Fast and Exact Outlier Detection in Metric Spaces: A Proximity Graph-based Approach

Daichi Amagata
Osaka University, PRESTO
Japan
amagata.daichi@ist.osaka-u.ac.jp

Makoto Onizuka
Osaka University
Japan
onizuka@ist.osaka-u.ac.jp

Takahiro Hara
Osaka University
Japan
hara@ist.osaka-u.ac.jp

ABSTRACT

Distance-based outlier detection is widely adopted in many fields, e.g., data mining and machine learning, because it is unsupervised, can be employed in a generic metric space, and does not have any assumptions of data distributions. Data mining and machine learning applications face a challenge of dealing with large datasets, which requires efficient distance-based outlier detection algorithms. Due to the popularization of computational environments with large memory, it is possible to build a main-memory index and detect outliers based on it, which is a promising solution for fast distance-based outlier detection.

Motivated by this observation, we propose a novel approach that exploits a proximity graph. Our approach can employ an arbitrary proximity graph and obtains a significant speed-up against state-of-the-art. However, designing an effective proximity graph raises a challenge, because existing proximity graphs do not consider efficient traversal for distance-based outlier detection. To overcome this challenge, we propose a novel proximity graph, MRPG. Our empirical study using real datasets demonstrates that MRPG detects outliers significantly faster than the state-of-the-art algorithms.

1 INTRODUCTION

Outlier detection is a fundamental task in many applications, such as fraud detection, health check, and noise data removal [1, 33]. These applications often employ distance-based outlier detection (DOD) [21] (as described later), because DOD is unsupervised, can be employed in any metric spaces, and does not have any assumptions of data distributions. This paper addresses the DOD problem.

Motivation. DOD requires a range threshold r and a count threshold k as input parameters. Given a set P of objects, an object $p \in P$ is an outlier if there are less than k objects $p' \in P$ such that $dist(p, p') \leq r$, where $dist(p, p')$ evaluates the distance between p and p' in a data space. (Density-based clustering also employs this definition to identify noises [2, 16].) Motivated by a recent trend of machine learning-related applications, we are interested in an efficient solution that can be employed in any metric spaces.

Classification, prediction, and regression utilize machine learning techniques, because they can provide high accuracy. To train high performance models, noises (i.e., outliers) should be removed from training datasets, because the performances of models tend to be affected by outliers [1, 24, 34]. It is now a common practice for many applications to remove noises as a pre-processing of training [7, 19], and DOD can contribute to this noise removal. Besides this, natural language processing, medical diagnostics, and image analysis also receive benefits from DOD. For example, DOD is utilized to make datasets clean and diverse by finding error or unique sentences from sentence embedding vectors [23]. Campos et al. tested Euclidean DOD on medical and image datasets and confirmed that DOD successfully finds unhealthy people and irregular images [11].

To cover these applications, a DOD technique needs to deal with many distance functions. This is because the above noise removal application can have many data types (e.g., multi-dimensional points, strings, and time-series) and word (sentence) embedding vectors usually exist in angular distance spaces [28]. In addition, they need to deal with large datasets [20], thereby a scalable solution for metric spaces is required. Due to the popularization of main-memory databases [37], in-memory processing of DOD on a large dataset is possible. Fast DOD would be achieved by building an efficient main-memory index offline. Some studies proposed metric DOD techniques [4, 21, 30], but they miss this observation.

Challenge. To design an efficient index-based solution for any metric spaces, we address the following challenges: (i) general and effective index to any r and k, (ii) space efficiency, and (iii) robustness to any metric spaces.

(i) General and effective index to any r and k. Because we do not know r and k in advance, an index has to deal with any r and k. Building an index that is general to r and k and effectively supports fast DOD is not trivial. The state-of-the-art algorithms [4, 30] build a simple data structure in an online fashion after r and k are specified. The pruning efficiency of this index built online is limited, so they need long time to detect outliers.

(ii) Space efficiency. Let p' be the k-th nearest neighbor of an object p. If $dist(p, p') \leq r$, p is not outlier. Therefore, if p stores a sorted array that maintains the distance to each object in P, whether p is an outlier or not can be evaluated in $O(1)$ time. However, this approach requires $O(n^2)$ space, where $n = |P|$, so is not practical.

(iii) Robustness to any metric spaces. Since we consider metric spaces and recent applications usually deal with middle or large dimensional data, robustness to any data types and dimensionality is important. Notice that we can employ range queries to evaluate whether given objects are outliers or not. A simple and practical solution is to build a tree-based index offline and iteratively conduct a range query on the index for each object. However, space-partitioning approaches like tree structures are efficient only for low-dimensional data. That is, the computational performances of existing techniques [4, 30] degrade on high-dimensional data.

Our Contributions. We overcome the above challenges and make the following contributions[1].

[1] This is the full version of [3].

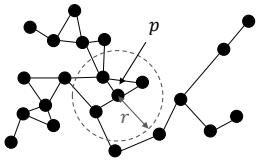


Figure 1: Example of a proximity-graph. Each object (black vertex) has links to its similar (nearby) objects.

• *Novel DOD algorithm that exploits a proximity graph* (Section 4). We propose a new technique for the DOD problem that filters non-outliers efficiently while guaranteeing the correctness by exploiting a proximity graph. In a proximity graph, an object p is a vertex, and each object has links to some of its similar objects, as shown in Figure 1, which assumes a Euclidean space. The following example intuitively explains the filtering power of a proximity graph (a non-outlier can be filtered in $O(k)$ time).

Example 1. *Let p be the center of the gray circle with radius r in Figure 1. Assume $k = 3$, and we can see that p is not an outlier by traversing its links.*

This novel idea of graph-based filtering yields a significant improvement, because it avoids the impact of the curse of dimensionality and we need to verify only non-filtered objects. Note that our algorithm (i) is orthogonal to any metric proximity graphs, (ii) is parallel-friendly, and (iii) detects all outliers correctly.

Furthermore, the above idea provides a new result: the time complexity of our solution is $O((f + t)n)$, where f is the number of false positives incurred by the filtering and t is the number of outliers. This result states that, if $f + t = o(n)$ in the worst case, our solution does not need $O(n^2)$ time. Real datasets usually have this case, whereas the existing DOD algorithms [4, 21, 30] essentially incur $O(n^2)$ time. (Empirically, our solution scales almost linearly to n on real datasets.)

• *Novel metric proximity graph* (Section 5). To maximize the performance of our solution, f should be minimized, and high reachability of neighbors (objects within distance being not larger than r) achieves this. Motivated by this observation, as our second contribution, we devise MRPG (Metric Randomized Proximity Graph), a new proximity graph specific to the DOD problem. When r or k is large, to evaluate whether p is not an outlier, we may need to traverse objects existing in more than 1-hop from p in a proximity graph. However, existing proximity graphs are not designed to consider reachability to neighbors, which increases f. The novelty of MRPG is that MRPG improves the reachability of neighbors by making pivot-based monotonic paths between objects with small distances, so that, for a given p, *we can greedily traverse p's neighbors from p*. The space of an MRPG is reasonable, i.e., linear to n.

How to build a MRPG efficiently is not trivial, so we also propose an efficient algorithm that builds a MRPG in linear time to n. We show that simply improving reachability between objects incurs $\Omega(n^2)$ time, which clarifies that our algorithm is much faster. Our MRPG building algorithm improves the reachability of neighbors while keeping a theoretically comparable efficiency with the state-of-the-art algorithm that builds an approximate K nearest neighbor graph [15] (and our algorithm is empirically faster).

• *Extensive experiments* (Section 6). We conduct experiments using various real datasets and distance functions. The results demonstrate that our algorithm significantly outperforms the state-of-the-art. Besides, MRPG provides faster response time than existing metric proximity graphs.

Besides the above contents, Section 2 defines the problem, Section 3 reviews related work, and Section 7 concludes this paper.

2 PROBLEM DEFINITION

Let P be a set of n objects (i.e., $n = |P|$). The neighbors of an object $p \in P$ are defined as follows:

Definition 1 (Neighbor). *Given a distance threshold r and an object $p \in P$, $p' \in P \backslash \{p\}$ is a neighbor of p if $dist(p, p') \leq r$.*

We consider that $dist(\cdot, \cdot)$ satisfies metric, i.e., triangle inequality. We next define distance-based outlier and our problem.

Definition 2 (Distance-based outlier). *Given a distance threshold r, a count threshold k, and a set of objects P, an object $p \in P$ is a distance-based outlier if p has less than k neighbors.*

Problem statement. *Given a distance threshold r, a count threshold k, and a set of objects P, the distance-based outlier detection problem finds all distance-based outliers.*

Hereinafter, a distance-based outlier is called an outlier. We use inliers to denote objects that are not outliers. As with recent works [17, 25, 29, 38], we focus on a single machine and static and memory-resident P. (If P is dynamic, we can use one of the state-of-the-art algorithms, e.g., [22, 32].)

3 RELATED WORK

Outlier detection in metric spaces. A nested-loop algorithm [21] is a straightforward solution for our problem. Given an object $p \in P$, this algorithm counts the number of neighbors of p by scanning P and terminates the scan when the count reaches k. This algorithm incurs $O(n^2)$ time, so does not scale to large datasets.

Given r, SNIF [30] forms clusters with radius $r/2$ (cluster centers are randomly chosen). If the distance between an object p and a cluster center is within $r/2$, p belongs to the corresponding cluster. From triangle inequality, the distances between any objects in the same cluster are within r. Therefore, if a cluster has more than k objects, they are not outliers. Even if a cluster has less than $k + 1$ objects, objects in the cluster do not have to access the whole P. This is because each object p can avoid accessing objects p' such that $dist(p, p') > r$ by using clusters.

DOLPHIN [4] is also a scan-based algorithm. This algorithm indexes already accessed objects to investigate whether the next objects are inliers. DOLPHIN can know how many objects exist within a distance from the current object p. If there are at least k objects within the distance, DOLPHIN does not need to evaluate the number of neighbors of p any more.

The main issue of the above algorithms is their time complexity. They rely on the (group-based) nested-loop approach and incur $O(n^2)$ time. Besides, they lose distance bounds for high-dimensional data due to the curse of dimensionality, rendering degraded performance. In addition to these solutions, an algorithm that exploits range search can also solve the DOD problem, as can be seen from

Definition 1. As one of baselines, we employ VP-tree [35], because [13] demonstrated that VP-tree is the most efficient solution for the range search problem in metric spaces. Each node of a VP-tree stores a subset P' of P, the centroid object in P', and the maximum value among the distances from the centroid to the objects in P'. A range search on VP-tree is conducted as follows. The lower-bound distance between a query and any node can be obtained by using the maximal value and triangle inequality. If this lower-bound distance is larger than r, the sub-tree rooted at this node is pruned (otherwise, its child nodes are accessed). How to build a VP-tree is introduced in Section 5.1.

Proximity graphs have been demonstrated to be the most promising solution to the k-NN search problem [25]. If the distance between an object p and its k-th NN is within r, p is not an outlier. From this observation, we see that proximity graphs have a potential to solve the DOD problem efficiently. Some proximity graphs [5, 17, 18] are dependent on L_2 space, so we review only proximity graphs that can be built in metric spaces.

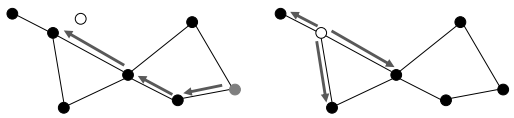
One of the most famous proximity graphs is KGraph. In this graph, each object is considered as a vertex and has links to its approximate K-NN (AKNN) objects (i.e., K is the degree of the graph). This graph is built by NNDescent algorithm [15]. Our proximity graph is also based on an AKNN graph, and we extend NNDescent to build an AKNN graph more efficiently in Section 5.1. Actually, simply employing KGraph may incur some problems. For example, its reachability to neighbors can be low if $k > K$.

Another famous proximity graph is based on navigable small-world network models [9]. In a graph based on this model, the number of hops between two arbitrary nodes is proportional to $\log n$. Building a graph based on this model incurs $O(n^2)$ time, thereby an approximate solution, NSW, was proposed in [26]. To speed up approximate nearest neighbor (ANN) search, its hierarchical version, HNSW, was proposed in [27]. The upper layers of HNSW are built by sampling objects in their lower layers. This structure enables search algorithms, whose start object is a *random* vertex (i.e., object) in HNSW, to skip unnecessary vertices and to reach a nearby (similar) vertex from a query object. When we evaluate the number of neighbors of $p \in P$, p can be considered as a query object. Figure 2(a) depicts the search process in ANN problem: it starts from a random vertex (the grey one) and traverses the proximity graph so that the next vertex is closer to the query object (white one) than the former one. On the other hand, in our problem, a query object is one of the objects in P. It is clearly better to traverse the graph from the query object for finding its neighbors, as shown in Figure 2(b). Therefore, we do not need the skipping structure of HNSW (thus do not consider it as a baseline).

Although these proximity graphs can be employed in our solution proposed in Section 4, they cannot optimize the performance of our solution. This is because they are not designed for the DOD problem and do not consider reachability to neighbors. Therefore, we propose a new proximity graph for the DOD problem that takes the reachability into account in Section 5.

4 OUR DOD ALGORITHM

Let t be the number of outliers in P. A range search in metric spaces with arbitrary dimensionality needs $O(n)$ time. Therefore, the DOD problem needs $\Omega(tn)$ time (because we have to evaluate not only outliers but also inliers). To scale well to large datasets, it is desirable that the time complexity of a solution nearly matches the lower-bound. Designing such a solution is however not straightforward. Our new technique for the DOD problem overcomes this non-trivial challenge.



(a) ANN problem (b) Our problem

Figure 2: Difference between our and ANN problems w.r.t. graph traversal which is represented by arrows. Grey and white vertices (objects) respectively represent a starting and a query vertex.

Main idea. Given P, the ratio of outliers in P is small (usually less than one percent) [36]. That is, most objects in P are inliers, so we should identify them as inliers quickly, to reduce computation time. The evaluation of whether or not p is an inlier can be converted to answering the problem of range counting with query object p and radius r. Therefore, to filter inliers quickly, we need an efficient solution for the problem of range counting, with early termination when the count reaches k. Proximity graphs recently have shown high potential for solving the approximate nearest neighbor search problem [25], thanks to their property of the connections between similar objects. This property is also promising for the range counting problem. Because, each object p has links to its similar objects in a proximity graph, we can efficiently count the number of neighbors of p by traversing the graph from p, *regardless of the dimensionality of the dataset.* Figures 1 and 2(b) depict its intuition.

To implement this idea, we propose a proximity graph-based solution, a novel approach for the DOD problem. Algorithm 1 describes its overview. This solution consists of a filtering phase (lines 2–5) and a verification phase (lines 7–10).

Filtering phase. In this phase, we filter inliers by exploiting a proximity graph G, which is built in one-time pre-processing phase. More specifically, we propose Greedy-Counting (Algorithm 2) to count the number of neighbors of an object p on G. Consider that a vertex v in G corresponds to an object p (v and p are hereinafter used interchangeably in the context of G). Let $v.E$ be the set of links between v and some other vertices. Given an object p, r, and k, Greedy-Counting greedily traverses G from v, as long as a visited vertex v' satisfies $dist(p, p') \leq r$, so as to count the number of neighbors of p. In other words, we first check $v.E$: for each $(v, v') \in v.E$ where v' has not been visited, we increment the count by one and insert p' into a queue Q, iff $dist(p, p') \leq r$. We next pop the front of Q, say v', check $v'.E$, and do the same as v. One exception appears in line 13, and this is necessary for MRPG, which is explained in Section 5.4. Greedy-Counting is terminated when the count reaches k or Q becomes empty. It is important to see that:

Lemma 1. *Our filtering does not incur false negatives.*

Proof. All proofs appear in Appendix. □

Algorithm 1: Proximity Graph-based DOD

Input: P, r, k, and a proximity graph G

```
/* Filtering phase */
P' ← ∅
for each p ∈ P do
    if Greedy-Counting(p, r, k, G) < k then
        P' ← P' ∪ {p}
/* Verification phase */
P_out ← ∅
for each p ∈ P' do
    if Exact-Counting(p, r, k) < k then
        P_out ← P_out ∪ {p}
return P_out
```

Verification phase. Let P' be the set of objects whose counts returned by Greedy-Counting are less than k. From Lemma 1, P' contains all outliers but does false positives (i.e., inliers but not filtered). We therefore have to verify whether or not objects in P' are really outliers. Exact-Counting in Algorithm 1 verifies them in the following way:

- For data with low intrinsic dimensionality[2], we conduct a range counting on a VP-tree [35] for each object in P'.
- For the other data, we use a linear scan, because this is more efficient than any indexing methods for high-dimensional data.

We terminate range counting or sequential scan for p when the count of p reaches k. Since this phase counts the exact number of neighbors for all outliers in P' and P' contains all outliers, *Algorithm 1 returns the exact answer.*

Time analysis. Hereinafter, we assume that the dimensionality is fixed. We analyze the time complexity of Algorithm 1.

Theorem 1. *Algorithm 1 requires $O((f+t)n)$ time, where f and t are respectively the numbers of false positives and outliers.*

Remark. From the above result, we see that our solution theoretically does not need $O(n^2)$ time, if $f + t = o(n)$ in the worst case. This holds in practice, so our result supports a significant speed-up over the existing (r, k)-DOD algorithms. We here note that (1) real datasets have $t \ll n$ [22, 36], and (2) t is usually dependent not on n but on data distributions. These and Theorem 1 suggest that our solution with a proximity graph yielding a small f can be (almost) linear to n in practice.

Multi-threading. Algorithm 1 iteratively evaluates objects independently, thereby can be parallelized. (Given multi-threads, each thread independently evaluates assigned objects in both the filtering and verification phases.) However, to exploit multi-threading, balancing the load of each thread is important. The early termination in the verification phase cannot function for outliers, since they do not have k neighbors. The filtering and verification costs of outliers are hence larger than those of inliers. That is, keeping load balance is hard in our problem theoretically, as we do not know outliers in advance. To relieve this, we employ a random partitioning approach for assigning objects into each thread.

[2]This is the minimum number of parameters (variables) that are needed to represent a given dataset. For example, when this is less than 5, it can be considered as low.

Algorithm 2: Greedy-Counting

Input: p_i, r, k, and a proximity graph G

```
count ← 0, Q ← {v_i}, check v_i as visited
while Q ≠ ∅ do
    v ← the front of Q
    Q ← Q\{v}
    for each v' ∈ v.E where v' has not been checked as visited do
        Check v' as visited
        if dist(p, p') ≤ r then
            count ← count +1
            if count = k then
                break
            Q ← Q ∪ {v'}
        else
            if p' is a pivot then
                Q ← Q ∪ {v'}
    if count = k then
        break
return count
```

5 MRPG

hile our DOD algorithm is orthogonal to any proximity graphs, its performance (i.e., f) depends on a given proximity graph. To maximize the performance, we have to minimize f in the filtering phase. Therefore, the main challenge of this section is to reduce f. To overcome this, in a proximity graph, neighbors of an arbitrary object p should be reachable from p for Greedy-Counting.

Consider an inlier p. To accurately identify p as an inlier in the filtering phase (i.e., to reduce f), a proximity graph G should have paths from p to its neighbors that can be traversed by Greedy-Counting. Our idea that achieves this is to introduce *monotonic path*, a path from p such that Greedy-Counting can traverse its neighbors in non-decreasing order w.r.t. distance.

Definition 3 (Monotonic path). *Consider two objects p_i and p_{i+x} in P. Let v_i, v_{i+1}, ..., v_{i+x} be a path from p_i to p_{i+x} in a proximity graph (that is, v_{i+j} has a link to v_{i+j+1} for all $j \in [0, x-1]$). Iff $dist(v_i, v_{i+j}) \leq dist(v_i, v_{i+j+1})$ for all $j \in [0, x-1]$, this path is a monotonic path.*

If G has at least one monotonic path between any two objects, G is a monotonic search graph (MSG) [14]. Although a MSG can reduce f, building it in metric spaces requires $\Omega(n^2)$ time (see Theorem 3), meaning that reducing f with a proximity graph that can be built in a reasonable time is not trivial. To solve this challenge, we propose MRPG (Metric Randomized Proximity Graph), an approximate version of MSG. MRPG incorporates the following properties to reduce f.

Property 1: each object has links to its approximate K-NNs.
Property 2: monotonic paths are created based on pivots (a subset of P).
Property 3: candidates of outliers have their exact K'-NNs, where $K' \geq K$.

The benefits of these properties are as follows. First, thanks to the first property, Greedy-Counting tends not to miss accessing similar objects. Second, the graph traversal in Algorithm 2 goes

through pivots. Assume that we now visit a pivot when counting the number of neighbors of p. If the pivot has a monotonic path to the neighbors of p, reachability between p and its neighbors is improved. Now the challenge is how to choose pivots to receive this benefit for many objects. Random sampling is clearly not effective because it produces biased samples from dense subspaces (objects in dense spaces are easy to reach their neighbors). Our approach is that we choose pivots from each subspace of P, because this approach can choose pivots from (comparatively) sparse spaces and reachability between objects existing in such spaces is also improved. (How to efficiently identify subspaces is introduced in Section 5.1.) Last, the third property is simple yet important. If objects that would be outliers have links to their exact K'-NNs, we can efficiently know whether or not they are outliers, if $k \leq K'$.

This section presents a non-trivial MRPG building algorithm, which satisfies the above properties through the following steps:

1. NNDescent+: this builds an AKNN graph. We extend a state-of-the-art AKNN graph building algorithm NNDescent, to quickly build it.
2. Connect-SubGraphs: this connects such sub-graphs to guarantee that MRPG is strongly connected, because an AKNN may have disjoint sub-graphs.
3. Remove-Detours: this creates monotonic paths by removing detours. We utilize a heuristic approximation.
4. Remove-Links: this removes unnecessary links to avoid redundant graph traversal.

The first step achieves the properties 1 and 3. Then, we obtain the property 2 in the third step. In Section 5.5, we show that this algorithm achieves *linear time to n*. That is, we achieve a reduction of f by using a MRPG (i.e., the three properties) that can be obtained in a reasonable time.

5.1 NNDescent+

MRPG is based on an AKNN graph, so we need an efficient algorithm for building an AKNN graph. Building an exact K-NN graph needs $O(n^2)$ time, thereby we consider an AKNN graph. NNDescent [15] is a state-of-the-art algorithm that builds an AKNN graph in any metric spaces. We first introduce it. (Note that the AKNN graph obtained by NNDescent satisfies only property 1.)

NNDescent. This algorithm is based on the idea that, given an object p and its similar object p', similar objects of p' would be similar to p. That is, approximate K-NNs of p can be obtained by accessing its similar objects and their similar ones iteratively. Given K and P, the specific operations of NNDescent[3] are as follows:

(1) For each object $p \in P$, NNDescent first chooses K random objects as its initial AKNNs.
(2) For each object $p \in P$, NNDescent obtains a similar object list that contains its AKNNs and reverse AKNNs. (If $p \in$ AKNNs of p', p' is a reverse AKNN of p, thereby how to obtain reverse AKNNs is trivial.) Given $p \in P$ and the objects p' in the similar object list of p, NNDescent accesses the similar object list of p'. If the list contains objects with smaller distances to p than those to the current AKNN of p, NNDescent updates its AKNNs.
(3) NNDescent iteratively conducts the above procedure until no updates occur (or a fixed iteration times).

Theorem 2. *NNDescent requires $O(nK^2 \log K)$ time.*

Drawbacks of NNDescent. The accuracy of the AKNN graph built by NNDescent is empirically high, but it has the following:

- The initial completely random links incur many AKNN updates in the second operation. Due to this initialization, each object cannot have links to its similar objects in an early stage, incurring unnecessary distance computations.
- The similar object list of p' is redundantly accessed even when the list has no updates from the previous iteration.

We overcome them by NNDescent+, an extension of NNDescent. This is of independent interest for building an AKNN graph.

NNDescent+. We overcome the first drawback by utilizing data partitioning that clusters similar objects and do the second drawback by maintaining the update status of similar object lists.

Initialization by VP-tree based partitioning. Each object needs to find its (approximate) K-NNs quickly, to reduce the number of update iterations. We achieve this by utilizing a VP-tree based partitioning approach.

Given an object set P, a VP-tree for P is built by recursive partitioning. Specifically, consider that a node of the VP-tree has P. If a node contains more objects than the capacity c, this node (or P) is partitioned into two nodes, left and right. (Otherwise, this node is a leaf node.) Let p be a randomly chosen object from P. The partitioning algorithm computes the distances between p and the other objects in P, sorts the distances, and obtains the mean distance μ. If an object $p' \neq p$ has $dist(p, p') \leq \mu$, it is assigned to the left child of p. Otherwise, it is assigned to the right one. This partition is repeated until no nodes can be partitioned.

We set $c = O(K)$. Consider a leaf node that is the left node of its parent. Let P' be the set of objects held by this leaf node. Objects in P' tend to be similar to each other, due to the ball-based partitioning property. Therefore, for each $p \in P'$, we set its K-NNs in P' as its initial AKNNs. This approach can have much more accurate AKNNs at the initialization stage than the random-based one. Besides, the efficiency of NNDescent is not lost.

Lemma 2. *NNDescent+ needs $O(nK^2 \log K)$ time at its initialization.*

Because of the random nature, some objects cannot be contained in P'. We hence do this partitioning a constant number of times. (For objects that could not be contained in P' after repeating the partitioning, random objects are set as their AKNNs.) It is also important to note that nodes, whose left child is a leaf node, are set as *pivots*, which are utilized in future steps. The ball-based partitioning makes pivots being distributed in each subspace of the given data space. This is also the reason why we use this partitioning approach. Note that we have $o(n)$ pivots. Algorithm 3 summarizes our initialization approach, and NNDescent+ replaces the first operation of NNDescent with Algorithm 3.

Skipping similar object lists with no updates. When obtaining the similar object list of an object p, NNDescent+ adds objects p',

[3]We consider the basic version of NNDescent in [15], because it is parallel-friendly (almost no synchronization).

Algorithm 3: PARTITION

```
Input: A set of objects P' ⊆ P
1  if |P'| > c then
2      p ← a randomly chosen object from P', D ← ∅
3      for each p' ∈ P' do
4          D ← ⟨dist(p, p'), p'⟩
5      μ ← the mean distance in D
6      L ← ∅, R ← ∅
7      for each p' ∈ P' do
8          if dist(p, p') ≤ μ then
9              L ← L ∪ {p'}
10         else
11             R ← R ∪ {p'}
12     PARTITION(L), PARTITION(R)
13     if |L| ≤ c then
14         Set p as a pivot
15 else
16     if P' = L then
17         for each p' ∈ P' do
18             Update AKNN of p' from P'
```

which are AKNNs or reverse AKNNs of p, to the similar object list iff AKNNs of p' have been updated in the previous iteration. We employ a hash table to maintain the AKNN update status of each object. The space complexity of this hash table is thus $O(n)$, and confirmation of the update status needs $O(1)$ amortized time for each object. Therefore, NNDESCENT+ reduces the cost of the second operation of NNDESCENT.

Exact K'-NN Retrieval. The above initialization and skipping approaches respectively reduce the number of iterations and unnecessary distance computations. However, for objects p such that their K-NNs are relatively far from p, the initialization may provide inaccurate results. The initialization approach clusters objects with small distances and is difficult to cluster objects with K-NNs that have large distances to them. This may generate biased clusters, which derives biased similar object lists in the second procedure. If this occurs, reachability to accurate K-NNs is degraded, and p may suffer from this. To alleviate this, NNDESCENT+ computes the exact K-NNs for such objects.

After the iterative AKNN updates (the third procedure in NNDESCENT), NNDESCENT+ sorts objects in P in descending order of the sum of the distances to their (approximate) K-NNs. If the sum is large, it is perhaps inaccurate. NNDESCENT+ picks the first m objects and retrieves their exact K'-NNs, where $K' \geq K$ is sufficiently large (but $K' \ll n$). We present why we use K' in Section 5.5. Note that m is a constant and $m \ll n$. Therefore, this approach incurs $O(n(K + \log n))$ time.

Now, we see the time complexity of NNDESCENT+.

LEMMA 3. *NNDESCENT+ requires $O(nK^2 \log K)$ time.*

Although NNDESCENT+ theoretically requires the same time as NNDESCENT, NNDESCENT+ is empirically faster (in most cases), because of reducing the number of iterations and pruning unnecessary similar object lists of neighbors. In addition, the procedure of NNDESCENT+ (except for obtaining reverse AKNNs) can exploit multi-threading (by using parallel for and parallel sort).

5.2 Connecting Sub-Graphs

Since $K \ll n$, an AKNN graph may have some disjoint sub-graphs. If this holds for the AKNN graph built by NNDESCENT+, GREEDY-COUNTING may not be able to traverse some of neighbors. We therefore make MRPG strongly connected[4]. Algorithm 4 details our approach CONNECT-SUBGRAPHS that consists of two phases.

Reverse AKNN phase (lines 1–3). In the first phase, we consider reverse AKNNs. Specifically, if an object p is included in AKNNs of p', p creates a link to p' (if p does not have it). The AKNN graph built by NNDESCENT+ is a directed graph. This phase converts it to an undirected graph. Although this is simple, reachability between objects and their neighbors can be improved, because reverse AKNNs of each object are (probably) similar to it.

BFS with ANN phase (lines 4–22). In the second phase, we propose a randomized approach that exploits breadth-first search (BFS) and ANN search on an AKNN graph. We confirm the connection between any two objects through BFS (from a random object). If BFS did not traverse some objects (line 14), the AKNN graph has some disjoint sub-graphs.

Let P' be a set of objects that have not been traversed by BFS. We make a path between a pivot in P' and a pivot in $P \backslash P'$. Let v'_{piv} be a random pivot in P'. Also, let V_{piv} be a set of random pivots in $P \backslash P'$ (note that $|V_{piv}|$ is a small constant). We search for an ANN object for v'_{piv} among $P \backslash P'$ and create links between them (lines 18–22). Since pivots are distributed uniformly in each subspace, this approach creates links between objects with small distances as much as possible, which is the behind idea of this phase.

To find an ANN, we employ the greedy algorithm proposed in [26]. The inputs of this algorithm are, a query object (v'_{piv}), a starting object ($v \in V_{piv}$), and a proximity graph. Given v, this algorithm traverses objects in $v.E$, computes the object v' with the minimum distance to v'_{piv}, goes to v', and repeats this until we cannot get closer to v'_{piv}. Let v_{ann} be the answer to this algorithm. We conduct this search for each $v \in V_{piv}$, select the object v_{res} with the minimum distance to v'_{piv}, and create links between v'_{piv} and v_{res}. Then, we re-start BFS from a random object in P' (already traversed objects are skipped). The above operations are repeated until BFS traverses all objects.

EXAMPLE 2. *Figure 3 illustrates an example of CONNECT-SUBGRAPHS. Figure 3(a) shows the AKNN graph obtained by NNDescent+ ($K' = K$ for ease of presentation). BFS has traversed the red-marked vertices, and now we conduct an ANN search, where the query and starting objects are respectively v_{piv} and v. The ANN search traverses the grey arrows (each traversed vertex selects the vertex that is the closest to v_{piv}) and obtains v_{res}. We then create a link between v_{piv} and v_{res}, as illustrated in Figure 3(b). After that, we re-start BFS from a random vertex, e.g., v', in Figure 3(b), that has not been traversed yet.*

[4] A similar idea has been proposed in [17], but how to add links to make a proximity graph strongly connected is different from our approach. In addition, [17] does not have a theoretical time bound to achieve it.

Algorithm 4: CONNECT-SUBGRAPHS

```
   Input: G
1  for each p ∈ P do
2      for each (v, v') ∈ v.E such that v' ∉ K'-NN do
3          v'.E ← v'.E ∪ {v}
4  P' ← P
5  while P' ≠ ∅ do
6      Q ← a random node (object) v (p) in P'
7      P' ← P'\{p}
8      while Q ≠ ∅ do
9          v ← the front of Q
10         Q ← Q\{v}
11         for each v' ∈ v.E do
12             if p' ∈ P' then
13                 P' ← P'\{p'}, Q ← Q ∪ {v'}
14     if P' ≠ ∅ then
15         v'_piv ← a random pivot in P'
16         V_piv ← a set of random pivots in P\P'
17         dist_min ← ∞, v_res ← v'_piv
18         for each v ∈ V_piv do
19             v_ann ← ANN-SEARCH(v, v'_piv, G)
20             if dist(v_ann, v'_piv) < dist_min then
21                 dist_min ← dist(v_ann, v'_piv), v_res ← v_ann
22         v'_piv.E ← v'_piv.E ∪ {v_res}, v_res.E ← v_res.E ∪ {v'_piv}
```

We set the maximum hop count for the ANN search. (It should be sufficiently large to make MRPG strongly connected, and is 10 in our implementation). This yields that the time complexity of this algorithm is $O(K)$ (since $|V_{piv}| = O(1)$). Then we have:

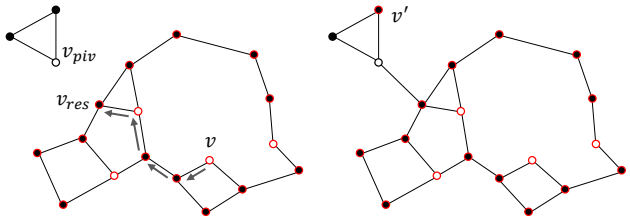
LEMMA 4. *CONNECT-SUBGRAPHS requires $O(nK)$ time.*

5.3 Removing Detours

If a path from an object p to its neighbor p' is not monotonic (i.e., it is a detour), GREEDY-COUNTING may not be able to access p'. For example, consider two objects p_1 and p_2 where $dist(p_1, p_2) \leq r$. Assume that there is only a single path between p_1 and p_2, e.g., $p_1 \rightarrow p_3 \rightarrow p_2$. If $dist(p_1, p_3) > r$, GREEDY-COUNTING cannot reach p_2 from p_1. This increases the number of false positives, so we consider making monotonic paths. We first demonstrate that making a monotonic search graph (MSG) is not practical. Then, we propose a pivot-based approximation.

Building a MSG. Theoretically, building a MSG needs $\Omega(n^2)$ time, because we have to check a path between all object pairs in P. We propose GET-NON-MONOTONIC(), which is based on BFS and searches for objects with no monotonic paths for a given object, to make a MSG.

GET-NON-MONOTONIC(). Given p_1, this function conducts BFS from p_1. Assume that we now access p_3 during BFS and BFS traversed a path $p_1 \rightarrow p_2 \rightarrow p_3$. If $dist(p_1, p_2) > dist(p_1, p_3)$, this path is a detour, so we need a monotonic path. We maintain objects, such that a monotonic path from p_1 to them could not be confirmed by BFS, and distances to them in an array A_1. After all objects are traversed, we sort A_1 in ascending order of distance.



(a) ANN search (b) Re-starting BFS

Figure 3: Example of CONNECT-SUBGRAPHS. White vertices represent pivots. BFS traversed red-marked vertices.

We conduct this function for each object. Now we have an array A_i for each object. We add a link between $A_i[j]$ and $A_i[j+1]$ for each $j \in [1, s-1]$, where s is the size of A_i. ($A_i[1]$ is linked to p_i.) This approach guarantees that a given proximity graph becomes a MSG. However, a huge cost is incurred.

THEOREM 3. *Building a MSG needs $O(n^2(K + \log n))$ time.*

Approximation by heuristic. This theorem proves that building a MSG is not practical. Note that it is not necessary to make monotonic paths between any two objects, because r and k are generally small [22, 31, 36]. It is thus important to retain monotonic paths to objects with small distances in practice. From this observation, we propose a heuristic that creates links between similar objects. In addition to the observation, our heuristic utilizes the following observations: (i) an AKNN graph has a property that similar objects of an object p tend to exist within a small hop count from p, and (ii) given p and its similar object p', similar objects of p' tend to be similar to p (i.e., the idea of NNDESCENT). That is, our heuristic is based on the idea: we can create necessary links for p if we traverse such objects appearing in observations (i) and (ii).

Algorithm 5 describes our heuristic. Line 1 samples $|P'|$ objects as target for making monotonic paths (we do not choose object with links to exact K'-NNs). Pivots are weighted for this sampling, since GREEDY-COUNTING traverses pivots. For each $p \in P'$, we do the following:

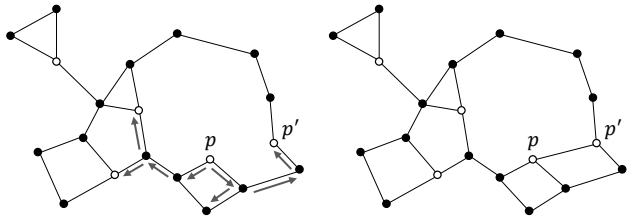
(1) We conduct 3-hop BFS from p (which terminates traversal when the hop count of the current object from p is 3), to obtain objects with no monotonic path from p (line 4). This corresponds to GET-NON-MONOTONIC() with a hop count constraint, and the objects obtained are maintained similarly.
(2) We sample $|P_{piv}|$ pivots with small distances to p (pivots existing within one hop from p and/or having their exact K'-NNs are not sampled). Then, for each $p' \in P_{piv}$, 2-hop BFS from p' is done, and we obtain objects with no monotonic path from p (lines 5–7). That is, BFS starts from p' but computes distances between p and the objects within two hops from p'.

After that, we create necessary links, similar to MSG building (lines 8–9). (We can increase the above hop counts to improve the accuracy, but the computational cost becomes significantly larger, which can be observed from Lemma 5.)

EXAMPLE 3. *We present an example of Algorithm 5. Figure 4(a), which shows the proximity graph obtained in Example 2, depicts 3-hop BFS.*

Algorithm 5: Remove-Detours

```
Input: G
P' ← a set of randomly chosen objects
P_non ← ∅
for each p ∈ P' do
    P_non ← P_non ∪ Get-Non-Monotonic(p, p, 3, G)
    P_piv ← a set of randomly chosen pivots
    for each p' ∈ P_piv do
        P_non ← P_non ∪ Get-Non-Monotonic(p, p', 2, G)
for each ⟨p, p'⟩ ∈ P_non do
    Create links between p and p'
```



(a) 3-hop BFS (b) Removing a detour

Figure 4: Example of Remove-Detours

For ease of presentation, assume $P' = \{p\}$, and 3-hop BFS is conducted from p. We see that the path from p to p' is a detour, i.e., is not a monotonic path. After sampling pivots near p and 2-hop BFS from them (not described here), we have $A = \{p'\}$. Hence we add a link between p and p', as shown in Figure 4(b).

Note that we set $|P'| = O(\frac{n}{K})$ and $|P_{piv}| = O(K)$. Recall that Get-Non-Monotonic() maintains A, and we limit the size of A so that $|A|$ is at most $O(K^2)$ by maintaining only objects with the smallest distances to p. Then, we have:

Lemma 5. *Remove-Detours needs $O(nK^2 \log K)$ time.*

5.4 Removing Links

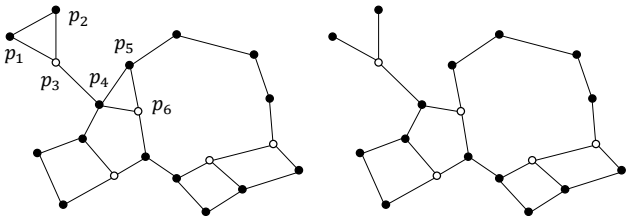
Since each object has links to its similar objects, p_1 and p_2, which is connected to p_1, may have links to other common objects, say p_3. If p_1 and p_2 are traversed by Greedy-Counting, p_3 is accessed at least two times. If there are many common links between objects within one hop, redundant accesses are incurred many times. To reduce them, Remove-Links removes links based on pivots.

If a non-pivot object p has a link to a pivot p', we remove links to common objects between p and p'. We do this link removal for each non-pivot object. (Because of this removal, lines 13–14 of Algorithm 2 are necessary.)

Example 4. *Figure 5 presents an example of Remove-Links. We use the graph obtained in Example 3. Two non-pivot objects p_1 and p_2 in Figure 5(a) respectively have a link to a common pivot p_3. Objects p_4, p_5, and p_6 have the same case. Therefore, links (p_1, p_2) and (p_4, p_5) are removed, then we have an MRPG shown in Figure 5(b).*

By using hash-based link management, it is trivial to see that

Lemma 6. *Remove-Links incurs $O(nK)$ time.*



(a) Checking common pivots (b) Removing unnecessary links

Figure 5: Example of Remove-Links

5.5 Discussion

From Lemmas 3–6, we see that:

Theorem 4. *We need $O(nK^2 \log K)$ time to build a MRPG.*

In addition,

Theorem 5. *The space complexity of a MRPG is $O(nK)$.*

In MRPG, there are objects that have links to their exact K'-NNs, and these objects have a larger distance to their approximate K-NNs compared with the other objects in P. It can be intuitively seen that these objects tend to be outliers for any (reasonable) r. Assume that p has links to its exact K'-NNs. If $K' \geq k$, we can evaluate whether p is outlier or not in $O(k)$ time, by traversing only its links. That is, if the count does not reach k, we can accurately determine that p is an outlier without verification, which reduces t in Theorem 1. For such objects p, we replace lines 4–5 of Algorithm 1 with the above operation. By setting a sufficiently large integer as K', when k is reasonable, MRPG detects outliers very quickly. (If $k > K'$, MRPG utilizes the original Algorithm 1 to keep correctness, so it does not lose generality). As analyzed in Section 4, the main cost of online processing is the verification cost. Therefore, reducing this cost from $O(n)$ to $O(k)$ yields significant efficiency improvement.

6 EXPERIMENTS

This section reports our experimental results. Our experiments were conducted on a machine with dual 12-core Intel Xeon E5-2687w v4 processors (3.0GHz) that share a 512GB RAM. This machine can run at most 48 threads by using hyper-threading. All evaluated algorithms were implemented in C++ and compiled by g++ 7.4.0 with -O3 flag. We used OpenMP for multi-threading.

Datasets. We used seven real datasets, Deep [6], Glove[5], HEPMASS[6], MNIST[7], PAMAP2[8], SIFT[9], and Words[10]. (For MNIST, we randomly sampled 3 million objects from the original dataset.) Table 1 summarizes their statistics and distance functions we used[11]. We normalized PAMAP2, so that the domain of each dimension is $[0, 10^5]$. We observed that the distance distribution of SIFT follows Gaussian mixture distribution and that of the other datasets follows Gaussian distribution.

[5] https://nlp.stanford.edu/projects/glove/
[6] https://archive.ics.uci.edu/ml/datasets/HEPMASS
[7] https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/multiclass.html
[8] https://archive.ics.uci.edu/ml/datasets/PAMAP2+Physical+Activity+Monitoring
[9] http://corpus-texmex.irisa.fr/
[10] https://github.com/dwyl/english-words
[11] We have updated the implementation for computing L_4-norm, and the experimental results on MNIST have also been updated from [3].

Table 1: Datasets

Dataset	Cardinality	Dim.	Distance function
Deep	10,000,000	96	L_2-norm
Glove	1,193,514	25	Angular distance
HEPMASS	7,000,000	27	L_1-norm
MNIST	3,000,000	784	L_4-norm
PAMAP2	2,844,868	51	L_2-norm
SIFT	1,000,000	128	L_2-norm
Words	466,551	1–45	Edit distance

Table 2: Default parameters

Dataset	r	k	Outlier ratio
Deep	0.93	50	0.62%
Glove	0.25	20	0.55%
HEPMASS	15	50	0.65%
MNIST	600	50	0.34%
PAMAP2	50,000	100	0.61%
SIFT	320	40	1.04%
Words	5	15	4.16%

Algorithms. We evaluated the following exact algorithms:

- State-of-the-art: *Nested-loop* [8], *SNIF* [30], *DOLPHIN* [4], and *VP-tree* [35], which are introduced in Section 3.
- Proximity graph-based algorithm: *NSW* [26], *KGraph* [15], *MRPG-basic*, and *MRPG*. MRPG-basic is a variant of MRPG, and, in NNDescent+, we compute the exact K-NNs for some objects, instead of K'-NNs. Therefore, by comparing MRPG with MRPG-basic, the efficiency of optimizing the verification is understandable. For outlier detection with NSW and KGraph, we used Algorithms 1 and 2 without lines 13–14 of Algorithm 2. They used the same verification phase as MRPG. We employed a VP-tree in the verification phase, i.e., Exact-Counting, on HEPMASS, PAMAP2, and Words.

We followed the original papers to set the system parameters in the state-of-the-art. For KGraph, MRPG-basic, and MRPG on PAMAP2, we set $K = 40$, and we set $K = 25$ for the other datasets. The number of links for each object in NSW is set so that its memory is almost the same as that of KGraph. For MRPG, we set $K' = 4 \times K$. Codes are available in a GitHub repository[12].

We set 12 and 8 hours as time limit for pre-processing (offline time) and outlier detection (online time), respectively. In cases that algorithms could not terminate pre-processing or detect all outliers within the time limit, we represent NA as the result.

Parameters. Table 2 shows the default parameters. They were specified so that the outlier ratio is small [12, 36] or clear outliers are identified[13]. We confirmed that the number of neighbors in each dataset follows power law and most objects have many neighbors. We used 12 (48) threads as the default number of threads for outlier detection (pre-processing). For outlier detection on Deep and MNIST, we used 48 threads, because they need time to be processed (due to large n and dimensionality).

[12]https://github.com/amgt-d1/DOD

[13]If objects have a small number of neighbors for a reasonable r such that (most of) the other objects have enough or many neighbors, they are clear outliers. We provided such r and k.

Table 3: Pre-processing time [sec]

Dataset	NSW	KGraph	MRPG-basic	MRPG
Deep	NA	20079.80	13417.40	17230.30
Glove	2333.47	923.83	755.54	791.53
HEPMASS	NA	7935.25	4345.63	5221.86
MNIST	33368.0	2972.96	1566.05	2281.55
PAMAP2	4522.14	1325.40	729.54	888.61
SIFT	4910.94	929.48	723.75	817.33
Words	871.27	455.15	707.08	868.62

Table 4: Decomposed time of pre-processing on Glove [sec]

Algorithm	KGraph	MRPG-basic	MRPG
NNDescent(+)	923.83	464.34	474.20
Connect-SubGraphs	-	20.36	24.28
Remove-Detours	-	278.21	271.41
Remove-Links	-	19.44	19.61

6.1 Evaluation of Pre-processing

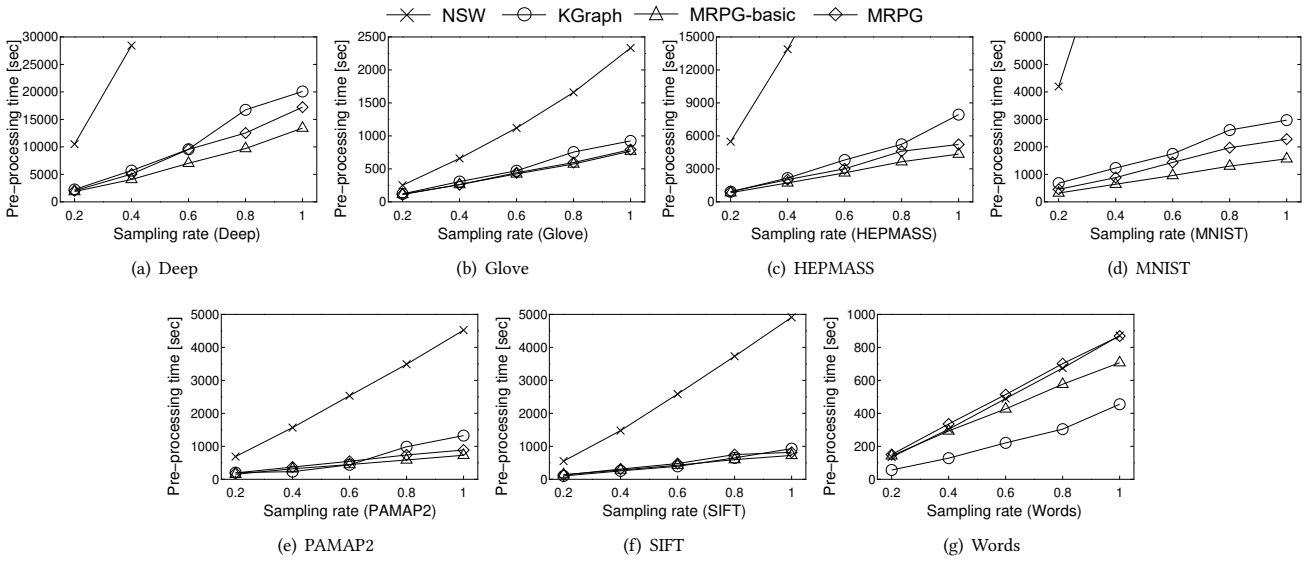
We first evaluate the pre-processing efficiencies of NSW, KGraph, MRPG-basic, and MRPG. Nested-loop, SNIF, and DOLPHIN do not have a pre-processing phase, whereas building a VP-tree took less than 310 seconds for each dataset.

MRPG(-basic) vs. KGraph. Table 3 presents the pre-processing time of each proximity graph at the default parameters. In most cases, building a MRPG-basic is the most efficient and building a MRPG is also more efficient than building a KGraph. This result is derived from the efficiency of NNDescent+. We depict the decomposed time of building a KGraph, MRPG-basic, and MRPG on Glove in Table 4, as an example. This table shows that NNDescent+ is faster than NNDescent, demonstrating the effectiveness of the VP-tree based partitioning approach and the skipping approach. Also, the other functions for building a MRPG do not incur significant costs. These provide a high performance for building a MRPG.

One exception appears in the Words case. We used edit distance for Words, and this distance function needs a large computational cost for objects with large dimensionality. We observed that objects, whose exact K'-NNs are computed, have large dimensionality, thereby exact K'-NN computation incurs a long time.

MRPG vs. MRPG-basic. Building a MRPG needs longer time than building a MRPG-basic. This is because, for some objects, we compute their K'-NNs where $K' > K$, during building a MRPG. That is, NNDescent+ for MRPG incurs longer time than that for MRPG-basic, as Table 4 presents.

NSW vs. the other proximity graphs. Table 3 shows that building a NSW consistently needs longer time than building the other proximity graphs. Because the NSW building algorithm is based on incremental object insertion, building a NSW cannot use multi-threads. This property lacks the scalability to large datasets and ones with large dimensionality. Therefore, NSW cannot be built on Deep and HEPMASS within a half day.



(a) Deep (b) Glove (c) HEPMASS (d) MNIST

(e) PAMAP2 (f) SIFT (g) Words

Figure 6: Impact of n (pre-processing time)

Table 5: Running time [sec]. Bold shows the winner.

Dataset	Nested-loop	SNIF	DOLPHIN	VP-tree	NSW	KGraph	MRPG-basic	MRPG
Deep	NA	NA	NA	NA	NA	8616.10	5474.10	**1966.17**
Glove	1045.47	1222.43	9277.89	1398.92	147.00	83.82	56.80	**2.63**
HEPMASS	17295.40	20360.80	NA	8597.23	NA	780.19	638.83	**38.40**
MNIST	15494.00	22579.80	NA	13836.60	1630.25	1702.57	1264.26	**918.91**
PAMAP2	422.40	1213.56	1819.90	208.55	22.16	23.77	18.16	**10.55**
SIFT	1427.74	1507.58	8684.08	2005.27	200.89	175.88	144.11	**11.32**
Words	1844.65	2086.50	7061.50	1021.39	498.34	393.95	374.08	**2.67**

Scalability test. Figure 6 illustrates the scalability to n when building the proximity graphs. We varied the size of P by random sampling (i.e., we varied sampling rate).

As discussed, NSW basically needs (much) larger time for building. This algorithm is competitive only in the case of Words, because its cardinality is smaller than the other datasets. KGraph, MRPG-basic, and MRPG have linear scalability to n, due to Theorems 2 and 4. Notice that MRPG scales better in most cases.

6.2 Evaluation of DOD Algorithms

We next evaluate outlier detection algorithms. Tables 5 and 6 describe the running time and index size of each algorithm, respectively. We did not test algorithms whose index could not be obtained within the time limit.

Our approach vs. state-of-the-art. Let us compare our approach, proximity graph-based solution (NSW, KGraph, MRPG-basic, and MRPG), with state-of-the-art (Nested-loop, SNIF, DOLPHIN, and VP-tree). Table 5 shows that our approach is clearly faster than the state-of-the-art, demonstrating its robustness to the distance functions listed in Table 1. This result is derived from the reduction of unnecessary distance computation. Specifically, in our approach (or a proximity graph), each object has links (or paths) to its neighbors. This yields efficient early termination, i.e., inliers are quickly identified. For example, MRPG is 397.5, 223.9, 15.1, 19.8, 126.1, and 382.5 times faster than the best algorithm among the state-of-the-art on Glove, HEPMASS, MNIST, PAMAP2, SIFT, and Words, respectively. The state-of-the-art could not detect outliers within the time limit on Deep (largest dataset), whereas MRPG and MRPG-basic successfully deal with it. Also, we see that *MRPG provides a significant speed-up* by sacrificing a bit longer pre-processing time than MRPG-basic. This speed-up is derived from the reduction of the verification cost by detecting (some) outliers in the filtering phase (see Section 5.5).

Table 6 shows that our approach needs a larger index size than the state-of-the-art (Nested-loop does not build an index, so its index size is 0). However, its index size is not significant, and recent main-memory systems afford to retain the proximity graph, as its space requirement is $O(nK)$.

MRPG(-basic) vs. the other proximity graphs. We next focus on the performances of the proximity graphs. Table 5 reports that MRPG is clear winner. Recall that, to make Algorithm 1 faster, we have to reduce the number of false positives f, as demonstrated in Theorem 1. Table 7 shows that MRPG and MRPG-basic reduce f more compared with KGraph and NSW, so we obtain faster running time than those of KGraph and NSW[14]. This fact demonstrates the effectiveness of monotonic paths, i.e., MRPG and MRPG-basic have a better reachability than the others. We notice that the performance difference between MRPG and KGraph is not significant on Deep

[14]We have corrected some errors in Table 7 from [3] (but this does not affect our claim in [3]).

Table 6: Index size [MB]

Dataset	Nested-loop	SNIF	DOLPHIN	VP-tree	NSW	KGraph	MRPG-basic	MRPG
Deep	0	NA	NA	324.35	NA	1405.94	5516.58	7350.83
Glove	0	13.26	69.14	54.86	188.62	167.91	460.48	438.76
HEPMASS	0	61.04	NA	265.39	NA	1195.35	2188.65	2450.84
MNIST	0	27.75	NA	117.80	417.95	404.29	589.08	591.27
PAMAP2	0	18.36	65.12	128.00	819.17	528.26	553.87	760.69
SIFT	0	8.10	47.00	39.99	157.58	140.54	433.48	489.14
Words	0	4.41	26.86	27.79	102.20	93.92	191.73	178.74

Table 7: Number of false positives after the filtering phase

Algorithm	NSW	KGraph	MRPG-basic	MRPG
Deep	NA	81,140	33,180	20,616
Glove	19,970	3,356	40	24
HEPMASS	NA	11,133	2,363	438
MNIST	7,079	4,698	2,509	2,061
PAMAP2	18,346	22,543	4,290	3,986
SIFT	4,899	2,513	585	51
Words	9,569	989	120	4

Table 8: Decomposed time of outlier detection on Glove [sec]

Algorithm	NSW	KGraph	MRPG-basic	MRPG
Filtering	1.28	0.86	2.43	1.98
Verification	147.00	82.96	57.03	0.65

and MNIST compared with the other datasets. In Deep, we observed that false positive objects of MRPG have only nearly k neighbors, which makes the early termination not function. In MNIST, we found that some objects having links to their exact K'-NNs are inliers and false positive objects have the same observation as with Deep[15]. The verification cost of outliers and false positives therefore still remains on them, as with the other proximity graphs. However, this can be alleviated by using additional CPUs (cores/threads), as shown in Figure 10.

Recall that each dataset follows a power law distribution w.r.t. the number of neighbors. If a dataset has many objects that are inliers but exist in sparse areas, f of MRPG tends to be large. This is because the reachability to their neighbors still tends to be lower than that to neighbors of dense objects. The number of inliers in sparse areas is affected by data distributions, so f between the datasets are different as in Table 7. For example, we observed that Deep is sparser than the other datasets[16], so its f is large.

Table 8 exhibits the time for filtering and verification on Glove. Due to the reachability, MRPG and MRPG-basic incur longer filtering time but this reduces the verification time the most. (This result is consistent for the other datasets.) Besides, the running time of MRPG is shorter than those of the other proximity graphs. This is due to the heuristic that objects, which would be outliers, have links to exact K'-NNs. They are usually outliers in real datasets and are identified as outliers when 1-hop links are traversed from them, so verification is not needed for them. This provides a (significant) speed-up, and MRPG is 1.3–140.1 times faster than MRPG-basic. Recall that, in most cases, MRPG needs less pre-processing time than the others. Therefore, in terms of computational performance, MRPG normally dominates the other proximity graphs.

As for index size, MRPG needs more memory than NSW and KGraph, because MRPG creates links to improve reachability. However, MRPG removes unnecessary links, so its index size is competitive with those of NSW and KGraph for datasets with skew, such as PAMAP2. The index size of MRPG is smaller than that of MRPG-basic on Glove and Words. This is also derived from the unnecessary link removal.

Effectiveness of Connect-SubGraphs and Remove-Detours. We evaluated (i) MRPG without Algorithms 4 and 5, (ii) MRPG without Algorithm 4, and (iii) MRPG without Algorithm 5, to investigate how they contribute to improving reachability. We here report the numbers of false positives only on PAMAP2 for the three variants of MRPG, because the result is consistent with those on the other datasets. The numbers of false positives provided by the first, second, and third variants are respectively 11937, 4712, and 9720. They are less than those of NSW and KGraph, see Table 7. This result verifies that Connect-SubGraphs is useful and Remove-Detours is important to improve reachability, i.e., provide fewer false positives. (Note that Remove-Links does not affect the number of false positives, since it does not improve reachability.)

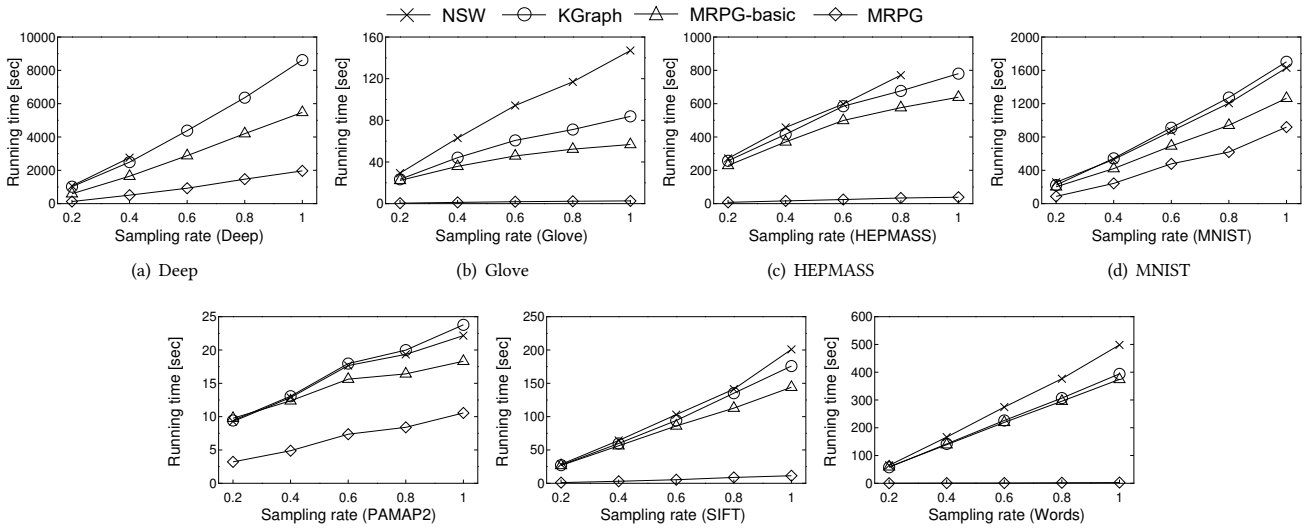
Varying n. Figure 7 studies the scalability of each proximity graph in the same way as in Figure 6 (parameters were fixed as the default ones.). Since Table 5 confirms the superiority of our approach over the state-of-the-art, we omit the results of the state-of-the-art.

As the sampling rate increases, the running time of each proximity graph becomes longer. This is reasonable, since both filtering and verification costs increase. We have three observations. The first one is that MRPG-basic keeps outperforming NSW and KGraph. Second, MRPG significantly outperforms the other proximity graphs. Last, MRPG and MRPG-basic scale better than the other proximity graphs, which confirms that pivot-based monotonic path creation provides their scalability.

In the case of Words, MRPG-basic and KGraph show similar performances. We observed that outliers in Words have large dimensionality. Because computing edit distance needs a quadratic cost to dimensionality, verification of outliers incurs a large computational cost. For example, with the default parameters, MRPG-basic (KGraph) took 2.43 (0.73) and 371.65 (393.23) seconds for filtering and verification, respectively. From the result in Table 7, we see that false positives in Words are verified quickly (by early termination)

[15] Although NSW has more f than that of KGraph, NSW is faster than KGraph on MNIST. We found that the false positives of NSW have more neighbors than those of KGraph, thus, for NSW, the early termination in the sequential scan functions, rendering its faster time.

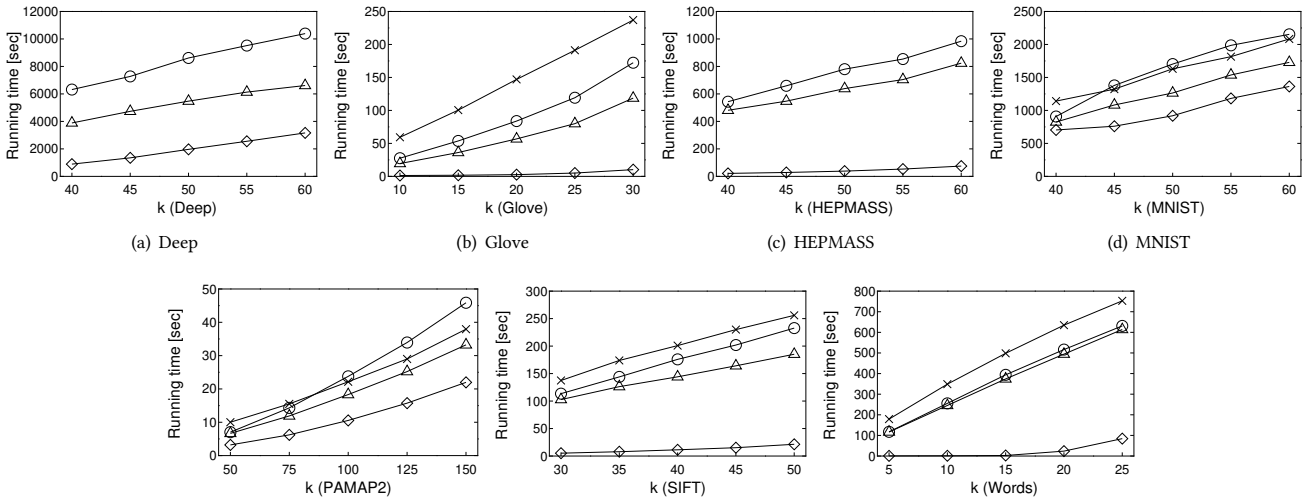
[16] The reasonable r of Deep is far from the mean of its distance distribution, compared with the other datasets.



(a) Deep (b) Glove (c) HEPMASS (d) MNIST

(e) PAMAP2 (f) SIFT (g) Words

Figure 7: Impact of n



(a) Deep (b) Glove (c) HEPMASS (d) MNIST

(e) PAMAP2 (f) SIFT (g) Words

Figure 8: Impact of k

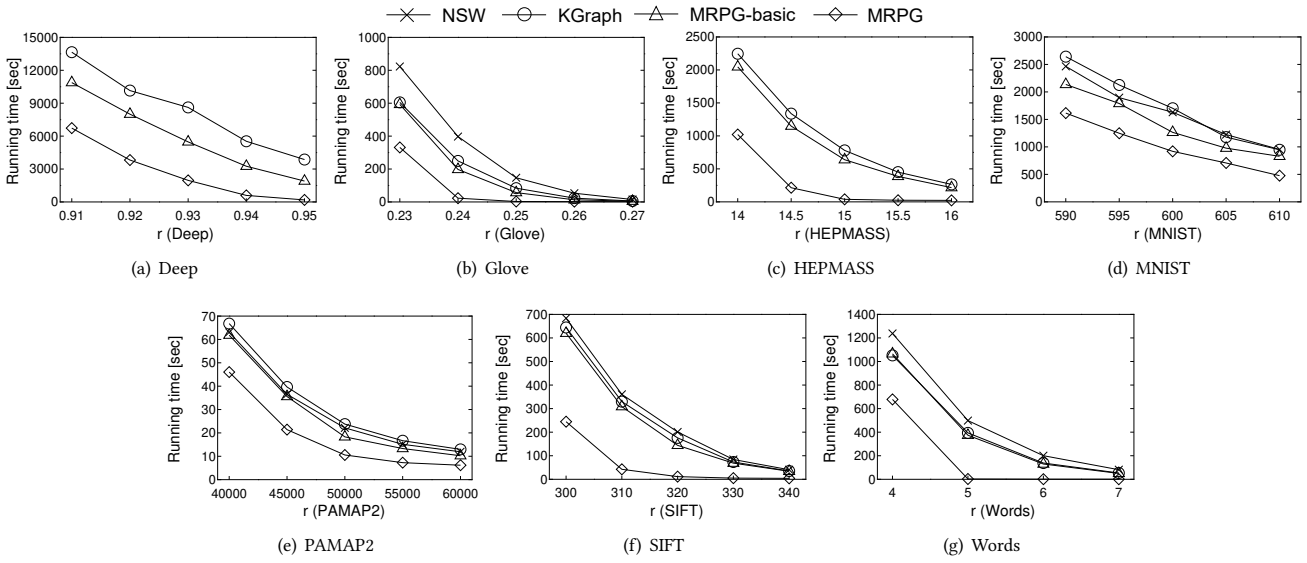
and the verification of outliers dominates the most computational time.

Varying k. We investigate the influence of outlier ratio by varying k. Figure 8 presents the results. As k increases, our approach needs to traverse more objects, rendering a larger filtering cost. In addition, as k increases, outlier ratio increases. Our approach hence needs more verification cost when k is large.

One difference between MRPG and the other proximity graphs is robustness to k, as MRPG(-basic) outperforms the other proximity graphs. This is derived from Connect-SubGraphs and Remove-Detours, i.e., functions that make MRPG different from KGraph. That is, the connectivity of the graph and the existence of monotonic paths (for similar objects) are important to exploit our algorithm.

Varying r. The result of experiments with varying distance threshold r is shown in Figure 9 (k is fixed at the default value). As r increases, the outlier ratio decreases, and vice versa. Similar to the results in Figure 8, MRPG keeps outperforming KGraph and NSW both when outlier ratio is high and low.

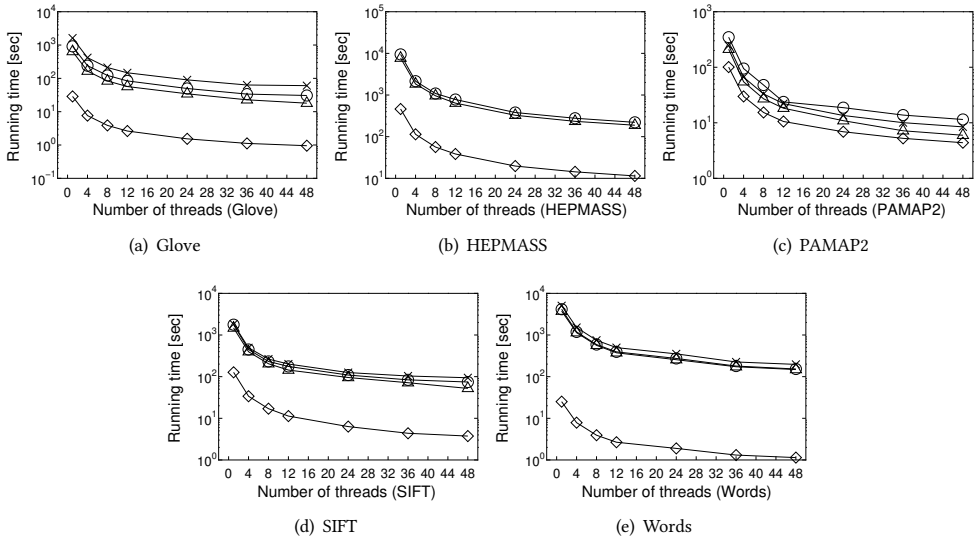
Varying the number of threads. Last, we demonstrate that our approach is parallel-friendly. Figure 10 shows the result on Glove, HEPMASS, PAMAP2, SIFT, and Words. We see that our solution exploits the available threads and has linear scalability to the number of threads, for each proximity graph. Also, the superiority among the proximity graphs does not change.



(a) Deep (b) Glove (c) HEPMASS (d) MNIST

(e) PAMAP2 (f) SIFT (g) Words

Figure 9: Impact of r



(a) Glove (b) HEPMASS (c) PAMAP2

(d) SIFT (e) Words

Figure 10: Impact of the number of threads

7 CONCLUSION

In this paper, we addressed the problem of distance-based outlier detection in metric spaces and proposed a novel approach, namely proximity graph-based algorithm. To exploit our DOD algorithm, we devised MRPG (Metric Randomized Proximity Graph), which improves reachability to neighbors and reduces the verification cost. Our experiments on real datasets confirm that (i) our DOD algorithm is much faster than state-of-the-art and (ii) MRPG is superior to existing proximity graphs.

ACKNOWLEDGMENTS

This research is partially supported by JSPS Grant-in-Aid for Scientific Research (A) Grant Number 18H04095, JST CREST Grant Number J181401085, and JST PRESTO Grant Number JPMJPR1931.

REFERENCES

[1] Charu C Aggarwal. 2015. Outlier Analysis. In *Data Mining*. 237–263.
[2] Daichi Amagata and Takahiro Hara. 2021. Fast Density-Peaks Clustering: Multicore-based Parallelization Approach. In *SIGMOD*.
[3] Daichi Amagata, Makoto Onizuka, and Takahiro Hara. 2021. Fast and Exact Outlier Detection in Metric Spaces: A Proximity Graph-based Approach. In *SIGMOD*. xxx–xxx.
[4] Fabrizio Angiulli and Fabio Fassetti. 2009. Dolphin: An Efficient Algorithm for Mining Distance-based Outliers in Very Large Datasets. *ACM Transactions on Knowledge and Data Discovery* 3, 1 (2009), 4.

[5] Sunil Arya and David M Mount. 1993. Approximate Nearest Neighbor Queries in Fixed Dimensions.. In *SODA*, Vol. 93. 271–280.
[6] Artem Babenko and Victor Lempitsky. 2016. Efficient Indexing of Billion-scale Datasets of Deep Descriptors. In *CVPR*. 2055–2063.
[7] Gustavo EAPA Batista, Ronaldo C Prati, and Maria Carolina Monard. 2004. A Study of the Behavior of Several Methods for Balancing Machine Learning Training Data. *SIGKDD Explorations Newsletter* 6, 1 (2004), 20–29.
[8] Stephen D Bay and Mark Schwabacher. 2003. Mining Distance-based Outliers in Near Linear Time with Randomization and a Simple Pruning Rule. In *KDD*. 29–38.
[9] Marian Boguna, Dmitri Krioukov, and Kimberly C Claffy. 2009. Navigability of Complex Networks. *Nature Physics* 5, 1 (2009), 74.
[10] Sergey Brin. 1995. Near Neighbor Search in Large Metric Spaces. In *VLDB*. 574–584.
[11] Guilherme O Campos, Arthur Zimek, Jörg Sander, Ricardo JGB Campello, Barbora Micenková, Erich Schubert, Ira Assent, and Michael E Houle. 2016. On the Evaluation of Unsupervised Outlier Detection: Measures, Datasets, and an Empirical Study. *Data Mining and Knowledge Discovery* 30, 4 (2016), 891–927.
[12] Lei Cao, Jiayuan Wang, and Elke A Rundensteiner. 2016. Sharing-aware Outlier Analytics over High-volume Data Streams. In *SIGMOD*. 527–540.
[13] Lu Chen, Yunjun Gao, Baihua Zheng, Christian S Jensen, Hanyu Yang, and Keyu Yang. 2017. Pivot-based Metric Indexing. *PVLDB* 10, 10 (2017), 1058–1069.
[14] DW Dearholt, N Gonzales, and G Kurup. 1988. Monotonic Search Networks for Computer Vision Databases. In *ACSSC*, Vol. 2. 548–553.
[15] Wei Dong, Charikar Moses, and Kai Li. 2011. Efficient k-nearest Neighbor Graph Construction for Generic Similarity Measures. In *WWW*. 577–586.
[16] Martin Ester, Hans-Peter Kriegel, Jörg Sander, Xiaowei Xu, et al. 1996. A density-based algorithm for discovering clusters in large spatial databases with noise.. In *KDD*. 226–231.
[17] Cong Fu, Chao Xiang, Changxu Wang, and Deng Cai. 2019. Fast Approximate Nearest Neighbor Search with the Navigating Spreading-out Graph. *PVLDB* 12, 5 (2019), 461–474.
[18] Ben Harwood and Tom Drummond. 2016. FANNG: Fast Approximate Nearest Neighbour Graphs. In *CVPR*. 5713–5722.
[19] Victoria Hodge and Jim Austin. 2004. A Survey of Outlier Detection Methodologies. *Artificial Intelligence Review* 22, 2 (2004), 85–126.
[20] Ihab F Ilyas and Xu Chu. 2019. *Data Cleaning*.
[21] Edwin M Knorr and Raymond T Ng. 1998. Algorithms for Mining Distance-based Outliers in Large Datasets. In *VLDB*, Vol. 98. 392–403.
[22] Maria Kontaki, Anastasios Gounaris, Apostolos N Papadopoulos, Kostas Tsichlas, and Yannis Manolopoulos. 2011. Continuous Monitoring of Distance-based Outliers over Data Streams. In *ICDE*. 135–146.
[23] Stefan Larson, Anish Mahendran, Andrew Lee, Jonathan K Kummerfeld, Parker Hill, Michael A Laurenzano, Johann Hauswald, Lingjia Tang, and Jason Mars. 2019. Outlier Detection for Improved Data Quality and Diversity in Dialog Systems. In *NAACL-HLT*. 517–527.
[24] Gilad Lerman and Tyler Maunu. 2018. An Overview of Robust Subspace Recovery. *Proc. IEEE* 106, 8 (2018), 1380–1410.
[25] Wen Li, Ying Zhang, Yifang Sun, Wei Wang, Mingjie Li, Wenjie Zhang, and Xuemin Lin. 2019. Approximate Nearest Neighbor Search on High Dimensional Data – Experiments, Analyses, and Improvement. *IEEE Transactions on Knowledge and Data Engineering* (2019).
[26] Yury Malkov, Alexander Ponomarenko, Andrey Logvinov, and Vladimir Krylov. 2014. Approximate Nearest Neighbor Algorithm based on Navigable Small World Graphs. *Information Systems* 45 (2014), 61–68.
[27] Yu A Malkov and DA Yashunin. 2020. Efficient and Robust Approximate Nearest Neighbor Search Using Hierarchical Navigable Small World Graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 42, 4 (2020), 824–836.
[28] Jeffrey Pennington, Richard Socher, and Christopher D Manning. 2014. Glove: Global Vectors for Word Representation. In *EMNLP*. 1532–1543.
[29] Martin Perdacher, Claudia Plant, and Christian Böhm. 2019. Cache-oblivious High-performance Similarity Join. In *SIGMOD*. 87–104.
[30] Yufei Tao, Xiaokui Xiao, and Shuigeng Zhou. 2006. Mining Distance-based Outliers from Large Databases in any Metric Space. In *KDD*. 394–403.
[31] Luan Tran, Liyue Fan, and Cyrus Shahabi. 2016. Distance-based Outlier Detection inData Streams. *PVLDB* 9, 12 (2016), 1089–1100.
[32] Luan Tran, Min Y Mun, and Cyrus Shahabi. 2020. Real-time Distance-based Outlier Detection in Data Streams. *PVLDB* 14, 2 (2020), 141–153.
[33] Hongzhi Wang, Mohamed Jaward Bah, and Mohamed Hammad. 2019. Progress in Outlier Detection Techniques: A Survey. *IEEE Access* 7 (2019), 107964–108000.
[34] Yisen Wang, Weiyang Liu, Xingjun Ma, James Bailey, Hongyuan Zha, Le Song, and Shu-Tao Xia. 2018. Iterative Learning with Open-set Noisy Labels. In *CVPR*. 8688–8696.
[35] Peter N Yianilos. 1993. Data Structures and Algorithms for Nearest Neighbor Search in General Metric Spaces. In *SODA*. 311–21.
[36] Susik Yoon, Jae-Gil Lee, and Byung Suk Lee. 2019. NETS: Extremely Fast Outlier Detection from a Data Stream via Set-based Processing. *PVLDB* 12, 11 (2019), 1303–1315.
[37] Hao Zhang, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, and Meihui Zhang. 2015. In-memory Big Data Management and Processing: A Survey. *IEEE Transactions on Knowledge and Data Engineering* 27, 7 (2015), 1920–1948.
[38] Vasileios Zois, Vassilis J Tsotras, and Walid A Najjar. 2019. Efficient Main-Memory Top-K Selection for Multicore Architectures. *PVLDB* 13, 2 (2019), 114–127.

A PROOFS

A.1 Proof of Lemma 1.

Given p_1, Greedy-Counting computes the distance between p_1 and p_2, only when p_2 is visited for the first time. Let p_3 be a neighbor of p_1. A proximity graph does not guarantee to have a path from p_1 to p_3 that can be traversed by Greedy-Counting. Therefore, the count (the number of neighbors of p_1) returned by Greedy-Counting is always k or less. As outliers have less than k neighbors, Greedy-Counting does not filter them. □

A.2 Proof of Theorem 1

Let ρ be the average number of accessed objects until Greedy-Counting terminates, for an object. The filtering phase incurs $O(\rho n)$ time. The verification phase incurs $O((f+t)n)$ time. Because ρ is small and often $\rho = O(k)$, we have $\rho \ll f + t$. Algorithm 1 hence requires $O((f + t)n)$ time. □

A.3 Proof of Theorem 2

The initial operation needs $O(nK \log K)$ time, because each object needs $O(K \log K)$ time to sort the random objects. To update the current AKNNs of an object p, NNDescent accesses the similar object list of p', where p' is one of the current AKNNs or reverse AKNNs of p. The amortized size of the similar object list of p' is $O(K)$, thus updating the current AKNN of p needs $O(K) \times K \times O(\log K) = O(K^2 \log K)$ time. That is, the second operation needs $O(nK^2 \log K)$ time. This is conducted iteratively, and the number of iterations is almost constant [15] (and can be fixed). The time complexity of NNDescent is then $O(nK^2 \log K)$. □

A.4 Proof of Lemma 2

Building a VP-tree needs $O(n \log n)$ time [10]. The number of leaf nodes, which are the left nodes of their parents, is $O(2^{\log n}) = O(n)$. This is because VP-tree is a balanced tree, i.e., its height is $O(\log n)$, and it has $2^{\log n}$ leaf nodes. Since each leaf node contains $O(K)$ objects, updating AKNNs of them needs $O(K^2 \log K)$ time. These facts conclude that this lemma holds. □

A.5 Proof of Lemma 3

Lemma 2 proves that the first procedure of NNDescent+ needs $O(nK^2 \log K)$ time. The second procedure of NNDescent+ is essentially the same as that of NNDescent. Therefore, it needs $O(nK^2 \log K)$ time. Exact K-NN retrieval, the last procedure of NNDescent+, needs $O(n(K + \log n))$ time. In total, the time complexity is $O(nK^2 \log K)$. □

A.6 Proof of Lemma 4

The reverse AKNN phase incurs $O(nK)$ time, since we scan all links. In the BFS with ANN phase, BFS needs $O(nK)$ time, as each object checks its links. The number of disjoint sub-graphs is at most $O(\frac{n}{K})$, since each object has at least K links. Our ANN search

incurs only $O(K)$ time, and lines 14–22 of Algorithm 4 need at most $O(\frac{n}{K}) \times O(K) = O(n)$ time. We now see that both the first and second phases need $O(nK)$ time, which proves this lemma. □

A.7 Proof of Theorem 3

BFS and sorting incur $O(nK)$ and $O(n \log n)$ time, respectively. We do these operations for each object, and $s \leq n$. Hence this theorem holds. □

A.8 Proof of Lemma 5

The proof of Theorem 3 suggests that 3-hop BFS needs $O(K^3 \log K^3)$ $= O(K^3 \log K)$ time for a target object. Similarly, 2-hop BFS needs $O(K^2 \log K)$ time, and this is done $O(K)$ times. We hence need $O(K^3 \log K) + O(K) \times O(K^2 \log K) = O(K^3 \log K)$ time for computing objects with no monotonic path from the target object. Since $|P'| = O(\frac{n}{K})$, lines 3–7 of Algorithm 5 need $O(\frac{n}{K}) \times O(K^3 \log K) = O(nK^2 \log K)$ time. The size of A_i is $O(K^2)$, so lines 8–9 of Algorithm 5 need $O(nK)$ time. Now we see that the lemma holds. □

A.9 Proof of Theorem 5

The AKNN graph built by NNDescent+ has $O(nK)$ links, and then links are added in Connect-SubGraphs and Remove-Detours. In Connect-SubGraphs, we add at most $O(\frac{n}{K})$ links, since the number of disjoint sub-graphs is at most $O(\frac{n}{K})$. On the other hand, in Remove-Detours, we add at most $O(\frac{n}{K}) \times O(K^2) = O(nK)$ links. As $\frac{n}{K} \ll nK$, a MRPG has at most $O(nK)$ links. □