The Moving $K$ Diversified Nearest Neighbor Query

Yu Gu*, Guanli Liu, Jianzhong Qi, Hongfei Xu, Ge Yu, Rui Zhang

Abstract—As a major type of continuous spatial queries, the moving $k$ nearest neighbor ($k$NN) query has been studied extensively. However, most existing studies have focused on only the query efficiency. In this paper, we consider further the usability of the query results, in particular the diversification of the returned data points. We thereby formulate a new type of queries named the moving $k$ diversified nearest neighbor query ($Mk$DNN). This type of queries continuously report the $k$ diversified nearest neighbors while the query object is moving. Here the degree of diversity of the $k$NN set is defined on the distance between the objects in the $k$NN set. Computing the $k$ diversified nearest neighbors is an NP-hard problem. We propose an algorithm to maintain incrementally the $k$ diversified nearest neighbors to reduce the query processing costs. We further propose two approximate algorithms to obtain even higher query efficiency with precision bounds. We verify the effectiveness and efficiency of the proposed algorithms both theoretically and empirically. The results confirm the superiority of the proposed algorithms over the baseline algorithm.

Index Terms—Moving nearest neighbor query, spatial diversity, safe region, approximate algorithm

1 INTRODUCTION

As a major type of moving queries, the moving $k$ nearest neighbor ($Mk$NN) query has been studied extensively [1], [2], [3], [4]. This query assumes a moving query object $q$ and a set of static data objects $O$. When $q$ is moving, the $Mk$NN query reports its $k$ nearest neighbors ($k$NN) continuously. Common applications of this query include finding nearest petrol stations continuously while one drives on highway, or nearest points of interest (POI) continuously while a tourist is walking around the city [4]. However, $Mk$NN queries tend to return $k$NNs that cluster together, i.e., data objects near $q$ tend to be near each other as well. These clustering $k$NNs may share common features, e.g., all in a less pleasant suburb, which make them all unsatisfactory query answers. To avoid clustering $k$NNs and to improve result usability, query result diversification is a popular technique. Originated from information retrieval [5], [6], [7], this technique tries to return results as different from each other as possible while all satisfying the query predicate.

Fig. 1 illustrates how query result diversification may improve the result of an $Mk$NN query. The query user $q$ is looking for a restaurant for dining as she is on a road trip. There are 9 restaurants $p_1, p_2, ..., p_9$ to be chosen from. An $Mk$NN query will return $\{p_1, p_2, p_3\}$ as the answer since they are the nearest to $q$. However, they are all in a busy area. While it is convenient to shop or to get a cup of coffee there, the traffic is congested and parking is difficult. Alternatively, a query result diversification technique may take the distance between the restaurants themselves into consideration, and return $\{p_2, p_4, p_7\}$ as the answer. This answer is more preferable. The three restaurants are all relatively close to the query user. Meanwhile, they are far away from each other and are distributed in different areas. The query user can choose the one in her preferred area, e.g., $p_4$, which is in a nice garden area. The example can also be thought of as a scenario of tourist hotspot recommendation, where the small squares in the figure represent the tourist hotspots. The $Mk$NN query may return tourist hotspots clustering together, which the query user may have all visited already. In this case, the user may prefer recommendations that are far away from each other. To find such query answers, we will need a new type of queries that consider the diversity of the query results.

Efforts [8], [9], [10] have been made to diversify the results for static $k$NN queries. However, there is still a lack of studies to diversify the results for moving queries. We aim to fill this gap by proposing a new type of queries called the moving $k$ diversified nearest neighbor ($Mk$DNN) query. As the query object $q$ is moving, this query continuously returns $k$ objects that are the nearest to $q$ and are as diversified as possible. Here, “nearest” is defined on spatial proximity, while “diversified” is defined on the difference between the nearest neighbors, which can have different meanings if viewed from different perspectives. In this study, we define diversity as the distance between the nearest neighbors, al-
though we can also use other types of attributes for diversification (e.g., type of cuisine in the restaurant example). The MkDNN query is then to find nearest neighbors that are as far away from each other as possible. This query definition will help avoid clustering $k$NNs and find diversified nearest neighbors as illustrated in Fig. 1.

The safe region technique [3], [4], [11], [12] is commonly used in processing moving queries. It first treats the query as a static one and computes the query answer together with a “safe region”. As long as the query point is still in this safe region, it is guaranteed that the current query answer is still valid. The query processing then becomes continuously checking whether the query point is still in the safe region. Only when the query point moves out of the safe region that the static query is computed again and the query answer as well as the safe region are updated. We follow this procedure to process the MkDNN query. Every time the current query answer becomes invalid, we recompute a static $k$ diversified nearest neighbor (kDNN) query.

As processing a static $k$DNN query has been shown to be NP-hard [13], our initial focus is to reduce the frequency of recomputing it as much as possible. Existing studies on moving queries have mostly used Voronoi diagram based safe regions to reduce the recomputation frequency. However, these safe regions are defined only on spatial proximity. They cannot handle diversity. We overcome this limitation by prefetching top-$m$ $k$DNN sets instead of only the top-1 $k$DNN set when the $k$DNN query is recomputed, where $m$ is a system parameter. Intuitively, nearby points should have similar $k$DNN sets. The prefetched $k$DNN sets will serve as a “cache”. When the query point moves, we first try to find the new $k$DNN set from this “cache”. Only when no prefetched set is valid that a query recomputation is needed. This reduces the recomputation frequency.

To further reduce the recomputation frequency, we propose an approximate algorithm that computes only the top-1 $k$DNN set, and keeps it as the query answer as long as it is a $\rho$-approximation of the true query answer, where $\rho$ is user defined parameter. We derive safe regions based on spatial proximity as well as diversity to efficiently determine validity of the current $k$DNN set.

In addition, we adapt a greedy algorithm to compute approximate $k$DNN sets at recomputation, and propose a strategy to maintain the approximate query answer with a precision bound. This strategy also uses prefetching. We design safe regions that can be computed efficiently to guard the validity of the prefetched candidate query answers. The resultant algorithm can reduce the cost of recomputation and the recomputation frequency at the same time, and hence achieves even higher efficiency.

We make the following contributions in this paper:

- We propose a new query type, the moving $k$ diversified nearest neighbor query (MkDNN), which ads diversity to the nearest neighbors.

- Since the static $k$DNN query is NP-hard, we approach the MkDNN query first by reducing the frequency of recomputing the $k$DNN queries when the query point moves. We propose a prefetching-based precise algorithm as well as a bounded approximate algorithm to achieve this goal.

- We further propose an algorithm that computes approximate $k$DNN query answers, and hence further reduces the overall processing costs.

- We analyze the costs of the proposed algorithms and evaluate the algorithm performance on real data sets. The result confirms the effectiveness and efficiency of the proposed algorithms.

The rest of the paper is organized as follows. We discuss related studies in Section 2. We present the problem setting in Section 3. We propose three MkDNN query algorithms in Sections 4 and 5. We study the performance of the proposed algorithms in Section 6 and conclude the paper in Section 7.

2 RELATED WORK

2.1 Moving $k$ nearest neighbor queries

MkNN queries have been studied extensively [2], [3], [14], [15], [16]. Early studies (e.g., [14]) have used sampling based approaches. They consider the query object’s trajectory as formed by discrete points, and process a static $k$NN query at each point. Between the discrete points, the query answer is inaccurate. To obtain more accurate query answer, two adjacent discrete points have to be very close, and the $k$NN query has to be computed frequently, which is less efficient.

Later studies [3], [16] use safe regions to reduce the query recomputation frequency. The safe region is a region where the query object can move freely without causing the current $k$NN answer to change. This allows only recomputing the $k$NN answer when the query object leaves the current safe region. Voronoi diagrams are commonly used to construct safe regions. In an order-$k$ Voronoi diagram [17], each cell is a safe region. As long as the query object is in an order-$k$ Voronoi cell, the $k$ data points forming this cell are the $k$NN answer. By definition, the order-$k$ Voronoi cell is the largest possible safe region. However, this cell is expensive to compute. The Retrieve-Influence-Set knn (RIS-kNN) algorithm [16] computes an order-$k$ Voronoi cell by a few (six on average) time-parameterized $k$NN queries (which are still quite expensive) when the $k$NN answer becomes invalid. The $V^*\text{-Diagram}$ approach [3] sacrifices the size of the safe region for higher efficiency of safe region computation. It approximates an order-$k$ Voronoi cell with the integrated safe region, which is more efficient to compute but smaller, and hence the query recomputation frequency is higher. The influential neighbor set approach [4] is the state-of-the-art MkNN algorithm. This approach uses safe guarding objects rather than safe regions, which is a small set of data points surrounding the current $k$NN set. Conceptually, the safe guarding objects still define a safe region, which has been shown to be equivalent to an order-$k$ Voronoi cell. Meanwhile, the safe guarding objects are much more efficient to compute. Therefore, the influential neighbor set approach achieves high overall query efficiency.

Other MkNN query studies (e.g., [15]) assume predefined linear query trajectories. The safe regions are then reduced to line segments on the trajectories. These studies have focused on the efficiency aspect of query processing. They cannot compute $k$ diversified nearest neighbors and hence are not applicable to our MkDNN queries.
2.2 Query result diversification

Originated from information retrieval, studies [5], [6], [7], [18] on query result diversification try to return results as different from each other as possible while all satisfying the query predicate, to improve query result usability.

Diversification has been considered static spatial queries such as the kNN query. For example, Lee et al. [8] find the nearest objects surrounding a static query point. Kreveld et al. [19] consider diversification in ranking spatial objects. They rank all objects in the database for every query, which has high costs. Zhang et al. [9] study the problem of diversification spatial keyword search on road networks. Their query predicate, to improve query result usability.

Fig. 2. A 3DNN query ($\lambda = 0.5$)

In these two functions, $\text{dis}(p_i, q)$ and $\text{div}(p_i, p_j)$ compute the distance and the degree of diversity between two points. We use $\text{dis}_m$ and $\text{div}_m$ to denote the maximum distance and the maximum degree of diversity, respectively. These two maximum values help normalize the two function values into the range $[0, 1]$. We compute the average $(\frac{\text{dis} - \text{dis}_m}{\text{dis}_m})$ of these function values to convert them into a similar scale.

The full function $f_q(S)$ can be written as follows.

$$f_q(S) = \frac{\lambda}{k} \sum_{p_i \in S} \left(1 - \frac{\text{dis}(p_i, q)}{\text{dis}_m}\right) + \frac{2(1 - \lambda)}{k(k - 1)} \sum_{p_i, p_j \in S} \frac{\text{div}(p_i, p_j)}{\text{div}_m}$$

(1)

In this paper we focus on spatial diversity. We define the degree of diversity between two data points $p_i$ and $p_j$ as the spatial distance between $p_i$ and $p_j$, i.e.,

$$\text{div}(p_i, p_j) = \text{dis}(p_i, p_j).$$

The function $\text{dis}(p_i, p_j)$ returns the Euclidean distance between $p_i$ and $p_j$. As a result, we have $\text{div}_m = \text{dis}_m.$ Equation 1 can then be rewritten as follows.

$$f_q(S) = \frac{\lambda}{k} \sum_{p_i \in S} \left(1 - \frac{\text{dis}(q,p_i)}{\text{dis}_m}\right) + \frac{2(1 - \lambda)}{k(k - 1) \text{dis}_m} \text{dis}(p_i, p_j)$$

(2)

Now we can define the $k$ diversified nearest neighbor query and the moving $k$ diversified nearest neighbor query.

Definition 1 (K diversified nearest neighbor (kDNN) query).

Given a set of static data objects $P$, a query point $q$, a query parameter $k$, and a user preference parameter $\lambda$, the $k$ diversified nearest neighbor query returns a size-$k$ set $S \subseteq P$, for any size-$k$ set $S' \subseteq P$, $f_q(S) \geq f_q(S')$.

Fig. 2 is an example, where $\text{dis}(q, p_1) = 30, \text{dis}(q, p_2) = 40, \text{dis}(q, p_3) = 50, \text{dis}(q, p_4) = 60, \text{dis}(q, p_5) = 70, \text{dis}(q, p_6) = 80$. Assuming $k = 3$ and $\lambda = 0.5$, then the kDNN set of $q$ should be $\{p_3, p_4, p_5\}$ instead of any other subsets of $P$. For example, $f_q(\{p_3, p_4, p_5\}) = 0.6438 > f_q(\{p_1, p_2, p_3\}) = 0.5822$. Intuitively, even though $\{p_1, p_2, p_3\}$ is slightly nearer to $q$, $\{p_3, p_4, p_5\}$ is more diversified (the points are farther away from each other). Overall, $\{p_3, p_4, p_5\}$ has a higher optimization function value.

When the query point $q$ is moving, i.e., $q$ may be at different locations at different timestamps, the query becomes a moving $k$ diversified nearest neighbor query.

Definition 2 (Moving k diversified nearest neighbor (MkDNN) query). Given a set of static data objects $P$, a moving query point $q$, a query parameter $k$, and a user preference parameter $\lambda$, the moving $k$ diversified
nearest neighbor query returns the kDNN set of q for every timestamp.

Following existing studies on moving kNN queries (e.g., [3], [4], [16]), we consider the general scenario where there is no constraint on the moving pattern of the query point q, i.e., q can move towards any direction with any speed at any time. When an MkDNN query is issued, we first compute a static kDNN query at q, and then continuously validate the kDNN set as q is moving. We recompute the kDNN set when it becomes invalid. We present both precise and approximate algorithms for kDNN computation and validation in the following sections.

4 MkDNN Algorithms Based on Precise kDNN Computation

Computing the kDNN query is essentially to solve the Maximum Dispersion problem, which is NP-hard [13]. In this section we assume that there exists an algorithm1 to compute the kDNN query, and focus on optimizing the kDNN validation phrase. The goal is to reduce the frequency of recomputing the kDNN to reduce the overall query costs.

4.1 A Precise Algorithm for kDNN Maintenance

We first reduce the kDNN computation frequency by prefetching the top-m kDNN sets, where m is a system parameter and will be chosen empirically. Here, the intuition is that two nearby locations l1 and l2 should share similar kDNN sets. It is likely that the top-1 kDNN set of l1 is also among the top m kDNN sets of l2. As such, the prefetched the top-m kDNN sets would serve as a “cache” for the MkDNN query. Only when none of the prefetched kDNN sets is still valid that the kDNN sets need to be recomputed. Note that if the enumeration based algorithm is used to compute the kDNN set, then the prefetching requires maintaining a size-m priority queue, although this would have very little overhead when m is small.

We use a priority queue denoted by Q to store the m kDNN sets prioritized by their \( f_q() \) function values. When the query point q moves, we first try to find the new kDNN set from Q. If found, then we return it, and update Q according to the new \( f_q() \) function values of the kDNN sets. Otherwise, we compute the new top-m kDNN sets, store them in Q, and return the top one as the query answer.

Next we detail how to maintain Q and compute the new kDNN set from it.

4.1.1 Incremental kDNN Maintenance

Safe radius. We introduce the safe radius to help maintain Q. This concept defines a region to guarantee that a size-k set \( S_i \subseteq P \) is “nearer” to q than another size-k set \( S_j \subseteq P \).

Definition 3 (Safe radius). Let \( S_i, S_j \) be two size-k subsets of \( P \) satisfying \( f_q(S_i) > f_q(S_j) \). The safe radius for \( S_i \) and \( S_j \), denoted by \( r_{i,j} \), is defined to guarantee \( f_q(S_i) > f_q(S_j) \) for the query point to move from \( q \) to any new location \( q' \) as long as \( dis(q, q') < r_{i,j} \).

Fig. 3 shows an example of the priority queue Q and the safe radius, where \( m = 6 \). We denote the kDNN sets in Q by \( S_1, S_2, ..., S_m \), according to their ranks in Q. The kDNN set \( S_1 = \{p_3, p_4, p_5\} \) has the largest \( f_q() \) function value, with

\[
S_i = \{p_3, p_4, p_5\} \\
S_j = \{p_1, p_4, p_5\} \\
S_k = \{p_2, p_5, p_6\} \\
S_l = \{p_3, p_6, p_7\} \\
S_m = \{p_4, p_6, p_8\} \]

where \( r_s = \frac{2.34}{1.16} > \frac{1.88}{0.45} = r_s \) for some priority queue Q.

We derive a mathematical definition for \( r_{i,j} \). The goal is to find \( r_{i,j} \) such that \( f_q(S_i) > f_q(S_j) \), where

\[
f_q(S_i) = \sum_{p_x,p_y \in S_i} \left( \lambda(2 - \frac{\text{dis}(q,p_x) + \text{dis}(q,p_y)}{\text{dis}_m}) + \frac{2(1-\lambda)}{\text{dis}_m} \text{dis}(p_x,p_y) \right) \\
f_q(S_j) = \sum_{p_x,p_y \in S_j} \left( \lambda(2 - \frac{\text{dis}(q,p_x) + \text{dis}(q,p_y)}{\text{dis}_m}) + \frac{2(1-\lambda)}{\text{dis}_m} \text{dis}(p_x,p_y) \right)
\]

According to triangle inequality, \( \forall p \in P: \)

\[
\text{dis}(q,p) + \text{dis}(q,q') \geq \text{dis}(q',p) \\
\text{dis}(q,p) - \text{dis}(q,q') \leq \text{dis}(q',p).
\]

We replace \( \text{dis}(q',p_x) \) and \( \text{dis}(q',p_y) \) in \( f_q(S_i) \) by \( \text{dis}(q,p_x) + \text{dis}(q,q') \) and \( \text{dis}(q,p_y) + \text{dis}(q,q') \), respectively:

\[
f_q(S_i) \geq \sum_{p_x,p_y \in S_i} \frac{2(1-\lambda)}{\text{dis}_m} \text{dis}(p_x,p_y) \frac{k(k-1)}{k-1} = f_q(S_i) - \frac{\lambda \text{dis}(q,q')}{\text{dis}_m}
\]

Similarly, we replace \( \text{dis}(q',p_x) \) and \( \text{dis}(q',p_y) \) in \( f_q(S_j) \) by \( \text{dis}(q,p_x) + \text{dis}(q,q') \) and \( \text{dis}(q,p_y) + \text{dis}(q,q') \):

\[
f_q(S_j) \leq \sum_{p_x,p_y \in S_j} \frac{2(1-\lambda)}{\text{dis}_m} \text{dis}(p_x,p_y) \frac{k(k-1)}{k-1} = f_q(S_j) + \frac{\lambda \text{dis}(q,q')}{\text{dis}_m}
\]

To let \( f_q(S_i) > f_q(S_j) \), we need:

\[
f_q(S_i) - \frac{\lambda \text{dis}(q,q')}{\text{dis}_m} > f_q(S_j) + \frac{\lambda \text{dis}(q,q')}{\text{dis}_m} \\
\Rightarrow \frac{f_q(S_i) - f_q(S_j)}{\text{dis}_m} > 2\lambda \text{dis}(q,q') \\
\Rightarrow \frac{\text{dis}(q,q')}{\frac{f_q(S_i) - f_q(S_j)}{\text{dis}_m}} < \frac{2\lambda}{\text{dis}_m}
\]

1. A straightforward enumeration based algorithm can enumerate all \( C_n^k \) combinations of the n data points to find the optimal kDNN set with \( O(n^k) \) time.
and hence it the top item in $Q$. We compute the safe radius $r_{1, m}, r_{2, m}, ..., r_{m-1, m}$ between the last item in the queue, $S_m$, and each of the other sets $S_1, S_2, ..., S_{m-1}$, respectively.

**Safe regions.** The safe radius helps define two circular safe regions centered at $q$. The first safe region has a radius of $r_{1, 2}$ (cf. the smaller circle centered at $q$ in Fig. 4), which guarantees that the kDNN set stays unchanged as long as the query point stays in this region. We call this region the **answer valid region** (AVR).

The second safe region guarantees that the new kDNN set is still among the $m$ kDNN sets in $Q$, which means that $Q$ is still valid. We call this region the **queue valid region** (QVR). As shown by the larger circle centered at $q$ in Fig. 4, the radius of QVR is $r_{1, m}$. Assume that the query point has moved to $q'$. Then as long as $\text{dis}(q, q') < r_{1, m}$, by the definition of the safe radius, $f_q(S_1) > f_q(S_i)$ for any $S_i \notin Q$. This means that any $S_i \notin Q$ cannot be the top kDNN set. Thus, the top kDNN set must still be in $Q$.

**Query updates.** Next we derive two properties of the safe radius as specified by the following two theorems. They support our query update algorithm.

**Theorem 1.** Assume that all size-$k$ subsets of $P$ are sorted in the descending order of their $f_q(\cdot)$ function values, denoted by $S_1, S_2, ..., S_m, ..., S_{k}$. Then, given three integers $i, j, k$, and $pt$:

1. If $1 \leq i < j < pt \leq C^k_m$, then $r_{i, pt} > r_{j, pt}$.
2. If $1 \leq pt < i < j \leq C^k_m$, then $r_{ip, t} < r_{jp, t}$.

**Proof 1.** See Appendix A.

Suppose that we have computed the kDNN sets and obtained the queue $Q$ at $q$. When the query point moves to $q'$, we can compare $\text{dis}(q', q)$ with the safe radius of all the kDNN sets in $Q$. For a kDNN set $S_j$, if its safe radius value $r_{j, m}(1 \leq j \leq m)$ satisfies $r_{j, m} > \text{dis}(q', q)$, then $f_q(S_j) > f_q(S_m)$ still holds and $S_j$ should still be a top-$m$ kDNN set. We put all the kDNN sets satisfying this condition into a new queue $Q'$ prioritized by their $f_q(\cdot)$ function values. We denote the last entry in $Q'$ by $S_{|Q'|}$.

For each remaining kDNN set $S_x \in Q$, we compute $f_q(S_x)$, and compare it with $f_q(S_{|Q'|})$. If $f_q(S_x) > f_q(S_{|Q'|})$, we also insert $S_x$ into $Q'$. Otherwise we simply omit $S_x$ since we can no longer guarantee it to be a top kDNN set.

The queue $Q'$ will then become the new $Q$ in the query maintenance, with the top entry $S'_1$ being the new top kDNN set at $q'$, as guaranteed by the following theorem.

**Theorem 2.** Let $S'_{|Q'|}$ be the last entry in $Q'$, i.e., $f_q(S'_{|Q'|})$ is the smallest among the entries in $Q'$. Then $\forall S \notin Q'$ that is a size-$k$ subset of $P$: $f_q(S'_1) > f_q(S)$.

**Proof 2.** See Appendix A.

According to this theorem, when the query point moves, we can keep updating $Q$ to $Q'$ and use $Q'$ as the new $Q$. Note that every time $Q$ is updated, we might remove some entries from it, and hence $Q$ will shrink. When $Q$ becomes empty, we need to recompute the top-$m$ kDNN sets.

Fig. 5 shows an example. When the query point moves to $q', r_{1, 6, r_{2, 6}}$, and $r_{3, 6}$ are greater than $\text{dis}(q', q)$. We insert $S_1, S_2$, and $S_3$ into a new queue $Q'$, prioritized by $f_q(S_i), i = 1, 2, 3$. We then compare $f_q(S_x), x = 4, 5, 6$ with $f_q(S'_1)$, and only keep the ones with $f_q(S_x) > f_q(S'_1)$.

### 4.1.2 The PCPM Algorithm

Algorithm 1 summarizes the query processing procedure. This algorithm provides precise query answers, and we call it the **Precise Computation and Precise Maintenance (PCPM)** algorithm. When a query comes, we first perform a kDNN computation, obtain $Q$ and the valid regions, and return the first entry of $Q$ as the answer (lines 1 to 3). Then we start query maintenance (lines 4 to 24). When the query point moves to a new location $q'$, we first test whether $q'$ is still in the AVR. If yes then the query answer has not changed and no further processing is needed (lines 6 to 8). Otherwise we further test whether $q'$ is in the QVR. If not then $Q$ has become invalid and we need to recompute the top-$m$ kDNN sets (lines 9 to 13). If $q'$ is still in the QVR, then $Q$ is still valid, i.e., the new kDNN set is still in $Q$. We update $Q$ with an auxiliary queue $Q'$ to locate the new kDNN set. We dequeue entries from $Q$. If an entry $S_j$'s safe radius $r_{j, m}$ is greater than $\text{dis}(q', q)$, it is still valid and we re-insert it back to $Q'$ prioritized by the new function value $f_q(S_j)$ (lines 14 to 18). Otherwise we only re-inserted it into $Q'$ if $f_q(S_j) > f_q(S'_{|Q'|})$ (lines 19 to 21). When this is done we replace $Q$ and $q$ by $Q'$ and $q'$, respectively, update the valid regions, and return $S_1$ in the new $Q$ (lines 22 to 24).

**Complexity.** The main cost in the above algorithm lies in $kdnSearch(\cdot)$, which is a function that computes the top-$m$ kDNN sets and it takes $O(n^k \log m)$ time. Next we derive how frequent this function is computed.

Assume that the query point moves at a constant speed $v$. In the worst case, the query point keeps moving away from the last point $q$ where $kdnSearch(\cdot)$ is called. When the query point reaches a new location $q'$ where
Algorithm 1: PCM

Input: $q$: query location
Output: $S_t$: kDNN set

1. $Q \leftarrow \text{kdnmSearch}()$
2. $\text{AVR} \leftarrow \text{circle}(q, r_{1,2}), \text{QVR} \leftarrow \text{circle}(q, r_{1,|Q|})$
3. reportResult($S_t$
4. while query continues do
5.   $q' \leftarrow$ query point’s new location
6.   if $q' \in \text{AVR}$ then
7.     reportResult($S_t$
8.     continue
9.   if $q' \notin \text{QVR}$ then
10.    $q \leftarrow q', Q \leftarrow \text{kdnmSearch}()$
11.   $\text{AVR} \leftarrow \text{circle}(q, r_{1,2}), \text{QVR} \leftarrow \text{circle}(q, r_{1,|Q|})$
12.  reportResult($S_t$
13.  continue
14. $Q'.clear()$
15. while not $Q$.empty() do
16.   $S_j \leftarrow Q.dequeue()$
17.   if $r_{j|Q|} > \text{dis}(q', q)$ then
18.     $Q'.enqueue(S_j$
19.   else
20.     if $f_q(S_j) > f_q(S_{Q'Q})$ then
21.       $Q'.enqueue(S_j$
22. $Q \leftarrow Q', q \leftarrow q'$
23. $\text{AVR} \leftarrow \text{circle}(q, r_{1,2}), \text{QVR} \leftarrow \text{circle}(q, r_{1,|Q|})$
24. reportResult($S_t$

\[ \text{dis}(q', q) > r_{1,m}, \text{the queue Q becomes invalid. This takes at least} t_1 = \frac{r_{1,m}}{v} = \frac{(f_q(S_j) - f_q(S_m)) \text{dis}_m}{2\nu \lambda} \text{time. Before the query point reaches q', Q is still valid. Updating Q at every timestamp takes a traversal on the queue (O(m)), recomputing the optimization function value for each entry (O(k^2)), and (possible) re-insertion of the entry into a new queue (O(log m)). This takes O(mk^2 log m) time.} \]

Therefore, in a period of $t_1$ timestamps, the worst case computation cost is $O((t_1 - 1)mk^2 \log m + n^k \log m)$. The amortized computation cost per timestamp is $O\left(\frac{(t_1 - 1)mk^2 \log m + n^k \log m}{t_1}\right)$, which is:

\[ \left(\frac{(f_q(S_1) - f_q(S_m)) \text{dis}_m - 2\nu \lambda mk^2 \log m + 2\nu \lambda n^k \log m}{(f_q(S_1) - f_q(S_m)) \text{dis}_m}\right). \]

Note that compared with $n^k \log m$, $(t_1 - 1)mk^2 \log m$ is negligible when $k > 2$. Therefore, the amortized computation cost can be simplified to be

\[ \frac{2\nu \lambda n^k \log m}{(f_q(S_1) - f_q(S_m)) \text{dis}_m}. \]

The I/O cost of computing the top-$m$ kDNN sets by enumeration depends on the size of the main memory and the data set. If the whole data set can fit in the memory, then the I/O cost is just to load the whole data set into memory, the cost of which is $O\left(\frac{1}{v}\right)$, where $v$ denotes the size of the data set and $B$ denotes the number of data points per disk page. The amortized I/O cost per timestamp is

\[ \frac{2\nu \lambda n^k \log m}{(f_q(S_1) - f_q(S_m)) \text{dis}_m}. \]

If the memory can only hold part of the data set, we need a disk-based algorithm for the enumeration, which is beyond the scope of this paper and will not be discussed further.

\[ \frac{1}{\rho + 1} \left(\frac{1}{\text{dis}_m} - \frac{1}{\text{dis}_m}\right) \]

This inequality defines a circular safe region centered at $q$ with a radius of $\frac{1}{\rho + 1} \left(\frac{1}{\text{dis}_m} - \frac{1}{\text{dis}_m}\right)$. We denote this safe region by $C(q, S)$, as shown in Fig. 6. When the query point is in this region, $S$ is a $\rho$-approximate query answer.

When $C(q, S)$ is derived, we have relaxed both sides of the inequality $f_q(S_{new}) \leq \rho \cdot f_q(S)$. We can derive a stricter

4.2 A $\rho$-Approximate Algorithm for kDNN Maintenance

In this subsection we propose an approximate algorithm to further reduce the frequency of kDNN recomputation. This algorithm does not prefetch the top-$m$ kDNN sets. Instead, it just computes the top-1 kDNN set. When the query point moves, this algorithm keeps checking whether the previously computed kDNN set is still valid. We denote this safe region by $C(q, S)$, as shown in Fig. 6. When the query point is in this region, $S$ is a "good" approximation of the true kDNN set. If yes then the previously computed kDNN set will be returned as the query answer. Otherwise the kDNN set is recomputed. Here, we use $\rho (\rho > 1)$ to denote the approximation ratio bound that defines a "good" approximation. We allow query users to set this bound.

4.2.1 Safe Regions for $\rho$-Approximation

Let the kDNN set computed at $q$ be $S$, and the kDNN set when the query point has moved to $q'$ be $S_{new}$. Then as long as $f_q(S_{new}) \leq \rho \cdot f_q(S)$, $S$ is a $\rho$-approximation of $S_{new}$. According to Section 4.1.1,

\[ f_q(S_{new}) \leq f_q(S_{new}) + \frac{\lambda \text{dis}(q, q')}{\text{dis}_m} \leq f_q(S) + \frac{\lambda \text{dis}(q, q')}{\text{dis}_m}, \]

\[ f_q(S') \geq f_q(S) - \frac{\lambda \text{dis}(q, q')}{\text{dis}_m}. \]

Thus, if $f_q(S) + \frac{\lambda \text{dis}(q, q')}{\text{dis}_m} \leq \rho (f_q(S) - \frac{\lambda \text{dis}(q, q')}{\text{dis}_m})$, then

\[ f_q(S_{new}) \leq \rho \cdot f_q(S) \] will hold. This means:

\[ \text{dis}(q', q) \leq \frac{\rho - 1}{\rho + 1} \frac{f_q(S) \text{dis}_m}{\lambda} \]

(3)
Therefor, the amortized computation cost can be simplified as 

\[
dis(q', q) \leq \frac{(\rho \cdot f_q(S) - f_q(S)) \cdot \text{dis}_m}{\lambda}
\]

(4)

This inequality defines a stricter safe region denoted by \( R(q, S) \). As Fig. 6 shows, \( R(q, S) \) is a region with an irregular shape. This safe region is more expensive to compute compared with \( C(q, S) \) because it requires \( f_q(S) \), which is changing as the query point moves.

Fig. 6 shows how the two safe regions are used in query processing. When the query point moves, we first test whether it is still in \( C(q, S) \) or \( R(q, S) \). If it is in either safe region (e.g., at \( q' \) or \( q'' \)), then there is no further computation. Only when the query point has moved out of both safe regions (e.g., at \( q^* \)) that we need to recompute the top \( k \) DNN set and the corresponding safe regions.

4.2.2 The PCρAM Algorithm

Algorithm 2 summarizes the above procedure, where \( \text{topKdnnSearch}() \) is a function that computes the top-1 \( k \) DNN set (not top-\( m \)). Since it maintains answers with an approximation ratio of \( \rho \), we call it the Precise Computation and \( \rho \)-Approximate Maintenance (PCρAM) algorithm.

Algorithm 2: PCρAM

<table>
<thead>
<tr>
<th>Input</th>
<th>q: query location, ( \rho ): approximation ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output</td>
<td>S: ( k ) DNN set</td>
</tr>
<tr>
<td>1</td>
<td>( S \leftarrow \text{topKdnnSearch}() )</td>
</tr>
<tr>
<td>2</td>
<td>while query continues do</td>
</tr>
<tr>
<td>3</td>
<td>( q' \leftarrow ) query point’s new location</td>
</tr>
<tr>
<td>4</td>
<td>if ( q' \in C(q, S) ) or ( q' \in R(q, S) ) then</td>
</tr>
<tr>
<td>5</td>
<td>( \text{reportResult}(S) )</td>
</tr>
<tr>
<td>6</td>
<td>else</td>
</tr>
<tr>
<td>7</td>
<td>( q \leftarrow q' )</td>
</tr>
<tr>
<td>8</td>
<td>( S \leftarrow \text{topKdnnSearch}() )</td>
</tr>
<tr>
<td>9</td>
<td>( \text{reportResult}(S) )</td>
</tr>
</tbody>
</table>

Complexity. Similar to the cost of the PCPM algorithm, the main cost in the above algorithm is in \( \text{topKdnnSearch}() \), which takes \( O(n^k) \) time.

Assume that the query point moves at a constant speed \( v \). In the worst case, the query point keeps moving away from the last point \( q \) where \( \text{topKdnnSearch}() \) is computed. When the query point reaches a new location \( q' \) where \( \text{dis}(q', q) > \frac{(\rho \cdot f_q(S) - f_q(S)) \cdot \text{dis}_m}{\lambda} \), the safe regions become invalid. This takes \( t_\rho = \frac{v \lambda}{(\rho \cdot f_q(S) - f_q(S)) \cdot \text{dis}_m} \) time. Before the query point reaches \( q' \), testing whether it is still in the safe regions requires computing \( f_q(S) \) with \( O(k^2) \) time.

Therefore, in a period of \( t_\rho \) timestamps, the worst case computation cost is \( O(t_\rho - 1)k^2 + n^k) \). The amortized computation cost per timestamp is \( O(\frac{(t_\rho - 1)k^2 + n^k}{t_\rho}) \), i.e.,

\[
\frac{((\rho \cdot f_q(S) - f_q(S)) \cdot \text{dis}_m - v \lambda k^2 + v \lambda n^k}{(\rho \cdot f_q(S) - f_q(S)) \cdot \text{dis}_m}.
\]

Compared with \( n^k \cdot (t_\rho - 1)k^2 \) is negligible when \( k > 2 \). Therefore, the amortized computation cost can be simplified to be

\[
\frac{n^k v \lambda}{(\rho \cdot f_q(S) - f_q(S)) \cdot \text{dis}_m}.
\]

The I/O cost of computing the top-1 \( k \) DNN sets by enumeration, assuming that the whole data set can fit in the main memory, is \( O(\frac{n}{\lambda}) \), where \( n \) denotes the size of the data set and \( B \) denotes the number of data points per disk page. The amortized I/O cost per timestamp is

\[
\frac{(\rho \cdot f_q(S) - f_q(S))B \text{dis}_m}{k v \lambda}.
\]

The algorithm requires the whole data set to be stored in the memory to compute the \( k \) DNN sets, plus the current top \( k \) DNN set and two circles (safe regions). The space complexity is \( O(n + k) \).

If a client-server based system is used, in \( t_\rho \) timestamps, the \( k \) DNN set only needs to be sent to the client for once. The communication cost is \( O(k) \) and the amortized communication cost is

\[
\frac{(\rho \cdot f_q(S) - f_q(S))B \text{dis}_m}{k v \lambda}.
\]

5 MaxDNN Algorithm Based on Approximate \( k \) DNN Computation

5.1 The MaxSumDispersion Algorithm

As the cost analysis above shows, even with approximate maintenance the overall cost is still quite high due to the enumeration cost \( O(n^k) \) in \( k \) DNN recomputation. In this section, we further propose an algorithm to avoid this high recomputation cost by approximation.

We first revisit the \( k \) DNN query. Similar to [9], [23], we define \( \theta_q(p_i, p_j) \) to be a function of two points \( p_i \) and \( p_j \):

\[
\theta_q(p_i, p_j) = \lambda(2 - \frac{\text{dis}(q, p_i) + \text{dis}(q, p_j)}{\text{dis}_m} + \frac{2(1 - \lambda)}{\text{dis}_m} \text{dis}(p_i, p_j)).
\]

We rewrite Equation 2 with \( \theta_q(p_i, p_j) \) which is the optimization function of the \( k \) DNN query. The equation becomes:

\[
f_q(S) = \frac{1}{k(k - 1)} \sum_{p_i, p_j \in S} \theta_q(p_i, p_j)
\]

(5)

This rewritten equation is very similar to the optimization function of the Maximum Dispersion problem [13]. The Maximum Dispersion problem finds a size-\( k \) subset \( S \) of \( P \) to maximize the following optimization function \( f(S) \):

\[
f(S) = \frac{1}{k(k - 1)} \sum_{p_i, p_j \in S} \theta(p_i, p_j)
\]

(6)

Equations 5 and 6 only differ in that \( \theta(p_i, p_j) \) in Equation 6 involves two points \( p_i \) and \( p_j \), while \( \theta_q(p_i, p_j) \) in Equation 5 involves a third point \( q \).

The Maximum Dispersion problem has been shown to be NP-hard [13], and a 2-approximate algorithm named MaxSumDispersion [24] has been proposed to solve the problem.

As summarized in Algorithm 3, the MaxSumDispersion algorithm simply generates all pairs of data points in \( P \), and insert the pairs (the two data points) into \( S \) according to their \( \theta() \) function values in the descending order (lines 2 to 4). Whenever a pair \((p_x, p_y)\) is chosen for the insertion, any remaining pairs formed by either \( p_x \) or \( p_y \) will be discarded from further consideration (line 5). This procedure ends when \( \lceil \frac{k}{2} \rceil \) pairs have been chosen, and \( S \) will then contain \( 2 \cdot \lceil \frac{k}{2} \rceil \) elements. If \( k \) is odd, \( 2 \cdot \lceil \frac{k}{2} \rceil = k - 1 \). In this case, a random data point from \( P \) is added to \( S \) (lines 6 and 7).
Algorithm 3: MaxSumDispersion

Input: $P$: data set, $k$: query parameter
Output: $S$: a size-$k$ subset of $P$ that maximizes $f(S)$

1. $S \leftarrow \emptyset$
2. for $i = 1$ to $\left\lceil \frac{k}{2} \right\rceil$ do
   3. find $(p_x, p_y) = \arg\max_{p_x, p_y \in P} \theta(p, p_i)$
   4. $S \leftarrow S \cup \{p_x, p_y\}$
   5. $P \leftarrow P \setminus \{p_x, p_y\}$
   6. if $k$ is an odd number then
      7. $S \leftarrow S \cup \{a \text{ random point in } P\}$
   8. return $S$

To process a $k$DNN query, we just need to replace $\theta(p_i, p_j)$ in the algorithm with $\theta_q(p_i, p_j)$. When $k$ is odd, instead of adding a random data point, we add the one from $P$ that is the nearest to the query point $q$. Since these modifications are straightforward, we omit the pseudo-code due to the space limit. For simplicity, in the following discussion, we still call this modified algorithm the MaxSumDispersion algorithm as long as the context is clear.

Next we prove that this algorithm is a 2-approximation algorithm for the $k$DNN query, following a similar procedure to the original MaxSumDispersion algorithm, which was proven to be 2-approximate [13].

Theorem 3. The MaxSumDispersion algorithm is a 2-approximation algorithm for the $k$DNN query.

Proof 3. See Appendix A.

Next we discuss how to process the $k$DNN query with the MaxSumDispersion algorithm.

5.2 A 2-Approximate Algorithm for $k$DNN Maintenance

We use prefetching to maintain the $k$DNN set computed by the MaxSumDispersion algorithm. In particular, when the MaxSumDispersion algorithm is called, we use a priority queue $Q$ to store not only the $\left\lceil \frac{k}{2} \right\rceil$ pairs found by the algorithm but also a few extra point pairs as a "cache" for query updates. Let $p_i$ be a data point in the $\left\lceil \frac{k}{2} \right\rceil$ pairs, and $(p_{\frac{k}{2} + 1}, p_{\frac{k}{2} + 2})$ be the $\frac{k}{2}$th pair. Then $Q$ also stores a point pair $(p_x, p_y)$ if the pair contains $p_i$ and $\theta_q(p_x, p_y) \geq \theta_q(p_{\frac{k}{2} + 1}, p_{\frac{k}{2} + 2})$. Further, we store $m$ more point pairs that have the largest $\theta_q()$ function values smaller than $\theta_q(p_{\frac{k}{2} + 1}, p_{\frac{k}{2} + 2})$. These extra pairs are stored in $Q$ together with the top $\left\lceil \frac{k}{2} \right\rceil$ point pairs in the descending order of their $\theta_q()$ function values. We will study the impact of the value of $m$ in the experimental section.

We modify the MaxSumDispersion algorithm to obtain $Q$ as shown in Algorithm 4. The modified algorithm only contains a few addition lines (lines 7 to 10) to collect the point pairs and store them into $Q$. It does not change the data points to be added to the $k$DNN set $S$, and hence is still a 2-approximate algorithm.

After this algorithm is called we can return the $k$DNN set $S$. When the query point moves, we keep finding the new $k$DNN set from the point pairs in $Q$ until they cannot guarantee a 2-approximation, i.e., $Q$ becomes invalid.

Safe region. Next we define a safe region to guarantee the validity of $Q$ with the safe radius for the data point pairs.

Algorithm 4: MaxSumDispersion-M$k$DNN

Input: $P$: data set, $k$: query parameter
Output: $S$: a size-$k$ subset of $P$ that maximizes $f(S), Q$: a priority queue to store the point pairs for query maintenance

1. $S \leftarrow \emptyset, Q \leftarrow \emptyset, i \leftarrow 1$
2. while $i \leq \left\lceil \frac{k}{2} \right\rceil$ do
   3. find $(p_x, p_y) = \arg\max_{p_x, p_y \in P} \theta_q(p, p_i)$
   4. if $p_x, p_y \notin S$ then
      5. $S \leftarrow S \cup \{p_x, p_y\}$
      6. $i++$
   7. $Q \leftarrow Q \cup \{(p_x, p_y)\}$
8. for $i = 1$ to $m$ do
   9. find $(p_x, p_y) = \arg\max_{p_x, p_y \in P} \theta_q(p, p_i)$
   10. $Q \leftarrow Q \cup \{(p_x, p_y)\}$
11. if $k$ is an odd number then
   12. $S \leftarrow S \cup \{\text{the point nearest to } q \text{ in } P \setminus S\}$
13. return $S, Q$

Definition 4 (Safe radius for two data point pairs). Let the query point be at $q$. Two pairs of data points $(p_{11}, p_{12})$ and $(p_{21}, p_{22})$ satisfy $\theta_q(p_{11}, p_{12}) > \theta_q(p_{21}, p_{22})$. The safe radius for this two pairs, denoted by $r_{p_{i,j}}$, is defined to guarantee $\theta_q(p_{i1}, p_{i2}) > \theta_q(p_{j1}, p_{j2})$ for the query point to move to a new location $q'$ as long as $\text{dis}(q, q') < r_{p_{i,j}}$.

Following the same procedure in Section 4.1.1 but replacing $f_q(S_i), f_q(S_j), f_q'(S_i)$, and $f_q'(S_j)$ with $\theta_q(p_{i1}, p_{i2}), \theta_q(p_{j1}, p_{j2}), \theta_q(p_{i1}, p_{i2}), \theta_q(p_{j1}, p_{j2})$, we can derive that if $\theta(p_{i1}, p_{i2}) > \theta(p_{j1}, p_{j2})$, then $\theta_q(p_{i1}, p_{i2}) > \theta_q(p_{j1}, p_{j2})$ as long as $\text{dis}(q, q') < (\theta_q(p_{i1}, p_{i2}) - \theta_q(p_{j1}, p_{j2})) \cdot \frac{1}{4\lambda}$, i.e.,

$$r_{p_{i,j}} = \frac{(\theta_q(p_{i1}, p_{i2}) - \theta_q(p_{j1}, p_{j2})) \cdot \frac{1}{4\lambda}}{\text{dis}(q, q')}$$

We omit the detailed derivation as it is straightforward. Similar to Theorem 1, we can derive the following property of $r_{p_{i,j}}$.

Theorem 4. Assume that all point pairs of $P$ are sorted in the descending order of their $\theta_q()$ function values, denoted by $(p_{i1}, p_{i2}), (p_{i2}, p_{i3}), \ldots, (p_{C_{i1}}, p_{C_{i2}})$. Then, given three integers $i, j,$ and $pt$:

1) If $1 \leq i < j < pt \leq C_i$, then $r_{p_{i,pt}} > r_{p_{j,pt}}$.
2) If $1 \leq pt < i < j \leq C_i$, then $r_{p_{pt,i}} < r_{p_{pt,j}}$.

Proof 4. See Appendix A.

Let $(p_{i1}, p_{i2})$ be a point pair in $Q$ where both $p_{i1}$ and $p_{i2}$ are in the current $k$DNN set $S$. Then $r_{p_{i1+1}, p_{i2}} < r_{p_{i1}, p_{i2}}, \forall j > i + 1$. When the query point moves from $q$ to $q'$,

$$\text{dis}(q, q') < r_{p_{i+1}, q'} \Rightarrow \text{dis}(q, q') < r_{p_{i,j}}, \forall j > i + 1.$$ 

This means that the point pair $(p_{i1}, p_{i2})$ is still safe, and that no point pairs after $(p_{i1}, p_{i2})$ will be chosen over $(p_{i1}, p_{i2})$ by the MaxSumDispersion algorithm.

Further, if this holds for every point pair in $Q$ where both points are in the current $k$DNN set $S$, then $S$ (and hence $Q$) is still valid, i.e.,

$$\text{dis}(q, q') < \min \{r_{p_{i+1}, p_{i2}} | p_{i1}, p_{i2} \in S\}.$$
We denote \( \min \{ rp_{i,i+1} | p_i, p_{i+1} \in S \} \) by \( rp_{\text{min}} \) and further refine \( rp_{\text{min}} \) as follows.

- If \( p_i, p_{i+1} \) are all in \( S \), then \( rp_{i,i+1} \) can be excluded from \( rp_{\text{min}} \) since swapping \( p_i, p_{i+1} \) and \( (p_{i+1}, p_i) \) will not change \( S \).
- If either \( p_{i+1} \) or \( p_{i+2} \) is in \( S \), then \( rp_{i,i+1} \) can be excluded from \( rp_{\text{min}} \) as well. This is because according to the MaxSumDispersion algorithm, the pair \( (p_{i+1}, p_i) \) will not be added to \( S \).

The refined \( rp_{\text{min}} \) defines the safe region to guard \( S \) and \( Q \).

**Computing the new kDNN set from \( Q \).** When the query point has moved out of the safe region defined by \( rp_{\text{min}} \), \( S \) becomes invalid. We first attempt to compute the new set of \( S \) from the data pairs in \( Q \). We divide \( Q \) into three parts:

- The last point pair of \( Q \), i.e., \( (p_{|Q|}, p_{|Q|}) \).
- Sub-queue \( Q_1 \) that contains every point pair \( (p_i, p_j) \) satisfying \( \text{dist}(q, q') < rp_{i,j} \) i.e., \( Q_1 = \{ (p_i, p_j) \mid \text{dist}(q, q') < rp_{i,j} \} \).
- Sub-queue \( Q_2 \) that contains the rest of the point pairs of \( Q \), i.e., \( Q_2 = \{ (p_i, p_j) \mid \text{dist}(q, q') \geq rp_{i,j} \} \).

The point pairs in \( Q_1 \) still have larger \( \theta_q() \) function values than that of any pair not in \( Q \), as formalized by the following theorem.

**Theorem 5.** \( \forall (p_i, p_j) \in Q_1, \langle p_i, p_j \rangle \notin Q, \) when the query point has moved from \( q \) to a new location \( q' \), we have \( \theta_q(p_i, p_j) > \theta_q(p_j, p_i) \).

**Proof.** See Appendix A.

We compute \( \theta_q() \) for the point pairs in \( Q_1 \), and insert them into a new priority queue \( Q' \) in the descending order of the new function values. The last point pair \( (p_{i'}, p_{j'}) \) in \( Q' \) has the smallest \( \theta_q() \) value, which is still larger than that of any point pair not in \( Q \), according to Theorem 5.

We add a point pair from \( Q_2 \) into \( Q' \) as well if its \( \theta_q() \) function value is larger than or equal to \( \theta_q(p_{i'}, p_{j'}) \).

Then we run the modified MaxSumDispersion algorithm on \( Q' \) and try to find \( \left\lfloor \frac{2}{3} \right\rfloor \) point pairs to form the new \( S \) and \( Q \). If successful then the new sets will be returned. Based on Theorem 5, the new set \( S \) is a 2-approximate answer set.

5.2.1 The 2AC2AM Algorithm

Algorithm 5 summarizes the above procedure. Since this algorithm computes and maintains 2-approximate query answers, we call it the 2-Approximate Computation and 2-Approximate Maintenance (2AC2AM) algorithm. At start, this algorithm computes an approximate kDNN set \( S \) and the priority queue \( Q \) using the MaxSumDispersion-MkDNN algorithm (line 1). Then query maintenance begins. When the query point moves from \( q \) to a new location \( q' \), if \( \text{dist}(q, q') < rp_{\text{min}} \), the query result stays unchanged and no further processing is needed (lines 4 to 8). Otherwise, the algorithm first updates the ranking of the point pairs in \( Q \) (line 9). Then it tries to find \( \left\lfloor \frac{2}{3} \right\rfloor \) pairs from the updated queue \( Q' \) that can contribute to the new kDNN set \( S \) (line 11). If successful then \( Q' \) becomes the new \( Q \) and the new set \( S \) is returned as the query result (lines 12 to 14). If not then the MaxSumDispersion-MkDNN algorithm is called again to obtain the new \( S \) and \( Q \) (line 16). This process repeats and updated query answers will be produced continuously.

Algorithm 5: 2AC2AM

**Input:** \( P \): data set, \( q \): query location, \( k \): query parameter

**Output:** \( S \): kDNN set

\begin{verbatim}
1 S, Q ← MaxSumDispersion-MkDNN(P, q, k)
2 reportResult(S)
3 while query continues do
4     q' ← query point's new location
5     rp_{\text{min}} ← \min \{ rp_{i+1} | p_i, p_{i+1} \in S \}
6     if dist(q', q) < rp_{\text{min}} then
7         reportResult(S)
8     continue
9     Q' ← update Q
10    q ← q'
11    compute S from Q'
12    if |S| = k then
13        Q ← Q'
14        reportResult(S)
15    else
16        S, Q ← MaxSumDispersion-MkDNN(P, q, k)
17        reportResult(S)
\end{verbatim}

**Complexity.** Storing the whole data set in the memory takes \( O(n) \) space. In addition, the algorithm will cache every point pair \( (p_x, p_y) \) if \( p_x \in S \) and \( p_y \notin S \), and that \( \theta_q(p_x, p_y) > \theta_q(p_j, p_y) \) where both \( p_x, p_y \in S \). In the worst case, this will result in \( (k-2)(n-1) \) pairs being cached. This occurs when there are two points \( (p_1, p_2) \) in \( S \) that form a pair with an extremely small \( \theta_q() \) function value, which leads to every point \( p_x \in S, p_x \notin p_1, p_2 \) (a total of \( k-2 \) points) to bring a point pair into the cache with every point in the data set (except for \( p_x \) itself, a total of \( n-1 \) pairs). The algorithm further caches another \( m \) pairs \( (p_x, p_y) \) where \( p_x, p_y \notin S \). Also, it will need to store the current top kDNN set. Therefore, the overall worst case space complexity is \( O(n(n-1)^2 + 2m) \) time. However, this extreme case rarely occurs in reality. For all the data sets tested in our experiments, the cache size is reasonably small that incurs little extra time to maintain.

MaxSumDispersion-MkDNN takes \( O((kn-k+m)n^2 \log(kn-k+m)) \) time to generate the cached pairs. Assume that the query point moves at a constant speed \( v \). In the worst case, the query point keeps moving away from the last point \( q \) where the approximate kDNN set \( S \) is computed. When the query point reaches a new location \( q' \) where \( \text{dis}(q', q) \geq rp_{\text{min}} \), the set \( S \) becomes invalid. This takes at least \( t_2 = \frac{rp_{\text{min}}}{v} \) time. Before the query point reaches \( q' \), computing \( rp_{\text{min}} \) for testing whether \( \text{dis}(q', q) < rp_{\text{min}} \) takes \( O(k) \) time.

Therefore, in a period of \( t_2 \) timestamps, the worst case computation cost is \( O((t_2-1)k \log(kn-k+m)) \). The amortized computation cost is \( O((t_2-1)k \log(kn-k+m)^3) \), i.e.,

\[
\frac{rp_{\text{min}} - v)k + v(k+m)n^2 \log n}{t_2}.
\]
Compared with $v(k + m)n^2 \log n$, $(r_{\min} - v)k$ is negligible. Therefore, the amortized computation cost can be simplified to be $\frac{r_{\min}}{k}$.

Assuming that the whole data set can fit in the main memory, the I/O cost of the MaxSumDispersion-M architecture is $O(\frac{n}{B})$, where $n$ denotes the size of the data set and $B$ denotes the number of data points per disk page. The amortized I/O cost per timestamp is $O(\frac{v}{r_{\min}B})$.

If a client-server based system is used, in $t_2$ timestamps, the $k$DNN set only needs to be sent once. The communication cost is $O(k)$ and is amortized to be $O(\frac{kv}{r_{\min}})$.

6 Experiments

We empirically compare the three proposed algorithms, PCPM (Section 4.1), PCρAM (Section 4.2), and 2AC2AM (Section 5) with a sampling-based algorithm denoted by BASE, which computes a $k$DNN query at every timestamp.

The algorithms are implemented in Java, and ran on a desktop computer with a 3.40GHz Intel Core i7-2600 CPU, 8GB memory, and 64-bit Windows operating system.

We use two real data sets. The first data set contains 501,841 Twitter check-in locations in Los Angeles and nearby regions extracted based on coordinates from the data set used in [25]. The second data set contains 40,629 Foursquare check-in venues in New York City and nearby regions extracted based on coordinates from the data set used in [26]. We denote the two data sets by “LA” and “NY”, respectively. Fig. 7 visualizes the two sets. We see that NY contains fewer data points overall and is more skewed.

We generate two types of trajectories for the query point, “random” and “directional”. In the random trajectories, the query point starts at a random point in the data space, and moves towards a randomly chosen new direction at every timestamp. In the directional trajectories, the query point also starts at a random point and chooses a random direction, but it then keeps moving towards this direction until reaching the boundary of the space, where a new direction towards within the space is chosen. By default, between two timestamps (i.e., two query computations) the query point moves for a distance interval that is randomly varying distribution. By default the LA data set is used.

We vary the query parameter $k$, queue size $m$, optimization weight parameter $\lambda$, approximation ratio $\rho$, query computation distance interval, and data set cardinality in the experiments. The value ranges and default values of these parameters are summarized in Table 1. Due to the inherently high computation complexity of the precise $k$DNN sets in PCPM and PCρAM, we use a data set of 500 data points and $k = 6$ by default so that we can test PCPM and PCρAM together with 2AC2AM under different settings of other parameters. We argue that this is not an unreasonable setting. In real applications such as restaurant or tourist hotspot finding, while the total number of restaurants or tourist hotspots may be large, we can request for additional keywords or categories (e.g., “pizza” or “museum”) to reduce the number of possible candidates to a few hundred. In addition, as the query user is moving it is difficult to browse through a large number of answers, especially when using a mobile device. Therefore, a small $k$ value such as 6 is not unreasonable. In such scenarios, both PCPM and PCρAM may be used. When the data set or $k$ is larger 2AC2AM may be used to provide approximate results.

6.2 Results

Effect of prefetching. PCPM and 2AC2AM both use prefetching to reduce $k$DNN recomputation. In particular, PCPM prefetches top-$m$ $k$DNN sets; 2AC2AM prefetches $m$ extra data pairs. We test the effect of $m$ and aim to find its optimal value to be used in the rest of the experiments.

Fig. 7. The data sets generated between 1 and 900 meters. We did not use a smaller distance interval for the following reason. The data points are distributed in a large space. A smaller distance interval will result in highly stable $k$DNN query answers and large safe regions, which benefits the proposed algorithms but not the BASE algorithm.

We generate 20 trajectories (100 timestamps each) for each set of experiments. We report the average CPU time per timestamp, the average total I/O cost (number of page accesses assuming a page size of 1KB and 50 data points per page) for processing each query, and the average total number of times the $k$DNN set is recomputed for each query. Here, this number of recomputation would represent the communication cost if a client-server based system were used. Further, to verify the effectiveness of the proposed approximation algorithms we report the precision of the algorithms computed by $\frac{f_{\rho}(S)}{f_{\rho}(S_*)}$, where $S$ denotes the $k$DNN set returned by the approximation algorithms and $S_*$ denotes the true $k$DNN set.

We vary the query parameter $k$, queue size $m$, optimization weight parameter $\lambda$, approximation ratio $\rho$, query computation distance interval, and data set cardinality in the experiments. The value ranges and default values of these parameters are summarized in Table 1. Due to the inherently high computation complexity of the precise $k$DNN sets in PCPM and PCρAM, we use a data set of 500 data points and $k = 6$ by default so that we can test PCPM and PCρAM together with 2AC2AM under different settings of other parameters. We argue that this is not an unreasonable setting. In real applications such as restaurant or tourist hotspot finding, while the total number of restaurants or tourist hotspots may be large, we can request for additional keywords or categories (e.g., “pizza” or “museum”) to reduce the number of possible candidates to a few hundred. In addition, as the query user is moving it is difficult to browse through a large number of answers, especially when using a mobile device. Therefore, a small $k$ value such as 6 is not unreasonable. In such scenarios, both PCPM and PCρAM may be used. When the data set or $k$ is larger 2AC2AM may be used to provide approximate results.

Effect of prefetching. PCPM and 2AC2AM both use prefetching to reduce $k$DNN recomputation. In particular, PCPM prefetches top-$m$ $k$DNN sets; 2AC2AM prefetches $m$ extra data pairs. We test the effect of $m$ and aim to find its optimal value to be used in the rest of the experiments.
As Fig. 8 shows, the CPU time, the number of \(k\)DNN recomputation, and the I/O cost of both algorithms drop initially as \(m\) increases, which is expected because a larger “cache” reduces the frequency of \(k\)DNN recomputation. However, when \(m\) continues to increase, the benefit of “caching” increases little and higher queue maintenance cost is required. As such, the algorithm costs rise back. We observe that: PCPM shows the best performance at \(m = 40\) on the LA data set and \(m = 60\) on the NY data set; 2AC2AM shows the best performance at \(m = 30\) on the LA data set and \(m = 50\) on the NY data set. The larger optimal values of \(m\) on a more skewed data set (i.e., the NY data set) can be explained by that, on a more skewed data set, the query answer becomes invalid more frequently in the denser area. It requires a larger preferred queue to reduce the recomputation frequency. In the rest of the experiments, we use these optimal values as the default values.

Fig. 8 (d) shows the precision of the two algorithms. As we can see, PCPM provides accurate query answers all the time (precision = 1), while 2AC2AM provides 2-approximation results (precision \(\geq 0.5\)). Note that as \(m\) increases, the accuracy of 2AC2AM decreases. This is because a large “cache” discourages \(k\)DNN recomputation as long as the current answer is still a 2-approximation, which means that the precision tends to stay at 0.5.

**Effect of query computation distance interval.** In Fig. 9, we compare the three proposed algorithms with the BASE algorithm on the LA data set using directional query trajectories, where the query computation distance interval is varied from 100 to 8100 meters. Note that, for every interval value used (e.g., 900), the query point speed is not fixed at the interval value per timestamp (e.g., 900 meters per timestamp). Instead, the query point speed is randomly chosen between 1 and the interval value at every timestamp. When the distance interval increases, the CPU cost, the number of \(k\)DNN recomputation, and the I/O cost increase for the three proposed algorithms while those of BASE stay stable. This is because BASE simply recomputes \(k\)DNN at every timestamp, which is not affected by the computation interval; the proposed algorithms rely on safe regions to reduce the recomputation frequency, which become invalid more frequently as the query distance interval increases. However, the three proposed algorithms outperform BASE consistently, which validates the effectiveness of our proposed techniques to reduce the recomputation frequency. In particular, 2AC2AM has the lowest CPU cost (Fig. 9 (a)), because it uses an approximate \(k\)DNN algorithm with a low complexity. Meanwhile, PC\(\rho\)AM has the lowest recomputation cost (Fig. 9 (b)). This is because of the strict bounds used by PC\(\rho\)AM that define large safe regions. Correspondingly, PC\(\rho\)AM also has the lowest I/O cost. In Fig. 9 (d), both BASE and PCPM show constant precision values of 1 since they are both precise algorithms. The two approximate algorithm 2AC2AM and PC\(\rho\)AM (\(\rho = 1.5\)) show precision values that are no less than 0.5 and 0.66, respectively. These confirm the effectiveness of the proposed algorithms.

We have also tested the algorithms on random query trajectories. The comparative performance of the algorithms is similar to that shown in Fig. 9. To keep the paper concise we have put the figure in Appendix B. In the following experiments we omit the figures on random query trajectories. A further observation is that the costs for the proposed algorithms are generally lower on random query trajectories. This is because when the query point moves randomly instead of directionally, its probability of staying in the current safe region is higher and hence the \(k\)DNN recomputation frequency is lower.

On the NY data set, the comparative performance of the algorithms also show a similar pattern, we omit the figures due to space limit (same in the following experiments).

**Effect of data set size.** Next we vary the data set size. In Fig. 10 we randomly sample data points from the LA data set to obtain data set of different sizes. As Fig. 10 (a) shows, computing a single \(k\)DNN query takes more than 20 seconds for the BASE algorithm on 500 data points. This is because the \(Mk\)DNN query is inherently complex. The
two proposed algorithms PCPM and PCρAM have lower average running time (around 1 second) per timestamp, but at timestamps where kDNN recomputation is needed, they would also need to run for more than 20 seconds. Only 2AC2AM can process the query in time for the larger data sets, which demonstrates the scalability of this algorithm.

The costs are even higher when larger data sets are used, since the safe regions will become smaller and kDNN set recomputation will become more frequent. Note that in Fig. 10 (a) we have omitted the points beyond 10000 seconds, as they are too high to be practical.

Fig. 11 shows the algorithm performance on data sets obtained by partitioning the NY data set. In particular, we partition the space by a 3 × 3 regular grid. As a result we obtain 9 blocks denoted by $b_0, b_1, ..., b_8$, each with 26207, 7342, 2926, 1674, 1144, 1023, 163, 150, 0 data points, respectively.

Note that $b_8$ is the top left block which does not contain any data points and hence no experiment is performed on it. As the figure shows, in $b_8$, only 2AC2AM can produce the query answers. All the other algorithms rely on precise kDNN computation, and cannot produce query answers in time on such dense and skewed data set. In the other blocks, the algorithms’ costs drop as the data points become fewer and the distribution becomes more even. The proposed algorithms again show better performance consistently.

Effect of $k$. Fig. 12 shows the results when we vary $k$ from 3 to 48. Note that BASE, PCPM, and PCρAM all involve computing precise kDNN answers, which is NP-hard. As $k$ increases their computational costs increase dramatically. In Fig. 12 (a) we have also omitted the points beyond 10000 seconds. Meanwhile, 2AC2AM uses an approximation algorithm for kDNN recomputation with much lower complexity. It scales much better with larger $k$ values.

Effect of $\lambda$. In the optimization function we use a parameter $\lambda$ to allow query users to set the preference on spatial proximity over diversity. Fig. 13 shows the impact of $\lambda$. Again the proposed algorithms outperform BASE. An observation is that the costs of the proposed algorithms increase as $\lambda$ increases. This can be explained by that a larger $\lambda$ value will lead to smaller safe regions, as shown by the safe region definition equations. This will lead to higher kDNN recomputation frequency and hence higher costs.

Effect of $\rho$. In PCρAM, we use a parameter $\rho$ to allow a query user to set the required approximation ratio. Fig. 14 shows the impact of $\rho$. We see that a larger value of $\rho$ can reduce the costs, because the required precision drops and less frequent kDNN recomputation is needed. Note that the precision value is still bounded by $\frac{1}{3}$ in all cases shown in Fig. 14 (d). This confirms the effectiveness of PCρAM. When $\rho = 1$, PCρAM produces precise query answers at every timestamp, which are the same as those produced by PCPM. However, its performance is worse because it does not “cache” any kDNN candidates, while PCPM has a queue $Q$ to serve as a “cache”. When $\rho = 2$,
We first obtain the \( k \)NN sets by setting the distance-diversity trade-off parameter \( \lambda \) to be 1 in PCPM (PCPM-\( \lambda = 1 \)). This way the optimization function is only determined by the spatial distance of the data points to the query point, not the diversity, and hence the algorithm will produce the \( k \)NN sets, not \( k \)DNN sets. We use these \( k \)NN sets as the benchmark for comparing the similarity between the different query answer sets. We then run PCPM at \( \lambda = 0.5 \) to produce precise \( k \)DNN sets (PCPM-\( \lambda = 0.5 \)). Similarly, we run PC\( \rho \)AM at \( \lambda = 1, 0.5 \) (PC\( \rho \)AM-\( \lambda = 1 \), PC\( \rho \)AM-\( \lambda = 0.5 \), and vary \( \rho \) from 1 to 2 (the precision is therefore varied from 1 to 0.5). Now the algorithm will produce approximate \( k \)NN and \( k \)DNN sets. We do the same for 2AC2AM as well (2AC2AM-\( \lambda = 1 \), 2AC2AM-\( \lambda = 0.5 \)). Its precision cannot be varied, however, because it is designed to return \( k \)DNN sets with an approximation ratio of 0.5.

We record the average jaccard similarity between the actual \( k \)NN set and the sets obtained as above. Here, the Jaccard similarity between two sets \( S_1 \) and \( S_2 \) is defined as:

\[
\text{similarity}(S_1, S_2) = \frac{|S_1 \cap S_2|}{|S_1 \cup S_2|}
\]

Fig. 15 shows the result. Specifically, PCPM-\( \lambda = 1 \) are the precise \( k \)NN sets. The similarity value is 1. PCPM-\( \lambda = 0.5 \) are the precise \( k \)DNN sets. The similarity to the \( k \)NN sets is just 0.4. For PC\( \rho \)AM, its similarity value drops as \( \rho \) increases (i.e., less precise answer sets are produced). When \( \lambda = 0.5 \) the \( k \)DNN sets computed are very different from the \( k \)NN sets. The similarity values are below 0.4 for all cases tested. In comparison the approximate \( k \)NN sets produced by PC\( \rho \)AM at \( \lambda = 1 \) are more similar to the actual \( k \)NN sets (similarity values above 0.5). These approximate \( k \)NN sets are still significantly different from the approximate \( k \)DNN sets produced by PC\( \rho \)AM at \( \lambda = 0.5 \). Similar difference is observed on the approximate \( k \)DNN sets produced by 2AC2AM-\( \lambda = 0.5 \), except that the set similarity is unaffected by \( \rho \) (note when \( \lambda = 1 \), 2AC2AM will compute precise \( k \)NN sets according to the MaxSumDispersion algorithm). Therefore, we can see that the \( k \)DNN sets are significantly different from the \( k \)NN sets, which confirms that effectiveness of the \( M \)kDNN query to diversity the query result.

7 Conclusion
We formulated the \( M \)kDNN query and conducted a comprehensive study. We first proposed two algorithms to reduce the query recomputation frequency, using the prefetching technique and the safe region technique, respectively. Our experiments show that they both can process the \( M \)kDNN query effectively. However, due to the high cost of query
recomputation, they still lacked computation efficiency. To overcome this limitation, we further proposed an algorithm that computes approximate kDNN sets at recomputation, which successfully reduces the recomputation cost and hence the overall query costs. We conducted a detailed cost analysis for the three proposed algorithms and extensive experiments to evaluate their performance. The results confirmed our cost analysis and the advantages of the proposed algorithms over the baseline algorithm.

Acknowledgment. This work is supported by the National Basic Research Program of China (973 Program) under Grant No. 2012CB316201 and the National Natural Science Foundation of China under Grant No. 61472071, 61433008, and 61402155. Jianzhong Qi is supported by the Melbourne School of Engineering Early Career Researcher Grant (project reference number 4180-E55), and the University of Melbourne Early Career Researcher Grant (project number 603049). Rui Zhang is supported by ARC Future Fellow project FT12010832.

REFERENCES

Yu Gu received his Ph.D. degree in computer software and theory from Northeastern University, China, in 2010. Currently, he is an associate professor and Ph.D. supervisor at Northeastern University, China. His current research interests include spatial data management and graph data management. He is a senior member of China Computer Federation (CCF) and a member of ACM.

Guanli Liu received his M.E. degree from Northeastern University, China, in 2015. His research interest is spatial data management. He is currently a software engineer at Baidu, China.

Jianzhong Qi is a lecturer in the Department of Computing and Information Systems at the University of Melbourne. He received his Ph.D degree from the University of Melbourne in 2014. He has been an intern at Toshiba China R&D Center and Microsoft Redmond in 2009 and 2013, respectively. His research interests include spatio-temporal databases, location-based social networks, information extraction, and text mining.

Hongfei Xu received his M.E. degree in computer science from Shenyang Jianzhu University in 2014. He is currently a Ph.D. candidate in computer software and theory at Northeastern University, China. His research interest is spatial data management.

Ge Yu received the Ph.D. degree in computer science from Kyushu University of Japan in 1996. He is currently a professor at Northeastern University, China. His research interests include distributed and parallel database, OLAP and data warehousing, data integration, graph data management, etc. He has published more than 200 papers in refereed journals and conferences. He is a member of the IEEE Computer Society, IEEE, ACM, and CCF.

Rui Zhang is currently an Associate Professor and Reader in the Department of Computing and Information Systems at The University of Melbourne, Australia. He received his Ph.D. degree from National University of Singapore in 2006. His research interest is data and information management in general, particularly in areas of spatial and temporal data analytics, moving object management, indexing techniques, high performance computing and data streams.