigcup$ancestor$(q)$;
11:     $R \leftarrow$SubtreeAttach$(r, S, C)$; // see Algo. 13
12:     Delete subtree rooted at $q$;
13:     for each point $p$ in $C - R$ do
14:       $S_r = S_r \bigcup$ancestor$(p)$, $S_q = S_q \bigcup$ancestor$(p)$;
15:     while $|R| > 0$ do
16:       Set $C' = \emptyset$, $S' = \emptyset$;
17:       for each point $p \in R$ do
18:         Set $C' = C' \bigcup$children$(p)$;
19:         $r \leftarrow$Insert$(r, p)$; // see Algo. 10
20:       for each point $p' \in S$ do
21:         Set $S' = S' \bigcup$children$(p')$;
22:       $R \leftarrow$SubtreeAttach$(r, S', C')$; // see Algo. 13
23:   else
24:     Find a leaf node $p^*$ $\in$leaf$(q)$ such that
        $d($parent$(q)$,$p^*) = d($parent$(q)$,leaf$(q))$;
25:     Set $S_r = S_r \bigcup$ancestor$(p^*)$, replace $q$ with $p^*$, $S_q =$
        $S_q \bigcup \{p^*\}$, and delete $q$;
26: Compute $rl$ for nodes in $S_r$ and quadrants for nodes in $S_q$;
27: return $r$;
```

Algorithm 13 SubtreeAttach

```
Input: root node $r$ of tree $T$, a set $S$ of candidate parent nodes,
    a set $C$ of subtrees' root nodes to be attached to nodes in
    $S$
Output: a set $R$ of points that cannot be attached to nodes in
    $S$
 1: if $|C| > 0$ then
 2:   for each point $p \in C$ do
 3:     for each $p' \in S$ sorted by distance to $p$ do
 4:       if $p'$ as the parent of $p$ is without violating several
          variants then
 5:         Attach subtree rooted at $p$ as the descendants of
            $p'$;
 6:     if $p$ cannot be as the child of any $p' \in S$ then
 7:       Set $R = R \bigcup p$;
 8: else
 9:   $R = \emptyset$
10: return $R$;
```

sibling node of q (Lines 15-22). In this case, we shrink the subtrees "level by level", and attempt to attach them to the corresponding nodes (see Lines 18, 21 and 22). Another case is that q may have no any sibling nodes (Lines 23-25). Our method employs the node in leaf level to handle it, which is similar to that in Algorithm 11. Note that, the function SubtreeAttach($\cdot$) shown in Algorithm 13 attempts to attach some subtrees, rooted at nodes in C, to some candidate parent nodes in S, and it returns a set of nodes, R, whose subtrees cannot be attached to any nodes in S.

Theorem 4: *The runtime for removal operation is bounded by* $O(c^{24}\log n)$.

Proof. It is easy to verify that the dominant step of our method takes $O(c^{20}\log n)$ time, since it calls the insertion operation (Line 19). In addition, the function Insert($\cdot$) is called at most $O(c^4)$ times, since the children of any node is at most $O(c^4)$. Hence, the theorem holds. □

9 Experimental Study

This section first describes the experimental settings (Section 9.1), and then covers the experimental results (Sections 9.2∼9.3), and finally discusses the limitations of our algorithms and summarizes our findings (Section 10).

9.1 Setup

Datasets. In our experiments, we employed several standard benchmark datasets which are obtained from http://archive.ics.uci.edu/ml/index.php to evaluate our algorithms. The used datasets include yearpredict, corel, covertype, artificial40, which are briefly described below:

- *yearpredict*. It is a dataset about the prediction of the release year of a song from audio features.
- *corel*. It is a dataset containing image features extracted from a Corel image collection.
- *covertype*. It is a forest cover type dataset.
- *artificial40*. It is a 10000×40 artificial dataset consists of randomly generated numbers.

dataset	number of points	dimension
yearpredict	515,345	90
covertype	581,012	55
corel	68,040	32
artificial40	10,000	40
syn	10,000∼10,000,000	10∼150

TABLE 1: Summary information of our datasets

Besides artificial40, we also used another synthetic dataset (called *syn*) when we studied the impact of different parameters. The points of the *syn* dataset was generated randomly. Let ψ and n be the dimension and number of data points, respectively. For the *syn* dataset, we set n =[10,000, ..., 10,000,000] and ψ =[10, ..., 150], where $n = 50000$ and $\psi = 20$ are the default settings. In addition, when we studied the kNN search, k was set to [1, ..., 20], where $k = 10$ is the default value. For ease of reference, Table 1 summarizes the statistics of these datasets.

Competitors. For brevity, we use CT++ to denote our method. In our experiments, we compared with the following competitors.

- CT: It is the original cover tree (CT) proposed in [1].
- SNACT: It is the simplified nearest ancestor cover tree (SNACT) proposed in [18].

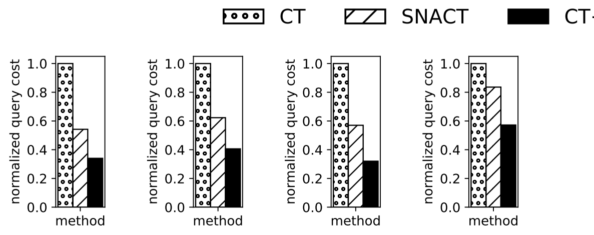
To investigate the effectiveness of the proposed techniques, we also implemented several other algorithms, which are the variants of SNACT: (i) QI, which employs the quadrant information; (ii) RI, which employs the range list information; and (iii) DI, which employs the direction information (including quadrant and vectorial angle). For a fair comparison, all algorithms were implemented in C++, and all tests were executed on a single machine with an Intel(R) Core(TM) E5-2620 CPU 2.40GHZ and 64GB RAM.

Metrics. To measure the query cost, we randomly generated 1,000 query points and used them as the query inputs, the execution time for performing the corresponding queries was recorded. As for the construction cost, we refer to the time for building the corresponding data structures (e.g., CT, SNACT, and CT++). When we studied the update cost, we randomly chose the points to be removed from the corresponding datasets, and then these points were inserted into the corresponding data structures. We removed/inserted 1000 points from/into the data structures, and recorded its update time. Similar to [18], we also normalized query/construction/update cost by the corresponding baseline, for ease of comparison. For example, two methods, say A and B, use 10 and 2 seconds for query, respetively, then the normalized values of the query cost are 1 and 0.2, respectively. That is, the number "1" or "1.0" in our results refers to the cost of the corresponding baseline. In addition, sometimes we also use "K" and "M" to denote the numbers "1,000" and "1,000,000" for short.

Roadmap. We first cover the main experimental results including query performance and effectiveness of proposed techniques. Then, we show the impact of different parameters including n, k, ϵ, and ψ. Remark that, other findings/results are placed in Appendix, due to space limit.

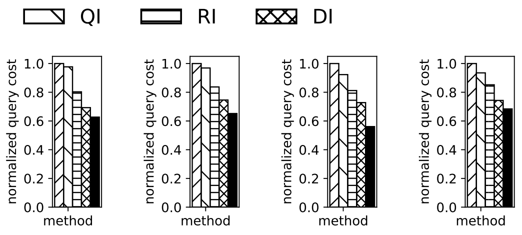
9.2 Query Performance and Effectiveness

The central goal of this work is to improve existing cover tree structures for nearest neighbor search, thus we studied the query performance and effectiveness of the proposed



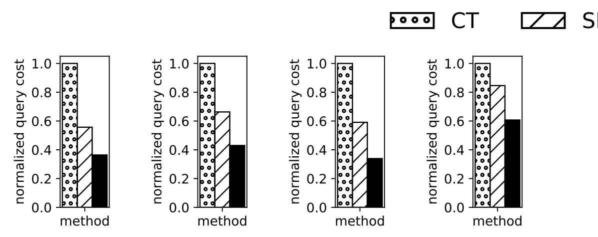
(a) yearpredict (b) covertype (c) corel (d) artificial40

Fig. 4: Query cost comparison.



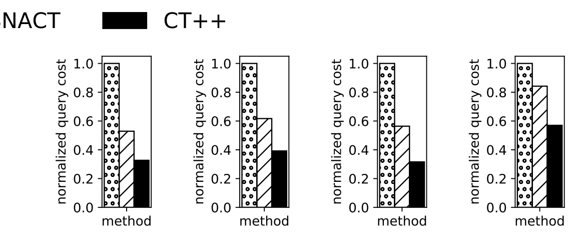
(a) yearpredict (b) covertype (c) corel (d) artificial40

Fig. 5: Study of effectiveness.



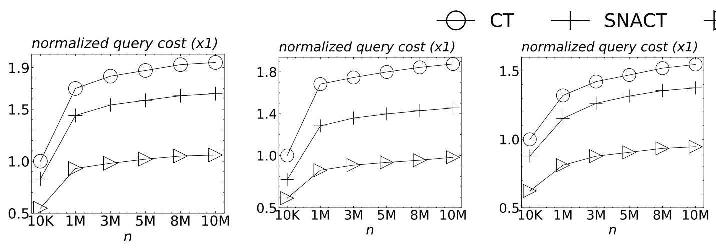
(a) yearpredict (b) covertype (c) corel (d) artificial40

Fig. 6: Approximate query cost comparison.



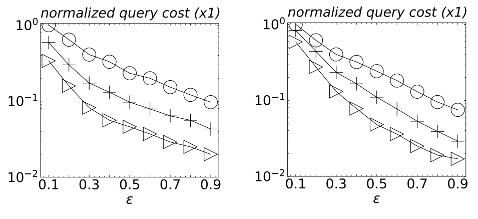
(a) yearpredict (b) covertype (c) corel (d) artificial40

Fig. 7: kNN query cost comparison.



(a) NN (b) kNN (c) ANN

Fig. 8: Vary n.



(a) corel (b) syn

Fig. 9: Vary ϵ.

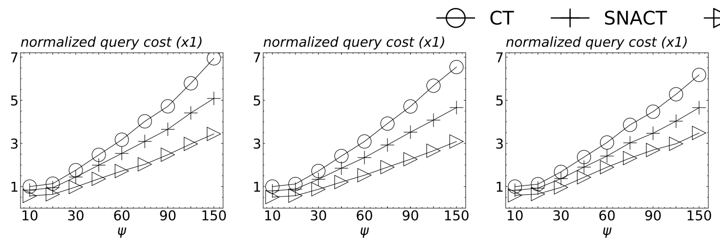
techniques in the first set of experiments. All the experimental results in this part were obtained based on four standard benchmark datasets mentioned in Section 9.1.

NN query. Figure 4 covers the comparison results of the nearest neighbor search. It can be seen that, among these methods our proposed method, CT++, achieves the best performance for all these datasets. For example, on the corel dataset, CT++ takes only 3.955 seconds to perform nearest neighbor search for 1,000 query points (i.e., about 4 milliseconds for each single nearest neighbor query on average), whereas SNACT and CT take 7.05 and 12.385 seconds, respectively. This essentially demonstrates the competitiveness and efficiency of our proposed method. On the other hand, we find that SNACT reduces the query cost by 16.4%∼45.8%, while our method, CT++, can reduce the query cost by 42.8%∼68%, compared against CT. Particularly, even for the stronger baseline SNACT, our method outperforms it by 31.6%∼43.9%. These observations further demonstrate the efficiency and competitiveness of our proposed method. Moreover, when we look a bit deeper into the results, we find that CT++ achieves different performance improvements on these datasets with different distributions, implying that data distributions may affect the performance.

Effectiveness. To understand the effectiveness of various techniques, we tested several algorithms including SNACT, QI, RI, DI, and CT++ on these benchmark datasets. Figure 5 plots the comparison results. On one hand, we realized that each technique is benefitial to the performance speedup. For example, on the corel dataset QI, RI, and DI used 6.507, 5.725, 5.132 seconds, respectively. Yet, SNACT used 7.05 seconds. Particularly, our method CT++, which integrates all these techniques, achieves the maximum performance speedup (about 37.3%, 34.8%, 43.9%, and 31.6% on yearpredict, covertype, corel, and artificial40 datasets, respectively). Additionally, it is usually that more nodes/subtrees are pruned, less distance computations invlove. To verify, we examined the number of distance computations. In general, CT++ involves less distance computations than others. For example, on the core dataset, the number of distance computations by CT++ is about 5.4K, while that of DI, RI, QI and SNACT are about 7.1K, 8.2k, 9.4K, 10.1K, respectively. Combining all these observations, it essentially confirms that pruning nodes/subtrees indeed speeds up the search process.

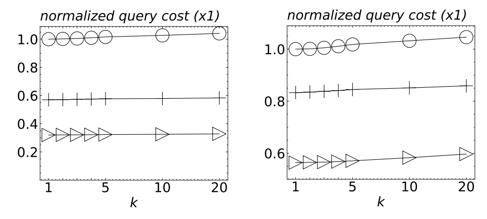
ANN query. Since SNACT [18] did not cover approximate nearest neighbor search, we adapted our approximate nearest neighbor search algorithm to achieve this. In brief, we revised Algorithm 3 by adding an "if" clause before Lines 3-5. The "if" clause is similar in spirit to that mentioned in Section 6. By fixing $\epsilon = 0.1$ and using 1,000 randomly generated query points, we studied the approximate query performance of these three methods. Figure 6 plots the comparison results over four benchmark datasets. Generally, compared against CT, SNACT obtains 15.4%∼44.3% performance speedup, while our method achieves 39.3%∼66.1% performance speedup. Particularly, even for the stronger competitor SNACT, our method outperforms it by 28.3%∼42.6%. These evidences further demonstrate the competitiveness of our proposed method, and also verify, from another perspective, the effectiveness of techniques developed in Sections 4 and 5. On the other hand, as expected, on all these datasets the approximate query cost is less than that of exact nearest neighbor query. For example, on the corel dataset CT, SNACT and CT++ used 12.385, 7.05 and 3.955 seconds, respectively, for performing exact nearest neighbor search based on 1,000 randomly generated points. Yet, with the same query points, they used only 3.494, 2.065 and 1.184 seconds for performing approximate nearest search, respectively. These results essentially demonstrate that the early termination strategy suggested in Section 6 is effective.

kNN query. As for kNN search, both CT [1] and SNACT [18] did not cover kNN query algorithms. We adapted our kNN algorithm to achieve this. That is, we also used a priority queue to store current k nearest neighbors and updated the queue in the search phase; others are the same as that in CT [1] and SNACT [18], respectively. By fixing $k = 10$ and using 1,000 randomly generated query points, we studied kNN query performance of these three methods. Figure 7



(a) NN (b) kNN (c) ANN

Fig. 10: Vary ψ.



(a) corel (b) syn

Fig. 11: Vary k.

plots the comparison results over four benchmark datasets. It can be seen that, among these methods CT++ achieves the best performance for all these datasets. Specifically, compared against CT, it achieves 67.3%, 60.8%, 68.4%, and 43% performance speedup on yearpredict, covertype, corel, and artificial40 datasets, respectively. For the stronger baseline SNACT, our method outperforms it by 38.2%, 36.5%, 44%, 32.3% on these datasets, respectively. This demonstrates the competitiveness of our method. Meanwhile, it essentially reflects, from another perspective, the effectiveness of main techniques suggested in Sections 4 and 5, since these three kNN algorithms used the same strategy discussed in Section 7. Another interesting finding is that, on all these datasets, the kNN query cost is only a little bit more expensive than that of nearest neighbor query, instead of k times cost of nearest neighbor query. For example, on the corel dataset, CT++ used 3.955 and 3.981 seconds to perform nearest neighbor search and kNN search respectively, based on 1,000 randomly generated points. This implies that the strategy suggested in Section 7 is effective (remark: a naive method to find k nearest neighbors is to execute k times nearest neighbor queries. Such a method is definitely expensive, since it could be about k times slower than our proposed method).

9.3 Impact of Parameters

This section presents *the impact of different parameters*, including dataset size n, number of nearest neighbors k, data point dimension ψ, and query accuracy ϵ, respectively.

Effect of n. Fixing $\psi = 20$, $k = 10$, and $\epsilon = 0.1$, we studied the impact of n on NN search, kNN search, and ANN search, respectively. Figure 8 plots the experimental results by varying n from 10K to 10M. It can be seen that there are several major features. Firstly, for all these algorithms the query cost goes up when n increases. This is mainly because there are more nodes in the corresponding trees, and thus these algorithms need more time to perform the corresponding searches. Nevertheless, the growth speed of the query cost is acceptable for all these algorithms. For example, even for the baseline method CT, when n is set to 10M it used less than 2 times of query cost, compared against n =10K. This essentially demonstrates that these algorithms have good scalability. Secondly, when we look a bit deeper, we can see that the curve of CT++ goes up slower. This implies that our proposed method has better scalability, compared against the baselines. This is mainly owing to the much powerful pruning mechanisms integrated in our method, which avoid more node traversals. Thirdly, our proposed method CT++ is obviously better than the competitors for all these queries, as shown in Figures 8(a), 8(b) and 8(c). These results are basically consistent with the results reported in Section 9.2. In other words, the results shown in Figure 8 essentially further demonstrate the efficiency and competitiveness of our proposed method.

Effect of ϵ. We studied the impact of ϵ on the approximate query using real and synthetic datasets. Fixing $n = 50,000$, $\psi = 20$ for the synthetic dataset, and we varied ϵ from 0.1 to 0.9 for both real and synthetic datasets. For the real dataset, we used corel as a representative, since other several datasets have the similar performance tendency. Figure 9 plots the experimental results. From these results, one can observe several major features. Firstly, the query cost goes down when ϵ increases. This is mainly because when larger ϵ is used, the bound used in the pruning strategy shall be more relax. This way, more levels (or nodes) in the tree are free from being traversed, and so the query cost is less. Secondly, the proposed method CT++ is obviously superior than the baselines. Specifically, compared against CT and SNACT, on average it reduces the query cost by about 70.9% and 46.5% (resp., about 78.4% and 52.1%) on the synthetic (resp., real) dataset, respectively. These results further validate the efficiency and competitiveness of our method.

Effect of ψ. Fixing $n = 50,000$, $k = 10$, and $\epsilon = 0.1$, we studied the impact of ψ on NN search, kNN search, and ANN search, respectively. Figure 10 plots the experimental results by varying ψ from 10 to 150. It can be seen that the curves go up when ψ increases, implying that larger ψ incurs much more query cost. This is mainly because computing distances between points are much more time-consuming when ψ is larger. Nevertheless, the query cost is acceptable. For example, even if we set $\psi = 150$, our proposed method used only 12.46 seconds to perform nearest neighbor queries for 1,000 randomly generated points (i.e., about 12 milliseconds for each single nearest neighbor query on average). Besides the above finding, one can also observe that our proposed method is superior than the competitors, and this advantage is more obvious when ψ is larger. This further demonstrates the efficiency and competitiveness of our method. Additionally, it also demonstrates that the proposed method is much more suitable for higher dimensional data, compared against the competitors.

Effect of k. We studied the impact of k on the kNN query using real and synthetic datasets. Fixing $n = 50,000$, $\psi = 20$ for the synthetic dataset, and we varied k from 1 to 20 for both real and synthetic datasets. With the similar argument, we used corel as a representative of the real datasets. Figure 11 plots the experimental results. One can observe that the query cost is slightly increasing when larger k is used. This is mainly because kNN search algorithms maintain a priority queue that stores k nearest neighbors found so far, and a priority queue with more elements (i.e., a larger k)

usually takes more time to adjust the queue. Nevertheless, one can see that the growth speed of the query cost is very slow when k increases. This demonstrates that parameter k has little impact on the overall performance. This reflects, from another perspective, the effectiveness and feasibility of our strategy suggested in Section 7. On the other hand, one can clearly observe that our proposed method significantly outperforms the competitors, regardless of synthetic or real dataset, or the size of k. This further validates the competitiveness of our method.

Summary. We find that (i) CT++ is feasible and competitive, compared against its competitors CT and SNACT. Our method CT++ can reduce the query cost by 42.8%∼68% compared against CT. Particularly, even for the stronger baseline SNACT, our method outperforms it by 31.6%∼43.9%. (ii) The techniques suggested in this paper are effective. Our method CT++, which integrates all these techniques, achieves the maximum performance speedup (about 37.3%, 34.8%, 43.9%, and 31.6% on yearpredict, covertype, corel, and artificial40 datasets, respectively). (iii) For ANN and kNN queries, our method CT++ also exhibits excellent performance. Compared against CT, it achieves 39.3%∼66.1% (resp., 43%∼ 68.4%) performance speedup, while it outperforms SNACT by 28.3% ∼42.6% (resp., 32.3%∼44%), in terms of ANN (resp., kNN). (iv) The query cost goes up when n (resp., ψ) increases, and our method exhibits better scalability than the competitors. The query cost of CT++ (resp., CT) with n = 80K is 1.27∼1.63 (resp., 1.25∼ 1.8) times that of CT++ (resp., CT) with n = 1K. (v) As for kNN query, the parameter k has little impact on the overall performance. The query cost of CT++ with $k = 20$ is only 1.02 times that of CT++ with $k = 1$. (vi) The query cost goes down when ϵ increases, and CT++ obviously outperforms the competitors especially when ϵ is larger. Compared against CT and SNACT, on average it reduces the query cost by about 70.9% and 46.5% (resp., about 78.4% and 52.1%) on the synthetic (resp., real) dataset, respectively. (vii) Our method also suffers from some limitations such as more construction and update cost. Nevertheless, the overall construction and update performance is acceptable, and the performance gap between CT++ and SNACT is significantly smaller than the one between SNACT and CT, as shown in Appedix C.

10 Conclusion

Cover trees based solutions for nearest neighbor search have been shown many advantages, including strong theoretical guarantees and quick response efficiency in practice. Although these existing solutions achieved significant improvement for nearest neighbor search, some important and useful features are not fully exploited. Inspired by these, this paper has revisited cover tree structures for nearest neighbor search, and proposed a novel method called CT++. Our key propositions are threefold: (i) the range list defines the concept of multi-level distance information that were not fully exploited in prior works; (ii) the concept of quadrant information was developed; it defines the relation among query point, parent and child nodes; this idea is thoroughly novel, compared against the existing cover tree structures; (iii) the concept of vectorial angle cosine was introduced into our algorithm to further speed up the search performance. In addition, we have extended our algorithms to answer the variants of nearest neighbor search, including approximate nearest neighbor search and k nearest neighbor search. We have justified that our proposed algorithms maintain the same theoretical guarantees in terms of runtime, and have experimentally validated that the proposed algorithms made queries much faster in practice. We have also discussed the limitations of our method, and demonstrated that the overall performance of our method is still favourable, compared against the competitors.

Acknowledgements

This work was supported by the NSFC (No. 61972425). Zhe Quan was supported in part by the National Key R&D Program of China (No. 2018YFB0204302). Kaiqi Zhao was supported in part by the Marsden Fund Council from Government funding (MFP-UOA2123), administrated by the Royal Society of New Zealand. Bin Yao was supported in part by the NSFC (61922054, 61872235, 61832017, and 61729202), and the National Key R&D Program of China (2020YFB1710202, and 2018YFC1504504).

References

[1] A. Beygelzimer, S. Kakade, and J. Langford. Cover trees for nearest neighbor. In *ICML*, pages 97–104, 2006.
[2] D. Cai. A revisit of hashing algorithms for approximate nearest neighbor search. *IEEE Trans. Knowl. Data Eng.*, 33(6):2337–2348, 2021.
[3] G. Casanova, E. Englmeier, M. E. Houle, P. Kröger, M. Nett, E. Schubert, and A. Zimek. Dimensional testing for reverse k-nearest neighbor search. *PVLDB*, 10(7):769–780, 2017.
[4] X. Chen, W. Sun, B. Wang, Z. Li, X. Wang, and Y. Ye. Spectral clustering of customer transaction data with a two-level subspace weighting method. *IEEE Trans. Cybern.*, 49(9):3230–3241, 2019.
[5] Y. Chen, X. Hu, W. Fan, L. Shen, Z. Zhang, X. Liu, J. Du, H. Li, Y. Chen, and H. Li. Fast density peak clustering for large scale data based on knn. *Knowl. Based Syst.*, 187, 2020.
[6] C. Chiu, A. Prayoonwong, and Y. Liao. Learning to index for nearest neighbor search. *IEEE Trans. Pattern Anal. Mach. Intell.*, 42(8):1942–1956, 2020.
[7] K. L. Clarkson. Nearest neighbor queries in metric spaces. *Discrete & Computational Geometry*, 22(1):63–93, 1999.
[8] A. Davitkova, E. Milchevski, and S. Michel. The ml-index: A multidimensional, learned index for point, range, and nearest-neighbor queries. In *EDBT*, pages 407–410, 2020.
[9] M. Drosou and E. Pitoura. Multiple radii disc diversity: Result diversification based on dissimilarity and coverage. *ACM Trans. Database Syst.*, 40(1):4:1–4:43, 2015.
[10] J. H. Friedman, J. L. Bentley, and R. A. Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Trans. Math. Softw.*, 3(3):209–226, 1977.
[11] X. Ge and P. K. Chrysanthis. Efficient prefdiv algorithms for effective top-k result diversification. In *EDBT*, pages 335–346, 2020.
[12] M. Golbabaee, Z. Chen, Y. Wiaux, and M. E. Davies. Cover tree compressed sensing for fast mr fingerprint recovery. In *MLSP*, pages 1–6, 2017.
[13] A. G. Gray and A. W. Moore. 'n-body' problems in statistical learning. In *NIPS*, pages 521–527, 2000.
[14] Y. Gu, G. Liu, J. Qi, H. Xu, G. Yu, and R. Zhang. The moving K diversified nearest neighbor query. In *ICDE*, pages 31–32, 2017.
[15] J. He, S. Kumar, and S. Chang. On the difficulty of nearest neighbor search. In *ICML*, 2012.
[16] W. Hong, X. Tang, J. Meng, and J. Yuan. Asymmetric mapping quantization for nearest neighbor search. *IEEE Trans. Pattern Anal. Mach. Intell.*, 42(7):1783–1790, 2020.
[17] G. Hu, J. Shao, D. Zhang, Y. Yang, and H. T. Shen. Preserving-ignoring transformation based index for approximate k nearest neighbor search. In *ICDE*, pages 91–94, 2017.

[18] M. Izbicki and C. R. Shelton. Faster cover trees. In *ICML*, pages 1162–1170, 2015.

[19] D. R. Karger and M. Ruhl. Finding nearest neighbors in growth-restricted metrics. In *STOC*, pages 741–750, 2002.

[20] R. Krauthgamer and J. R. Lee. Navigating nets: simple algorithms for proximity search. In *SODA*, pages 798–807, 2004.

[21] S. Li. An improved DBSCAN algorithm based on the neighbor similarity and fast nearest neighbor query. *IEEE Access*, 8:47468–47476, 2020.

[22] W. Li, Y. Zhang, Y. Sun, W. Wang, M. Li, W. Zhang, and X. Lin. Approximate nearest neighbor search on high dimensional data - experiments, analyses, and improvement. *IEEE Trans. Knowl. Data Eng.*, 32(8):1475–1488, 2020.

[23] S. Lisitsyn, C. Widmer, and F. J. I. Garcia. Tapkee: an efficient dimension reduction library. *JMLR*, 14(1):2355–2359, 2013.

[24] J. Liu, Z. Dou, X. Wang, S. Lu, and J. Wen. DVGAN: A minimax game for search result diversification combining explicit and implicit features. In J. Huang, Y. Chang, X. Cheng, J. Kamps, V. Murdock, J. Wen, and Y. Liu, editors, *SIGIR*, pages 479–488. ACM, 2020.

[25] Y. A. Malkov and D. A. Yashunin. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE Trans. Pattern Anal. Mach. Intell.*, 42(4):824–836, 2020.

[26] N. Monath, A. Kobren, A. Krishnamurthy, M. R. Glass, and A. McCallum. Scalable hierarchical clustering with tree grafting. In A. Teredesai, V. Kumar, Y. Li, R. Rosales, E. Terzi, and G. Karypis, editors, *KDD*, pages 1438–1448, 2019.

[27] L. Prokhorenkova and A. Shekhovtsov. Graph-based nearest neighbor search: From practice to theory. In *Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13-18 July 2020, Virtual Event*, volume 119 of *Proceedings of Machine Learning Research*, pages 7803–7813. PMLR, 2020.

[28] N. Segata and E. Blanzieri. Fast and scalable local kernel machines. *JMLR*, 11:1883–1926, 2010.

[29] F. Shen, Y. Yang, L. Liu, W. Liu, D. Tao, and H. T. Shen. Asymmetric binary coding for image search. *IEEE Trans. Multimedia*, 19(9):2022–2032, 2017.

[30] J. Song, T. He, L. Gao, X. Xu, A. Hanjalic, and H. T. Shen. Binary generative adversarial networks for image retrieval. In *AAAI*, pages 394–401, 2018.

[31] Y. Tagami. Annexml: Approximate nearest neighbor search for extreme multi-label classification. In *SIGKDD*, pages 455–464, 2017.

[32] B. Tang, M. L. Yiu, and K. A. Hua. Exploit every bit: Effective caching for high-dimensional nearest neighbor search. In *ICDE*, pages 45–46, 2017.

[33] N. Tziortziotis, C. Dimitrakakis, and K. Blekas. Cover tree bayesian reinforcement learning. *JMLR*, 15(1):2313–2335, 2014.

[34] J. K. Uhlmann. Satisfying general proximity/similarity queries with metric trees. *Inf. Process. Lett.*, 40(4):175–179, 1991.

[35] J. Wang, T. Zhang, J. Song, N. Sebe, and H. T. Shen. A survey on learning to hash. *IEEE Trans. Pattern Anal. Mach. Intell.*, 40(4):769–790, 2018.

[36] B. Yao, Z. Chen, X. Gao, S. Shang, S. Ma, and M. Guo. Flexible aggregate nearest neighbor queries in road networks. In *ICDE*, pages 761–772, 2018.

[37] M. Zaheer, S. Kottur, A. Ahmed, J. M. F. Moura, and A. J. Smola. Canopy fast sampling with cover trees. In *ICML*, pages 3977–3986, 2017.

[38] W. Zhao, S. Tan, and P. Li. SONG: approximate nearest neighbor search on GPU. In *36th IEEE International Conference on Data Engineering, ICDE 2020, Dallas, TX, USA, April 20-24, 2020*, pages 1033–1044. IEEE, 2020.

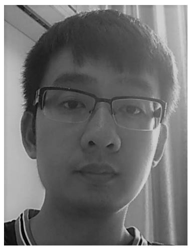
[39] Y. Zheng, Q. Guo, A. K. H. Tung, and S. Wu. Lazylsh: Approximate nearest neighbor search for multiple distance functions with a single index. In *SIGMOD*, pages 2023–2037, 2016.

[40] J. Zhou, E. Agichtein, and S. Kallumadi. Diversifying multi-aspect search results using simpson's diversity index. In M. d'Aquin, S. Dietze, C. Hauff, E. Curry, and P. Cudré-Mauroux, editors, *CIKM*, pages 2345–2348. ACM, 2020.

Zhi-Jie Wang received the PhD degree in computer science from the Shanghai Jiao Tong University, Shanghai, China. He is currently an associate professor at the College of Computer Science, Chongqing University (CQU), Chongqing, China. Prior to that, he worked with the Sun Yat-sen University, Guangzhou, China. His current research interests include data mining, artificial intelligence, big data. He is a member of CCF, IEEE, and ACM.

Mengdie Nie received the Bachelor degree from the South China University of Technology, Guangzhou, China, and the Master degree from the Sun Yat-sen University, Guangzhou, China, resepectively. Since 2020, she worked as a software engineer in Pingduoduo Inc, Shanghai, China. Her research interests include data mining, algorithm design & analysis, query processing, etc.

Kaiqi Zhao received the Ph.D from the School of Computer Science and Engineering, Nanyang Technological University, Singapore. He is currently a senior Lecturer at the School of Computer Science, the University of Auckland, Auckland, New Zealand. Prior to that, he worked as a Research Fellow at the Nanyang Technological University, Singapore. His current research interests include mining geographical social media, knowledge discovery, and machine learning.



Zhe Quan received the PhD degree in computer science from the University de Picardie Jules Verne, France. He is currently an associate professor at the College of Computer Science and Technology, Hunan University (HNU), Changsha, China. Before joining HNU, he worked at the National University of Defense Technology, Changsha, China. His main research interests include machine learning, artificial intelligence, parallel and high-performance computing, etc.

Bin Yao received the PhD degree in computer science from the Florida State University, the United States. He is currently a Professor with the Department of Computer Science and Engineering, Shanghai Jiao Tong University, Shanghai, China. His research interests include mananagement of indexing of large databases, query processing in spatial and multimedia databases, string and keyword search, and scalable data analytics.

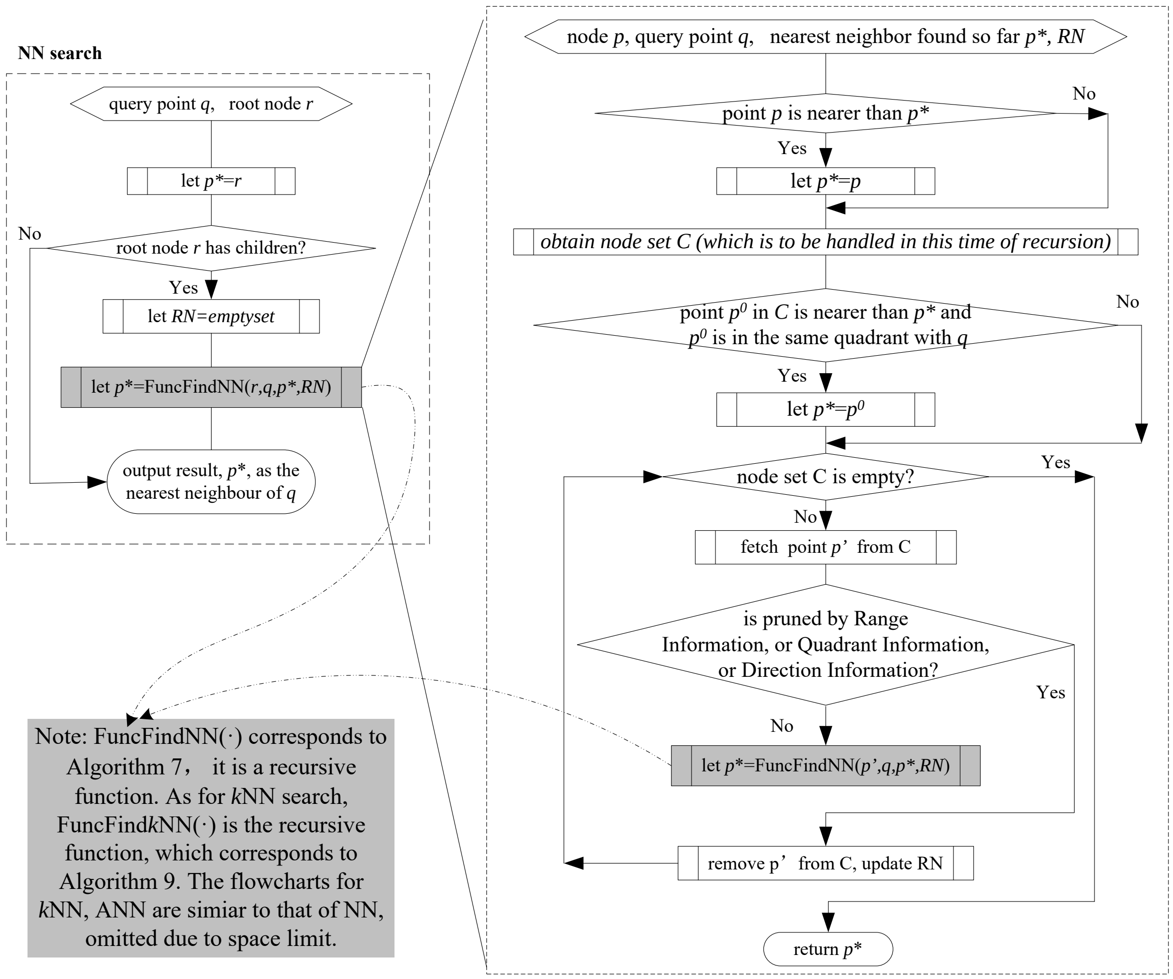
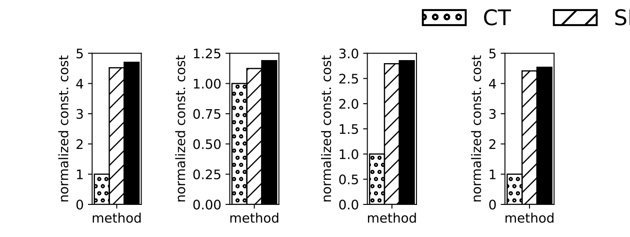
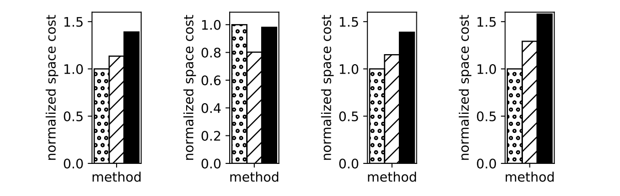

NN search
query point q, root node r
let p*=r
root node r has children?
No
Yes
let RN=emptyset
let p*=FuncFindNN(r,q,p*,RN)
output result, p*, as the nearest neighbour of q
Note: FuncFindNN(·) corresponds to Algorithm 7, it is a recursive function. As for kNN search, FuncFindkNN(·) is the recursive function, which corresponds to Algorithm 9. The flowcharts for kNN, ANN are simiar to that of NN, omitted due to space limit.
node p, query point q, nearest neighbor found so far p*, RN
point p is nearer than p*
No
Yes
let p*=p
obtain node set C (which is to be handled in this time of recursion)
point p0 in C is nearer than p* and p0 is in the same quadrant with q
No
Yes
let p*=p0
node set C is empty?
Yes
No
fetch point p' from C
is pruned by Range Information, or Quadrant Information, or Direction Information?
Yes
No
let p*=FuncFindNN(p',q,p*,RN)
remove p' from C, update RN
return p*

Fig. A1: A detailed flowchart of our NN search.



(a) yearpredict (b) covertype (c) corel (d) artificial40

Fig. A2: Consturction time cost comparison.



(a) yearpredict (b) covertype (c) corel (d) artificial40

Fig. A3: Space cost comparison.

Appendix

A. The Flowchart of Our Main Algorithm.

For ease of understanding our algorithms, Figure A1 shows the flowchart of our algoirithm.

B. Complexity Results of CT, SNACT, and CT++.

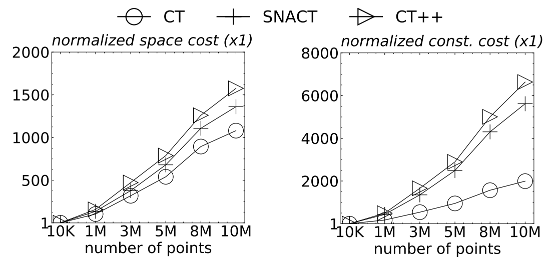
Table A1 compares the compelexity results of CT, SNACT, and CT++. Our results are the same as that in SNACT.

methodology	space	construction	query
CT	$O(n)$	$c^6 n \log n$	$c^{12} \log n$
SNACT	$O(n)$	$c^{12} n \log n$	$c^6 \log n$
CT++	$O(n)$	$c^{12} n \log n$	$c^6 \log n$

TABLE A1: Complexity related to space, construction, and query; where c is the expansion constant, and n is the number of objects. For SNACT and/or CT++, the number of nodes in the data structure is exactly n.

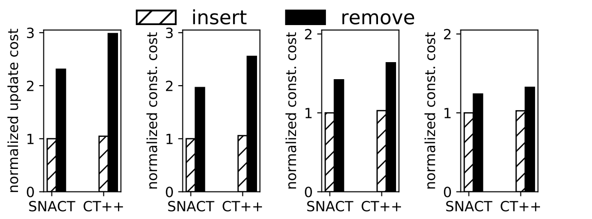
C. Other Results and Discussion

Although we have witnessed many advantages of our method, it is necessary to mention the limitations of our method. Overall, the limitations are mainly arisen from the indexing part. To understand, Figure A2 plots the construct time cost on these benchmark datasets. It can be seen that, (i) among these methods CT consumes less construction time, while SNACT and CT++ consume more time to construct the corresponding data structures (e.g., on the core dataset we used only 129.732 seconds to construct CT, we instead used 362.335 and 370.114 seconds to construct SNACT and



(a) space (b) construction

Fig. A4: Space and construction cost vs. data volume n.



(a) yearpredict (b) covertype (c) corel (d) artificial40

Fig. A5: Update cost comparison.

CT++, respectively); and (ii) the performance gap between CT and SNACT is large on most of these datasets, while the gap between SNACT and CT++ is small on almost all these datasets. As for the above phenomena, it is mainly because SNACT and CT++ need to readjust (i.e., rebalance) the trees in order to maintain the nearest ancestor invariant. Naturally, they use more time to construct corresponding trees. The large performance gaps on some datasets (e.g., yearpredict, corel, artifical40) imply that there are much more nodes that may violate the nearest ancestor invariant, and so more cost is used to readjust the trees in the construction phase.

The readers could be curious why the gap between CT and SNACT is relatively small on the covertype dataset, as shown in Figure A2(b). In fact, this is mainly because the number of nodes in SNACT is only about 60% of the number of nodes in CT, and so this reduces the performance gap (although SNACT needs to readjust the tree), while the number of nodes in SNACT is close to that in CT on other several datasets. For example, on the covertype dataset there are about 970K and 580K nodes in CT and SNACT, respectively. In contrast, on the corel dataset there are about 78K and 68K nodes in CT and SNACT, respectively. In fact, Figure A3 also reflects similar findings. That is, (i) on the covertype dataset, SNACT uses less space than CT (17.8 MB vs. 22.2 MB); and (ii) on most of datasets SNACT and CT++ use more space than CT (e.g., on the corel dataset, CT uses about 1.79 MB, while SNACT and CT++ consume 2.04 MB and 2.32 MB, respectively). As for (ii), this is mainly because SNACT needs to store distance information $maxdist(\cdot)$ for each non-leaf node and CT++ needs to store some extra distance and quadrant information. Besides the results on these benchmark datasets, we also compare the trend of space (resp., construction) cost over different data volume. On one hand, from Figure A4(a) we can easily see that the space cost increases when the data volume enlarges. This is because the data structures need to store more nodes when the data size increases. On the other hand, we can find the similar trend about the construction cost, as shown in Figure A4(b). This is mainly because more nodes need to be inserted into the data structures.

Additionally, Figure A5 plots the update performance comparison on these benchmark datasets. It can be seen that, (i) CT++ consumes more time than SNACT for both insertion and removal operations; and (ii) removal operations are much more time-consuming than insertion operations. As for (i), the main reason could be that CT++ needs to maintain additional properties such as quadrant information, and so more update cost is used when inserting or removing nodes. As for (ii) the main reason could be that removal operations need to shrink the subtrees "level by level" when some nodes cannot be attached to any sibling node of q. Nevertheless, the overall update performance is acceptable. For example, on the corel dataset, our method CT++ used only 5.35 and 8.51 seconds to perform insertions and removals of 1,000 points, respectively.

Remark that, similar to SNACT, our method sacrifices some performance (e.g., construction and update) to achieve stronger search performance, which is the central goal of our paper. Besides the efficiency of the proposed method, the completeness of the results is also important. One could argue that, it is possible that the duplicated data points may exist in some datasets. As for these cases, an immediate solution, similar to other data structructs such as binarty tree, is to use an interger variable, say *count*, to remember the number of duplicate points, instead of creating multiple entries for them in the tree.

Furthermore, some readers may realize that our method relies on well-defined, totally ordered coordinates in the dimensions to construct the quadrants, it would be hard to index unstructured data such as text, trajectories, etc., where a distance measure can be defined (e.g., Hausdroff distance). This could be viewed as a limitation or a new challenge needing to be further researched (we leave this as the futuer work). Nevertheless, it could be interesting to compare it with some spatial data structures (e.g., k-d tree, Quad-tree, R-tree). To this end, we conduct a set of experiments to investigate them. Specifically, we generate randomly a set of two-dimensional points, and then use these algorithms to perform nearest neighbor queries, respectively. Figure A6 shows the comparison results. It can be seen that, R-tree and Quad-tree perform better when the data size is small. With the increase of the data size, CT++ and k-d tree exihibit better performance, and k-d tree performs always better than CT++. Nonetheless, k-d tree is mainly used in the scenario when the Euclidian dimension is low, as mentioned in [15,16]. Moreover, we would like to mention that R-tree and its variants (called also R-tree family) could be still the most favorably choice for spatial data management, due to the excellent characteristics including dynamically update, handling various geometries, etc.

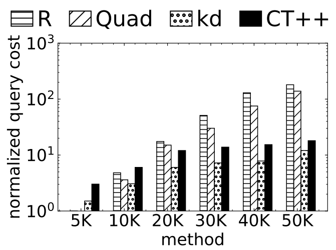


Fig. A6: A comparison with other data structures.