The VLDB Journal (2022) 31:797–821
https://doi.org/10.1007/s00778-022-00729-1

REGULAR PAPER

Fast, exact, and parallel-friendly outlier detection algorithms with proximity graph in metric spaces

Daichi Amagata[1] · **Makoto Onizuka**[1] · **Takahiro Hara**[1]

Received: 24 June 2021 / Revised: 28 December 2021 / Accepted: 6 January 2022 / Published online: 27 January 2022

Abstract
In many fields, e.g., data mining and machine learning, distance-based outlier detection (DOD) is widely employed to remove noises and find abnormal phenomena, because DOD is unsupervised, can be employed in any metric spaces, and does not have any assumptions of data distributions. Nowadays, data mining and machine learning applications face the challenge of dealing with large datasets, which requires efficient DOD algorithms. We address the DOD problem with two different definitions. Our new idea, which solves the problems, is to exploit an in-memory proximity graph. For each problem, we propose a new algorithm that exploits a proximity graph and analyze an appropriate type of proximity graph for the algorithm. Our empirical study using real datasets confirms that our DOD algorithms are significantly faster than state-of-the-art ones.

Keywords Distance-based outlier detection · Metric space · Proximity graph

1 Introduction

Outlier detection is a fundamental task in many applications, such as fraud detection, health check, and noise data removal [5,17,50]. As described later, these applications often employ distance-based outlier detection (DOD) [32], because DOD is unsupervised, can be employed in any metric spaces, and does not have any assumptions of data distributions.

Motivation Many applications that implement classification, prediction, and regression often utilize machine learning techniques, because they can provide high accuracy. To train high performance machine learning models, noises (i.e., outliers) should be removed from training datasets, because the performances of models tend to be affected by outliers [5,37,51]. It is now common practice for many applications to remove noises as pre-processing for training [14,29], and DOD can contribute to this noise removal. Moreover, 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 errors or unique sentences from sentence embedding vectors [36]. Campos et al. confirmed that DOD successfully finds unhealthy people from medical data and irregular data from image datasets [19].

To cover these applications, DOD techniques need to be available in metric spaces. This is because the above applications can have many data types (e.g., multi-dimensional points, strings, and time-series), which exist not only in Euclidean spaces but also in other spaces. For instance, word embedding vectors usually exist in angular distance spaces [43]. We here note that DOD has several definitions. Let P be a set of objects.

1. (r, k)-DOD [32]: an object $p \in P$ is an outlier iff it has less than k other objects $p' \in P$ such that $dist(p, p') \leq r$, where $dist(p, p')$ is the distance between p and p'.
2. (N, k)-DOD-max [45]: an object $p \in P$ is an outlier iff the distance to its k-th nearest neighbor is ranked in the top-N among P.
3. (N, k)-DOD-avg [10]: an object $p \in P$ is an outlier iff the average distance to its k nearest neighbors is ranked in the top-N among P.

The above applications often manage large datasets, e.g., to train an accurate model, thereby techniques for these DOD

✉ Daichi Amagata
amagata.daichi@ist.osaka-u.ac.jp

Makoto Onizuka
onizuka@ist.osaka-u.ac.jp

Takahiro Hara
hara@ist.osaka-u.ac.jp

[1] Graduate School of Information Science and Technology, Osaka University, 1-5 Yamadaoka, Suita, Osaka, Japan

problems need to deal with large datasets [30]. There exist some algorithms that address the above DOD problems in metric spaces, but they are essentially based on nested-loop and suffer from $O(n^2)$ time, where $n = |P|$. The scalability issue therefore remains. This article aims at developing scalable algorithms for DOD in the first two definitions, because [19] demonstrates that (N, k)-DOD-avg returns almost the same result as that of (N, k)-DOD-max. Hereinafter, we use (N, k)-DOD to denote (N, k)-DOD-max.

Challenges Due to the popularization of main-memory databases [55], in-memory processing of DOD on a large dataset is possible. Exact DOD would be achieved efficiently by building an effective main-memory index offline. Existing metric DOD techniques [9,26,32,42,48] miss this observation. To design an efficient index-based solution for any metric spaces, we address the following challenges: general and effective index to any parameters (r, k) or (N, k), space efficiency, robustness to any metric spaces, and, for the (N, k)-DOD problem, quickly obtaining a tight threshold and upper-bound distance.

For the first challenge, we do not know the parameters in advance, so an index has to deal with any (r, k) or (N, k). Building an index that is general to these parameters and effectively supports fast and exact DOD is not trivial. The state-of-the-art algorithms for (r, k)-DOD [9,48] and for (N, k)-DOD [26] build a simple data structure in an online fashion after (r, k) or (N, k) are specified. The pruning efficiency of such indexes is limited, so they need a long time to detect outliers.

A trivial index for fast and exact DOD is to store a sorted array that maintains the distance to each object in P, for each $p \in P$. From this array, whether p is an outlier or not can be evaluated in $O(1)$ time for the (r, k)-DOD problem. This is because, if $dist(p, p') \leq r$ where p' is the k-th nearest neighbor of p, p is not an outlier. Also, checking the k-th element of the array for each object can solve the (N, k)-DOD problem in $O(n \log N)$ time. This approach, however, requires $O(n^2)$ space, so it is infeasible to employ this index and does not solve the second challenge.

Third, since we consider metric spaces and recent applications usually deal with middle or high dimensional data, robustness to any data types and dimensionality is important. Notice that we can employ (i) range queries to evaluate whether given objects are outliers for the (r, k)-DOD problem and (ii) k-NN queries to measure the distances to the k-th NN for the (N, k)-DOD problem. A simple and practical solution is to build a tree-based index offline and run a range or k-NN 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 space-partitioning techniques [9,48] degrade on high-dimensional data.

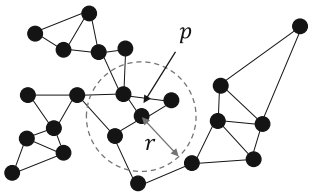


Fig. 1 Example of a proximity graph. Each object (black vertex) has links to its nearby objects, and r is a distance threshold. For $k = 3$, p is not an outlier

Last, the main bottleneck of the (N, k)-DOD problem is to compute the exact k-NNs. To avoid this as much as possible, we need (i) a technique that computes a tight threshold with a small cost and (ii) a technique that efficiently computes a tight upper-bound of the k-NN distance for each object. This is because, if an upper-bound of the k-NN distance of p is less than a given threshold, computing the exact k-NN distance of p can be safely pruned. However, these techniques are not trivial, since they have to overcome all of the first three challenges.

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

(1) *New algorithm for the (r, k)-DOD problem.* We propose a new solution for the (r, k)-DOD problem that filters non-outliers efficiently while guaranteeing correctness by exploiting a proximity graph. In a proximity graph, an object p corresponds to a vertex, and each object has links to some of its similar objects, as shown in Fig. 1, which assumes a Euclidean space. Example 1 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 Fig. 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 not-filtered objects. Note that our algorithm (i) is orthogonal to any metric proximity graphs, (ii) is easily parallelizable, and (iii) detects all outliers correctly.

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 (not-filtered objects in 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. This usually holds for real datasets, whereas the existing DOD

[1] Our preliminary version [7] solves the (r, k)-DOD problem. This journal article includes additional contents: we show that (1) our idea for the (r, k)-DOD problem is also useful for the (N, k)-DOD problem and can improve the theoretical and practical computation time and (2) more experimental results than those in our preliminary version.

algorithms [9,32,48] essentially incur $O(n^2)$ time. Empirically, our solution scales almost linearly to n on real datasets.
(2) *Novel metric proximity graph*. To maximize the performance of the above solution, f should be minimized. Let us define neighbors of p as the objects p' such that $dist(p, p') \leq r$. The above solution greedily traverses p's neighbors from p in a given proximity graph. If this proximity graph has paths such that the neighbors of p are reachable from p in a greedy manner, f decreases. (We use *reachability* in this greedy traversal context.)

Motivated by this observation, we devise MRPG (Metric Randomized Proximity Graph), a new proximity graph that is specific to the (r, k)-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, which increases f. The novelty of MRPG is that MRPG improves the reachability to neighbors by making pivot-based monotonic paths between objects with small distances. Furthermore, the space of a MRPG is reasonable to fit into main memory, because it is linear to n.

How to build a MRPG efficiently is not trivial, so we 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 to neighbors while keeping a theoretically comparable efficiency with the state-of-the-art algorithm that builds an approximate K nearest neighbor graph [23], and our algorithm is empirically faster.
(3) *New algorithm for the* (N, k)*-DOD problem*. By utilizing the idea of the first contribution, we propose Prograd, a proximity graph-based algorithm for the (N, k)-DOD problem. Prograd is also easy to parallelize and yields a tight threshold in $\Theta(Nn)$ time. Our filtering technique needs only $O(k)$ time for each object, so it can filter unqualified objects in $O(kn)$ time. Prograd computes (verifies) the exact k-NNs only for not-filtered objects, which needs $O(f'n)$ time, where f' is the number of verified objects. This algorithm hence runs in $O((N + k + f')n)$ time.

Without the sorted arrays described earlier, we need $\Omega(Nn)$ time to solve the (N, k)-DOD problem, because we need to run k-NN search, which needs $\Omega(n)$ time in metric spaces [34], at least N times. Generally, $N > k$, so $O((N + k + f')n) \approx O((N + f')n)$ and Prograd yields a small f' in practice. This suggests that *Prograd could nearly match the lower-bound* in practice.
(4) *Extensive experiments*. We conduct experiments using various real datasets and distance functions. The results demonstrate that our algorithms significantly outperform the state-of-the-art ones. The codes of our algorithms are available in the GitHub repositories.[2,3]

Organization The rest of this article is organized as follows. Section 2 formally defines the problems of this article, and Sect. 3 reviews related work. Sections 4 and 5 solve the (r, k)-DOD and (N, k)-DOD problems, respectively. Section 6 reports our experimental results, and Sect. 7 concludes this article.

2 Preliminary

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., non-negativity, identity of indiscernible, symmetry, and triangle inequality. We next define (r, k)-distance-based outlier and the problem which we solve in Sect. 4.

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

Problem 1 *((r, k)-DOD) Given a distance threshold r, a count threshold k, and a set of objects P, the (r, k)-DOD problem finds all (r, k)-distance-based outliers.*

If p has a small number (k) of neighbors within a sufficiently large distance r from p, it is clearly an outlier. Some density-based clustering algorithms employ this concept to identify noises [6]. The distance threshold r would be specified by domain experts, but, if applications/users do not have sufficient knowledge about P, specifying r may be difficult. Hence, we also consider the (N, k)-DOD problem, which alleviates this issue and can control the number of outliers. This problem is formally defined below.

Definition 3 (k nearest neighbors) The k nearest neighbors of $p \in P$ are k objects in $P \backslash \{p\}$ whose distances to p are the smallest among $P \backslash \{p\}$.

Let $dist_k(p)$ be the distance between p and its k-th nearest neighbor. In Sect. 5, outliers are defined as:

Problem 2 *((N, k)-distance-based outlier) Given P, k, and N, the top-N objects with the largest $dist_k(\cdot)$ among P are (N, k)-distance-based outliers (ties are broken arbitrarily).*

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

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

Then, our problem in Sect. 5 is:

Problem 3 *((N,k)- DOD) Given P, k, and N, this problem finds (N,k)-distance-based outliers.*

An (r,k)- or (N,k)-distance-based outlier is hereinafter called an outlier when the context is clear. We use inliers to denote objects that are not outliers. As in recent works [24,38,44,56], we assume that P is static and resides in the main memory of a single machine. The objective of this article is to develop fast and exact algorithms for Problems 1 and 2.

3 Related work

(r,k)-DOD problem in metric spaces A nested-loop algorithm [32] is a straightforward solution to this problem. This algorithm counts the number of neighbors of a given $p \in P$ by a linear scan of P and terminates it when the count reaches k. This algorithm incurs $O(n^2)$ time, so does not scale to large datasets.

Given r, SNIF [48] forms clusters with radius $r/2$. (Cluster centers are randomly chosen.) If the distance between 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. 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, because p can avoid accessing p' such that $dist(p,p') > r$ by using clusters. However, this approach does not function well on high-dimensional data, due to the curse of dimensionality.

DOLPHIN [9] 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 (r,k)-DOD problem, as can be seen from Definition 1. As a baseline, we employ VP-tree [52], because [20] 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 maximum 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 Sect. 4.2.1.

(N,k)-DOD problem in metric spaces ORCA [15] is the first algorithm that tries to improve practical computation time and is robust to dimensionality. This algorithm is based on nested-loop with early termination (see Lemma 7). ORCA does not consider obtaining a tight threshold quickly, so it cannot exploit the early termination and lacks scalability.

RBRP [25,26] is an improved version of ORCA. This algorithm first partitions P into disjoint subsets, so that each subset contains objects being similar to each other. When computing the exact k-NNs of an object p, RBRP first scans the subset to which p belongs, in order to quickly obtain a tight upper-bound of $dist_k(p)$. However, RBRP has the same drawback as ORCA.

DIODE [42] also employs a nested-loop approach and is the first work that tries to obtain a tight threshold and a tight upper-bound of $dist_k(p)$ quickly. In its pre-processing, DIODE partitions P into disjoint subsets. For each subset, DIODE computes its density. In addition, for each object $p \in P$, DIODE computes the distance to each subset and obtains the access order of subsets for p. Given N and k, DIODE accesses the subsets in ascending order of their densities, based on the assumption that objects p in a subset with a low density have large $dist_k(p)$, to quickly tighten the threshold. DIODE computes the k-NNs of each object in a similar way to RBRP. Because its density estimation may not be accurate, it does not guarantee to tighten the threshold efficiently. Hence, its performance may be worse than those of ORCA and RBRP.

Since the state-of-the-art algorithms rely on a nested-loop approach, they are easy to parallelize. However, the improvement is limited, because they suffer from $O(n^2)$ time.

Other outlier detection works There are other approaches for outlier detection: density-based (e.g., KDE [31] and LOF [18]), angle-based [35], and model-based ones [39]. Density- and angle-based approaches have a similar parameter setting to DOD. On the other hand, a state-of-the-art model-based one (iForest) [39] does not use distance and parameters to evaluate outliers, thus is efficient[4] and may be useful if specifying parameters is difficult.

The above approaches have different applicability, for example, iForest and angle-based approach do not fully cover metric spaces, because they deal with only d-dimensional vectors (metric spaces include other data types, such as

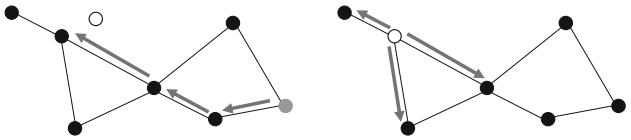
[4] We empirically investigated that our DOD algorithms are still faster than iForest [1]. For example, our algorithms took less than 11, 39, and 13 s to evaluate outliers on Glove, HEPMASS, and PAMAP2 datasets, respectively (see Sect. 6). On the other hand, iForest needed approximately 112, 936, and 349 s on the datasets, respectively.

strings). Furthermore, it is important to note that [27] empirically compared DOD with the above approaches on more than 20 datasets w.r.t. effectiveness. Through ROC-AUC and average precision tests, it shows the following results: (i) The accuracy of DOD is competitive with that of iForest (known state-of-the-art) and usually better than those of the others ([19] also reported similar results). (ii) DOD is more robust against the dimensionality than iForest. These facts also motivate us to address the DOD problems.

Proximity graphs have been demonstrated to be the most promising solution to the approximate k-NN search problem *in in-memory setting* [38]. If the distance between an object p and its approximate k-th NN is within r, p is not an outlier. Also, if this distance is less than an intermediate threshold of the (N, k)-DOD problem, we can skip computing $dist_k(p)$. From these observations, we see that proximity graphs have the potential to solve the (r, k)- and (N, k)-DOD problems efficiently. Some proximity graphs [11,24,28,38] are dependent on L_2 space, so we review proximity graphs in metric spaces.

One of the most famous proximity graphs is KGraph. In this graph, each object corresponds to a vertex and has links to its approximate K-NN (AKNN) objects (i.e., K is the degree of the graph). NNDESCENT [23] is a state-of-the-art algorithm to build an AKNN graph. Our proximity graph is also based on an AKNN graph, and we extend NNDESCENT to build an AKNN graph more efficiently in Sect. 4.2.1. 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 [16]. In a graph based on this model, the number of hops between two arbitrary vertices 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 [40]. To accelerate approximate nearest neighbor (ANN) search, its hierarchical version, HNSW, was proposed in [41]. The upper layers of HNSW are built by sampling objects in their lower layers. This structure aims at skipping redundant vertices to quickly reach vertices with small distances to a query. When we evaluate the number of neighbors of $p \in P$ or an upper-bound of $dist_k(p)$, p is considered as a query object. Figure 2a depicts the search process in the ANN problem: it starts from a random (or fixed) vertex (the grey one) and traverses the proximity graph so that the next vertex is closer to the query object (the 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 Fig. 2b. Therefore, we do not need the skipping structure of HNSW. (If we use the approach in Fig. 2b, HNSW is reduced to NSW, thus we do not consider HNSW as a baseline.)



(a) ANN problem **(b)** Our problem

Fig. 2 Graph traversal difference between our and ANN problems (represented by arrows). Grey and white vertices respectively represent the starting object and a query object

Although these proximity graphs can be employed in our solution to the (r, k)-DOD problem, they cannot optimize the performance of our solution. This is because they do not consider reachability to neighbors. Moreover, if our solution to the (N, k)-DOD problem simply employs them, it cannot deal with any k. We therefore (i) propose a new proximity graph for the (r, k)-DOD problem that takes reachability into account in Sect. 4.2 and (ii) extend AKNN graph so that it can be generally and efficiently utilized for the (N, k)-DOD problem in Sect. 5.

4 Our solution to the (r, k)-DOD problem

4.1 New framework for the (r, k)-DOD problem

Let t be the number of outliers in P. A range search in metric spaces of arbitrary dimension needs $O(n)$ time. Therefore, when we do not have an index with $O(n^2)$ space introduced in Sect. 1, Problem 1 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 (1) the time complexity of a solution nearly matches this lower-bound and (2) the space complexity is linear to n (so as to easily fit into the main-memory). Designing such a solution is however not straightforward. Our new technique for the (r, k)-DOD problem overcomes this non-trivial challenge. Table 1 summarizes the symbols frequently used in Sect. 4.

Main idea Given P, the ratio of outliers in P is small (usually less than 1%) [53]. That is, most objects in P are inliers, so we should identify them as inliers quickly, to reduce the computation time. The evaluation of whether p is an inlier can be converted to answering the problem of range counting with query object p and radius r. To filter inliers quickly, therefore, we need an efficient solution for the problem of range counting, with early termination when the count reaches k. Recently, proximity graphs have shown high potential for solving the approximate nearest neighbor search problem in in-memory setting [38], thanks to the connections between similar objects. Proximity graphs are also promising for solving the range counting problem. Because each object p has links to its similar objects in a proximity

Table 1 Overview of symbols in Sect. 4

Symbol	Description
P	A set of objects
n	Cardinality of P, i.e., $\|P\|$
p	An object
r	A threshold for distance
k	A threshold for the number of neighbors
G	A proximity graph
v	A vertex (i.e., object) in the context of G
K	An initial degree of G
f	The number of false positives
t	The number of outliers
Q	A queue for graph traversal

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 2b depict its intuition, and Example 1 gives an overview of our filtering. Moreover, such a proximity graph does not incur a high space cost, because its space cost is reasonable and usually $O(nK)$, where K is an application-specified degree. Last, to return the exact result set, we just have to verify only not-filtered objects by using an exact range search technique (e.g., linear scan).

To implement the above idea, we propose a proximity graph-based solution, a novel approach to 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).

4.1.1 Filtering phase

In this phase, we filter inliers by exploiting a proximity graph G, which is built in a one-time pre-processing phase. Specifically, we propose GREEDY- COUNTING (Algorithm 2) to count the number of neighbors of an object $p \in P$. Consider that a vertex v in G corresponds to $p \in P$ (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 $dist(p, p') \leq r$, 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 $dist(p, p') \leq r$. (One exception appears in line 13, and this is necessary for MRPG, see Sect. 4.2.4). We next pop the front of Q, say v', check $v'.E$, and do the same as for v. GREEDY- COUNTING is terminated when the count reaches k or Q becomes empty.

Example 2 We use Fig. 1 to describe an example of the filtering phase. Assume $k = 3$. For p in Fig. 1, Algorithm 1 traverses its neighbors from p through the links. GREEDY-COUNTING first accesses the three objects p' that have links from p (by using Q), and we see that $dist(p, p') \leq r$. Therefore, p is an inlier, and Algorithm 1 filters p. Algorithm 1 runs this operation for the other objects iteratively.

Algorithm 1: Proximity Graph-based (r, k)-DOD

```
   Input: P, r, k, and a proximity graph G
 1 /* Filtering phase */
 2 P' ← ∅
 3 for each p ∈ P do
 4     if GREEDY- COUNTING(p, r, k, G) < k then
 5         P' ← P' ∪ {p}
 6 /* Verification phase */
 7 P_out ← ∅
 8 for each p ∈ P' do
 9     if EXACT- COUNTING(p, r, k) < k then
10         P_out ← P_out ∪ {p}
11 return P_out
```

Algorithm 2: GREEDY- COUNTING

```
   Input: p_i, r, k, and a proximity graph G
 1 count ← 0, Q ← {v_i}, check v_i as visited
 2 while Q ≠ ∅ do
 3     v ← the front of Q
 4     Q ← Q\{v}
 5     for each v' ∈ v.E s.t. v' has not been visited do
 6         Check v' as visited
 7         if dist(p, p') ≤ r then
 8             count ← count +1
 9             if count = k then
10                 break
11             Q ← Q ∪ {v'}
12         else
13             if p' is a pivot then
14                 Q ← Q ∪ {v'}
15     if count = k then
16         break
17 return count
```

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

Proof We omit all proofs in Sect. 4 due to space limitation. They appear in [8]. □

4.1.2 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 may do some false positives (i.e., inliers that are not filtered). EXACT- COUNTING in Algorithm 1 verifies whether a given object $p \in P'$ is really an outlier or not in the following way:

– For datasets with low (intrinsic) dimensionality, we conduct a range counting on a VP-tree [52].
– For the others, we use a linear scan, because this is more efficient than any indexing methods for high-dimensional data [21,47].

We terminate verifying 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*.

4.1.3 Analysis

Hereinafter, we assume that the dimensionality is fixed.

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

Remark 1 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 $t \ll n$ [33,53], and (2) t is usually dependent not on n but on the 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 (e.g., see Fig. 9h).

Multi − threading When we say that an algorithm is *parallel-friendly*, we mean that the algorithm can be parallelized with no algorithmic modifications and (almost) no synchronization. Then, the filtering and verification phases of Algorithm 1 are parallel-friendly, because they evaluate each object *independently*. In terms of implementation, assuming the usage of OpenMP, we just need to add "`#pragma omp parallel for`" before the for-loops at lines 3 and 8 of Algorithm 1. This also clarifies that no algorithmic modifications and synchronization are required.

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. We therefore consider a random partition for assigning objects into each thread, in the expectation that each thread has an almost equal load. To (approximately) achieve this without shuffling the dataset (i.e., without doing an additional operation), we use "`#pragma omp parallel for schedule (dynamic)`". This approach practically functions well (e.g., see Fig. 12).

4.2 MRPG

Although our (r, k)-DOD algorithm is orthogonal to any proximity graphs, its performance (i.e., f) depends on a given proximity graph. To maximize the performance, we should minimize f in the filtering phase while keeping a small space cost. Therefore, the main challenge of this section is to reduce f with a proximity graph whose space is linear to n.

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 4 (Monotonic path) Consider two objects p_i and p_{i+x} in P. Let $v_i, v_{i+1}, \ldots, v_{i+x}$ be a path from p_i to p_{i+x} in a proximity graph. 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) [22]. 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.

> 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 many 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 Sect. 4.2.1.) Last, the third property is simple yet important. If objects 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 MRPG building algorithm, which satisfies the above properties through the following steps:

1. NNDescent+: this builds an AKNN graph. We extend NNDescent, a state-of-the-art AKNN graph building algorithm, to quickly build it.
2. Connect- SubGraphs: because an AKNN graph may have disjoint sub-graphs, this step connects such sub-graphs to guarantee that MRPG is strongly connected.
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 properties 1 and 3. We obtain property 2 in the third step. Section 4.2.5 shows 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 (Theorem 4). Moreover, steps 1–3 are carefully designed so that, for a fixed K, each step adds at most $O(n)$ links, thus MRPG is space-efficient (Theorem 5).

4.2.1 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 [23] 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 is based on the idea that, given an object p and its similar object p', objects similar to p' would be similar to p. That is, AKNNs of p can be obtained by accessing its similar objects and their similar ones iteratively. Given K and P, the specific procedures of NNDescent[5] 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 does the above procedure until no updates occur (or α times, where $\alpha = O(1)$).

Because [23] did not analyse the time complexity of NNDescent, we here introduce it.

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

Drawbacks of NNDescent NNDescent provides an AKNN graph with empirically high accuracy, but it has the following drawbacks: (1) The initial random links incur many AKNN updates in the second procedure. Due to this initialization, each object cannot have links to its similar objects in an early stage, incurring unnecessary distance computations. (2) The similar object list of p' is redundantly accessed even when the list has no updates from the previous iteration.

NNDescent+ overcomes the above drawbacks (i) by utilizing data partitioning that clusters similar objects and (ii) by maintaining the update status of similar object lists. In addition, (iii) NNDescent+ obtains the exact K'-NNs for objects with large distances to their K-NNs.

(i) *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. Consider that a node of the VP-tree contains P. If a node contains more objects than the capacity c, this node (or P) generates (or is partitioned into) its two child 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 $p \in P'$ tend to be similar to each other, since each $p \in P'$ has $dist(p, p') \leq \mu$, where p' is the parent of this node. Therefore, for each $p \in P'$, we set its K-NNs in P' as its initial AKNNs. We note that 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

[5] We consider the basic version of NNDescent in [23].

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'
```

left child is a leaf node, are set as *pivots*, which are utilized in future steps. The VP-tree-based partitioning makes pivots being distributed in each subspace of the given data space. This is also the reason why we use this partitioning. Algorithm 3 summarizes our initialization approach, and NNDESCENT+ replaces the first procedure of NNDESCENT with Algorithm 3.

(ii) *Skipping similar object lists with no updates.* When obtaining the similar object list of an object p, NNDESCENT+ adds objects 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.

(iii) *Exact K'-NN Retrieval.* The above initialization and skipping approaches respectively reduce the number of iterations and unnecessary distance computations. However, for an object p such that its K-NNs are relatively far from p, the initialization may provide inaccurate results. The initialization approach clusters objects with small distances. However, given an object p with large distances to its K-NNs, it is difficult for this approach to make p and its K-NNs belong to the same cluster (i.e., node). They may belong to (totally) different clusters, and this may make accurate K-NNs of p not reachable from p in the second procedure. 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 (or it tends to be an outlier). 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 Sect. 4.2.5. Note that m is a constant and $m \ll n$. Therefore, this approach incurs $O(n(K + \log n))$ time. We then have:

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

NNDESCENT+ and NNDESCENT require the same theoretical time, but NNDESCENT+ is empirically faster in most cases. In addition, the procedure of NNDESCENT+ (except for obtaining reverse AKNNs) can exploit multi-threading (via parallel for/sort).

4.2.2 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.[6] Algorithm 4 details our approach, which consists of two phases.

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

BFS with ANN phase (lines 4–24). In the second phase, we propose a randomized approach that exploits breadth-first search (BFS) and ANN search on a proximity graph. We confirm the connection between any two objects through BFS (from a random object). If this BFS has not traversed 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 the 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 retrieve an ANN object for v'_{piv} among $P \backslash P'$ and create links between v'_{piv} and its ANN (lines 18–24). 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 [40]. The inputs of this algorithm are a query object

[6] A similar idea was proposed in [24], but how to add links to make a proximity graph strongly connected is different from our approach. In addition, [24] does not have a theoretical time bound to achieve it.

Algorithm 4: CONNECT- SUBGRAPHS

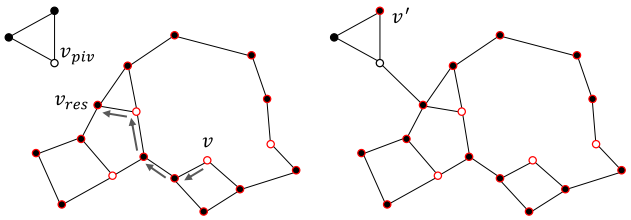
```
Input: G
for each p ∈ P do
    for each (v, v') ∈ v.E such that v' ∉ K'-NN do
        v'.E ← v'.E ∪ {v}
P' ← P
while P' ≠ ∅ do
    Q ← a random vertex (object) v (p) in P'
    P' ← P'\{p}
    while Q ≠ ∅ do
        v ← the front of Q
        Q ← Q\{v}
        for each v' ∈ v.E do
            if p' ∈ P' then
                P' ← P'\{p'}, Q ← Q ∪ {v'}
    if P' ≠ ∅ then
        v'_piv ← a random pivot in P'
        V_piv ← a set of random pivots in P\P'
        dist_min ← ∞, v_res ← v'_piv
        for each v ∈ V_piv do
            v_ann ← ANN- SEARCH(v, v'_piv, G)
            if dist(v_ann, v'_piv) < dist_min then
                dist_min ← dist(v_ann, v'_piv)
                v_res ← v_ann
        v'_piv.E ← v'_piv.E ∪ {v_res}
        v_res.E ← v_res.E ∪ {v'_piv}
```

(v'_{piv}), a starting object ($v \in V_{piv}$), and a proximity graph. Given v, this algorithm traverses objects in $v.E$, identifies the object v' with the minimum distance to v'_{piv}, goes to v', and repeats this until we cannot get closer to v'_{piv}. 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 restart BFS from a random object in P'. (Already traversed objects are skipped.) The above operations are repeated until BFS traverses all objects.

Example 3 Figure 3 illustrates an example of CONNECT-SUBGRAPHS. Figure 3a 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 Fig. 3b. After that, we restart BFS from a random vertex, e.g., v', in Fig. 3b, that has not been traversed yet. Then, we see that this graph is strongly connected.

We set the maximum hop count for the ANN search. This yields that the time complexity of this algorithm is $O(K)$ (since $|V_{piv}| = O(1)$). Then we have:

Lemma 4 *Algorithm 4 requires $O(nK)$ time.*



(a) ANN search **(b)** Restarting BFS

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

4.2.3 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 To make a MSG, we propose GET- NON-MONOTONIC(). We then theoretically show that building a MSG needs $\Omega(n^2)$ time.

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 from p_1 to p_3. We maintain, in an array A_1, pairs $\langle p_j, dist(p_1, p_j)\rangle$, where p_j is an object such that BFS could not confirm a monotonic path from p_1 to p_j. After all objects are traversed, we sort A_1 in ascending order of distance. Note that $A_1[0] = \langle p_1, 0\rangle$.

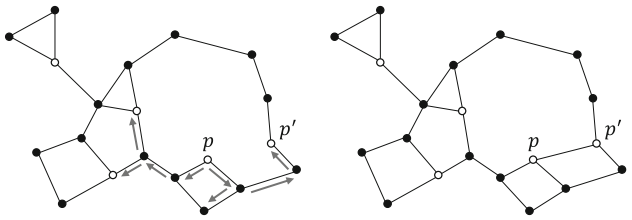
We invoke this function for each object. Now we have an array A_i for each object p_i. We add a link between the two objects appearing in $A_i[j]$ and $A_i[j+1]$ for each $j \in [0, s-1]$, where s is the size of A_i. This approach guarantees that a given proximity graph becomes a MSG. However, a huge cost is incurred.

Theorem 3 *We need $O(n^2(K + \log n))$ time to build a MSG.*

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 not so large [33,49,53]. 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 the property that similar objects of an object p tend to exist within a small hop count from p, and

Algorithm 5: REMOVE- DETOURS

```
   Input: G
1  P' ← a set of randomly chosen objects
2  P_non ← ∅
3  for each p ∈ P' do
4  |   P_non ← P_non ∪ GET- NON- MONOTONIC(p, p, 3, G)
5  |   P_piv ← a set of randomly chosen pivots
6  |   for each p' ∈ P_piv do
7  |   |   P_non ← P_non ∪ GET- NON- MONOTONIC(p, p', 2, G)
8  for each ⟨p, p'⟩ ∈ P_non do
9  |   Create links between p and p'
```



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

Fig. 4 Example of REMOVE- DETOURS

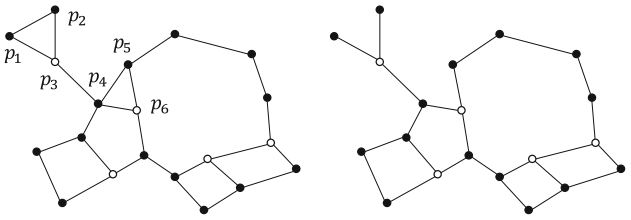
(ii) given p and its similar object p', objects similar to p' tend to be similar to p (i.e., the idea of NNDESCENT). That is, our heuristic is based on the idea: we can create necessary links for p by traversing such objects appearing in observations (i) and (ii).

Algorithm 5 describes our heuristic. Line 1 samples $|P'|$ objects as targets to make monotonic paths. (We do not choose objects that have links to their 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 reaches three from p), to obtain objects with no monotonic path from p (line 4). This is a hop count constrained version of GET- NON- MONOTONIC(), 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 to obtain objects with no monotonic path from p (lines 5–7).

After that, we create necessary links, similar to MSG building (lines 8–9). (See [8] for the setting of 3-hop and 2-hop.)

Example 4 Figure 4a, which shows the proximity graph obtained in Example 3, depicts 3-hop BFS. 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 Fig. 4b.



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

Fig. 5 Example of REMOVE- LINKS

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 $|A| = O(K^2)$ by maintaining only objects with the smallest distances to p. Then, we have:

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

4.2.4 Removing links

Notice that 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 twice. 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 5 Figure 5 depicts an example of REMOVE- LINKS that uses the graph obtained in Example 4. Two non-pivot objects p_1 and p_2 in Fig. 5a have a link to a common pivot p_3. Objects p_4, p_5, and p_6 have the same case. Links (p_1, p_2) and (p_4, p_5) are hence removed, and we have a MRPG shown in Fig. 5b.

By using hash-based link management, we have:

Lemma 6 REMOVE- LINKS *incurs $O(nK)$ time.*

4.2.5 Discussion

From Lemmas 3–6, we see that:

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

In addition, a MRPG is easy to fit into modern main-memory because its space cost is reasonable:

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 larger distances to their AKNNs than the other objects in P. It can be intuitively seen that these objects tend to be outliers. Assume that p has links to its exact K'-NNs. If $K' \geq k$, we can evaluate whether p is an 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, we replace lines 4–5 of Algorithm 1 with the above operation. We set a sufficiently large integer as K' to usually have $K' \geq k$, and, in this case, 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 Sect. 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.

5 Our solution to the (N, k)-DOD problem

Given a number of outliers N, when we do not have the sorted distance arrays with $O(n^2)$ space (described in Sect. 1), Problem 3 requires $\Omega(Nn)$ time. This is because we need to run k-NN search, which incurs $\Omega(n)$ time in metric spaces [34], for at least N objects. Recall that existing algorithms are based on nested-loop, so they incur $O(n^2)$ time. To reduce the *practical* time, they early stop computing the k-NNs by using the distance to an approximate k-th nearest neighbor.

Consider a subset S of P, and assume that $p' \in S$ is the k-th nearest neighbor of an object p among S. Then, we have $dist(p, p') \geq dist_k(p)$, where $dist_k(p)$ is the distance between p and its k-th nearest neighbor among P. That is, $dist(p, p')$ is an upper-bound of $dist_k(p)$. Let $udist_k(p)$ be an upper-bound of $dist_k(p)$. This is utilized as follows [15]:

Lemma 7 *Let τ^* be the N-th largest $dist_k(\cdot)$ among P. Furthermore, let τ be $dist_k(p)$ of an intermediate top N-th object p, i.e., $\tau \leq \tau^*$. Assume that, for $p_i \in P$, an algorithm has computed $dist(p_i, p_j)$ for each $p_j \in S$, where $p_i \neq p_j$ and $|S| \geq k$. Then p_i has $udist_k(p)$. If $udist_k(p_i) \leq \tau$, p_i is not an outlier.*

From this lemma, the computational efficiency can be improved, but the existing algorithms need $O(n)$ time to obtain a tight $udist_k(p_i)$, limiting the scalability. A non-trivial research question in this section is: can we solve Problem 3 while avoiding $O(n^2)$ time and keeping $O(n)$ space? We answer this question by proposing Prograd (*Pro*ximity *gr*aph-based *a*lgorithm for k*n* *d*istance-based DOD).

Prograd is surprisingly simple, but it is time- and space-efficient and parallel-friendly. Its main properties are as follows: (1) Prograd exploits Lemma 7 more efficiently than the state-of-the-art algorithms: different from them, Prograd obtains a tight $udist_k(p_i)$ in $O(k)$ time for each object $p_i \in P$. (2) Prograd obtains a tight τ in $\Theta(Nn)$ time, not $O(n^2)$ time. (3) Prograd runs in $O((N + k + f')n)$ time with $O(n)$ space, and f', the number of not-filtered objects informally, is much less than n, while the state-of-the-art algorithms have $f' = O(n)$. Table 2 summarize the symbols frequently used in this section.

Table 2 Overview of symbols in Sect. 5

Symbol	Description
P	A set of objects
n	Cardinality of P, i.e., $\|P\|$
p	An object
N	The number of outliers
$\mathcal{R}$	The (intermediate) result set
k	The number of nearest neighbors
$dist_k(\cdot)$	The distance to the k-th NN object
τ	The (intermediate) top-N largest $dist_k(\cdot)$.
$udist_k(\cdot)$	An upper-bound of $dist_k(\cdot)$
P_{cand}	A set of candidate objects for outliers
Q	A queue for graph traversal

Main idea We show that the idea in Sect. 4.1 is also applicable to the problem in this section. That is, we again exploit a proximity graph to obtain both a tight τ and a tight $udist_k(p_i)$.

Consider a proximity graph of P, for example an AKNN graph. It is clear that $p \in P$ can obtain a tight $udist_k(p)$ by traversing the proximity graph from p. Besides, we can compute $udist_K(p)$ offline, since K is independent of k. It can be seen that p would exist in a sparse (or low density) space if $udist_K(p)$ is large. Outliers exist in a sparse space and are not sensitive to k [19], and real datasets tend to have this observation, see Sect. 5.3. Now we observe that a tight τ would be obtained by accessing *only* N objects with the largest $udist_K(\cdot)$. This cost is much cheaper than those of the existing algorithms, but this threshold can support more powerful filtering than them.

Overview As with Algorithm 1, Prograd employs a filter-and-verification framework. However, different from Algorithm 1, Prograd requires a strongly connected proximity graph to obtain $udist_k(\cdot)$ for an arbitrary k. From the problem definition and the above idea, the proximity graph should be modeled by AKNN graph. We therefore employ a strongly connected AKNN graph as the index of Prograd. This index is obtained in $O(nK^2 \log K)$ time by using NNDescent+ and Algorithm 4. During building this, each object p has an upper-bound of $dist_K(p)$, $udist_K(p)$. We sort P in descend-

Algorithm 6: Prograd (Filtering Phase)

Input: P, N, k, and τ
Output: P_{cand} (a set of objects that are not filtered) and $\mathcal{R}$

```
1  R ← ∅
2  /* (1) Computing τ */
3  for each i ∈ [1, N] do
4      R ← R ∪ {p_i}
5  τ ← dist_k(R[N])
6  /* (2) Filtering */
7  for each i ∈ [N + 1, n] do
8      P_visit ← {p_i}, Q ← ∅
9      for each (p_i, p_j) ∈ p_i.E do
10         Q ← Q ∪ {p_j}
11     while Q ≠ ∅ do
12         p ← Q[0], Q ← Q\{p}
13         if p ∉ P_visit then
14             P_visit ← P_visit ∪ {p}, dist ← dist(p_i, p)
15             Update k-NNs of p
16             if udist_k(p_i) ≤ τ then
17                 Mark p_i as filtered
18                 break
19             if p_i accessed m' distinct objects then
20                 break
21             for each (p, p') ∈ p.E do
22                 Q ← Q ∪ {p'}
23 P_cand ← a set of not-filtered objects ∉ R
24 Sort P_cand in descending order of udist_k(·)
```

ing order of the upper-bound at the end of pre-processing. Obviously, this index needs only $O(n)$ space (for a fixed K).

Prograd runs Algorithm 6 and then runs Algorithm 7. Simply put, Prograd first obtains a threshold τ, and then filters inliers based on Lemma 7. After this filtering, Prograd verifies not-filtered objects to investigate the correct answer. By exploiting $udist_k(\cdot)$ and optimizing the accessing order, Prograd further skips unnecessary k-NN computations in this verification phase, different from the nested-loop approach.

5.1 Filtering phase

Algorithm 6 details the filtering phase. Prograd maintains the (intermediate) result with $\mathcal{R}$, an ordered set of N objects, which is sorted in descending order of $dist_k(\cdot)$.

(1) *Computing* τ. Recall that P is sorted in descending order of $udist_K(\cdot)$. Without loss of generality, we assume $P = \{p_1, p_2, \ldots, p_N, \ldots\}$. Prograd computes the exact k-NNs of the first N objects in P and adds them into $\mathcal{R}$. After that, Prograd sets $\tau = dist_k(\mathcal{R}[N])$.

As noted earlier, this heuristic is based on the idea that objects with large $udist_K(\cdot)$ would exist in a sparse space so are probable to be outliers. In practice, this heuristic provides a tight τ and yields effective filtering. Recall that, in NNDescent+, some objects have the exact distances to their K'-NNs, where $K' > K$. We discuss how to exploit this property to optimize Prograd in Sect. 5.3.

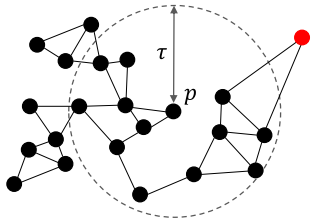


Fig. 6 Example of filtering in Prograd

(2) *Proximity graph-based filtering*. Given an object $p_i \notin \mathcal{R}$, Prograd tries to filter p_i based on τ. Our filtering is based on BFS, because each object p can access its similar objects first by starting BFS from p on the strongly connected AKNN graph.

For each $p_i \in P \backslash \mathcal{R}$, when Prograd visits p for the first time during the BFS from p_i, Prograd computes $dist(p_i, p)$ and updates the intermediate k-NNs of p_i. If $udist_k(p_i) \leq \tau$, Prograd filters p_i from Lemma 7, then terminates the BFS of p_i. Otherwise, Prograd continues the BFS until it accesses m' objects.

After that, Prograd obtains a set P_{cand} of objects $p \notin \mathcal{R}$ such that $udist_k(p) > \tau$. At the end of this phase, Prograd sorts P_{cand} in descending order of $udist_k(\cdot)$. This is necessary to further prune the exact k-NN computation in the next phase.

Example 6 We use Fig. 6 to describe an example of filtering in Prograd. For simplicity, assume $N = 1$ and $k = 2$. We assume that the red object in Fig. 6 has the largest $udist_K(\cdot)$. Then, τ is the distance to its 2nd nearest neighbor. Now focus on the object p, which is at the center of the dotted circle with radius τ. Prograd runs BFS from p to compute $udist_2(p)$. We have $udist_2(p) < \tau$ when Prograd accesses the objects connected with p, so Prograd marks p as filtered. The same operation is conducted for the other objects iteratively.

Correctness and efficiency Trivially, we have $\tau \leq \tau^*$, where τ^* is the exact threshold, thus our filtering guarantees the correctness, i.e., the outliers exist in $\mathcal{R}$ or P_{cand}. Note that, thanks to the strongly connected nature, BFS from an arbitrary object can access at least k other objects. We here clarify the efficiency of the filtering phase. By setting $m' = O(k)$, we have:

Lemma 8 *Algorithm 6 requires* $O((N + k)n)$ *time.*

Proof It is apparent that the first step, computing τ, requires $\Theta(Nn)$ time, since Prograd computes the exact k-NNs of

the first N objects in P. For a given object p, our filtering computes the distances to at most $m' = O(k)$ other objects, thus it requires at most $O(kn)$ time. The last sorting requires $O(|P_{cand}| \log |P_{cand}|)$ time. In the worst case, $|P_{cand}| = O(n)$ but generally $N + k > \log n$. (In practice, we have $|P_{cand}| \ll n$.) Now the correctness of this lemma is clear. □

$Multi-threading$ This phase has three main parts: computing τ, filtering, and sorting. Computing the exact k-NNs of the first N objects is easy to parallelize, as each k-NN computation can be done independently. Similar to the filtering in Algorithm 1, our filtering in this section is parallel-friendly, since computing $udist(\cdot)$ is an independent operation. Section 4.2.1 has already introduced that sorting can be parallelized. Then, we see that Algorithm 6 is parallel-friendly.

Does batch filtering improve efficiency? One may consider filtering multiple objects in a batch. Let $\hat{P}_i^k$ consist of p_i and its approximate k-NNs. If arbitrary two objects in $\hat{P}_i^k$, say p_a and p_b, have $dist(p_a, p_b) \leq \tau$, they are not outliers, as they all have $udist_k(\cdot) \leq \tau$.

Assume that we collect a set of objects p_j such that $dist(p_i, p_j) \leq \frac{\tau}{2}$ during we compute $udist_k(p_i)$ in Algorithm 6. Then, if the set size is larger than k, all objects in this set have $udist_k(\cdot) \leq \tau$ from triangle inequality. In this case, we can filter all objects in this set in a batch (i.e., we do not need to compute $udist_k(p_j)$ for each p_j in this set). However, this approach gives little gain. This is because (1) having such sets is generally hard in particular on middle- or high-dimensional data due to the curse of dimensionality, and (2) the main cost in this phase is derived from computing τ, since $\Theta(Nn) > O(kn)$. We thus do not consider this batch filtering to keep the filtering phase concise.

5.2 Verification phase

In this phase, Prograng verifies not-filtered objects and computes the exact outliers. It is important to see that we do not necessarily verify (i.e., compute the exact k-NNs of) all objects in P_{cand}.

Corollary 1 *Assume that $P_{cand} = \{p_i, p_{i+1}, \ldots, p_{i+s}\}$, where $s = |P_{cand}| - 1$. If we have $udist_k(p_{i+j}) \leq \tau$, objects p_{i+l} such that $l \in [j, s]$ cannot be outliers.*

Proof We have $udist_k(p_i) \geq \ldots \geq udist_k(p_{i+s})$, since P_{cand} is sorted by $udist_k(\cdot)$. Hence, if $udist_k(p_{i+j}) \leq \tau$, from Lemma 7, objects p_{i+l}, where $l \in [j, s]$, have $dist_k(p_{i+l}) \leq \tau$. □

From this corollary, when we have $udist_k(p_{i+j}) \leq \tau$, we can safely prune all objects p_{i+l} such that $l \in [j, s]$. This upper-bound-based access order can minimize the cost of computing the exact k-NNs while guaranteeing the correctness. This is also an advantage over existing (N, k)-DOD algorithms, because our *access order*-based termination is not available for their nested-loop approach.

Algorithm 7: Prograng (Verification Phase)

```
Input: P, N, k, τ, and P_cand
Output: R (a set of N outliers)
for each p_i ∈ P_cand do
    if udist_k(p_i) > τ then
        dist_k(p_i) ← distance to the k-th NN of p_i
        if dist_k(p_i) > τ then
            Update R and τ
    else
        break
```

Algorithm 7 elaborates this phase. To exploit the above corollary, Prograng sequentially verifies the objects in P_{cand}. Assume that now Prograng verifies an object $p_i \in P_{cand}$. If $udist_k(p_i) > \tau$, Prograng computes $dist_k(p_i)$ by using a linear scan of P with Lemma 7, and then updates $\mathcal{R}$ and τ. Otherwise, we can guarantee that $\mathcal{R}$ has the exact outliers, so Prograng terminates the verification.

Lemma 9 *Algorithm 7 requires $O(f'n)$ time, where f' is the number of verified objects.*

Proof Prograng computes $dist_k(p_i)$ through a linear scan of P, which immediately derives this lemma. □

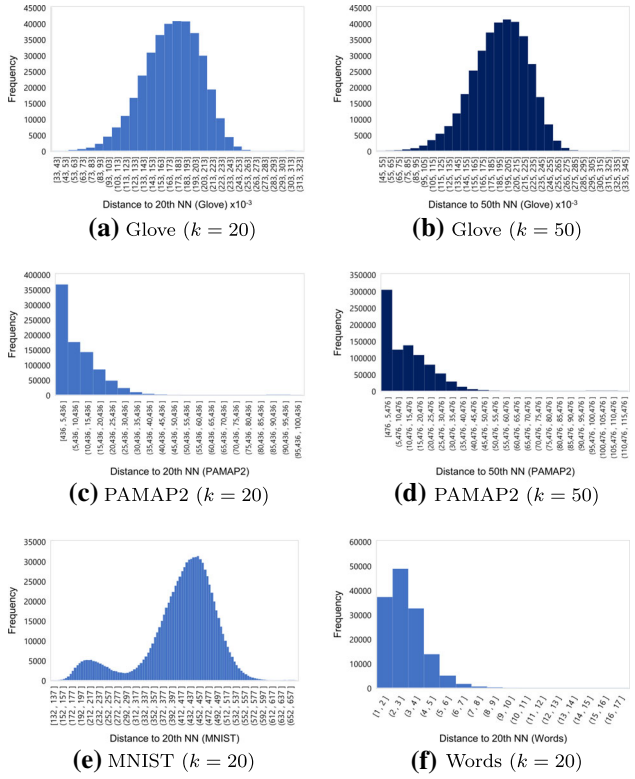
$Multi-threading$ This phase is also easy to parallelize. Since Prograng computes the exact k-NNs of a given $p \in P_{cand}$ through a linear scan of P, we parallelize this scan by partitioning P into equal-sized disjoint subsets. Each thread computes the k-NNs in its assigned subset. Then Prograng merges each local result to obtain $dist_k(p)$. This merge incurs a negligible cost, suggesting that this is also parallel-friendly.

5.3 Analysis

We present the main result of this section. From Lemmas 8 and 9 , we have:

Theorem 6 *The time complexity of Prograng is $O((N + k + f')n)$, where f' is the number of verified objects.*

Remark 2 From the above result, it is easy to see that Prograng outperforms the state-of-the-art algorithms if f' is small, since they theoretically incur $O(n^2)$ time. (For datasets with $f' = o(n)$, Prograng is always better than the state-of-the-art algorithms theoretically.) We generally have $N > k$ as with Theorem 1, then $O((N+k+f')n) = O((N+f')n)$. Practically, Prograng often has $f' < N$, so it nearly matches the $\Omega(Nn)$ time of Problem 3 with $O(n)$ space and is much faster than the state-of-the-art algorithms.



(a) Glove ($k = 20$) **(b)** Glove ($k = 50$)
(c) PAMAP2 ($k = 20$) **(d)** PAMAP2 ($k = 50$)
(e) MNIST ($k = 20$) **(f)** Words ($k = 20$)

Fig. 7 Distribution of distance to the k-th nearest neighbor

Why can f' be small? We analyze Prograd's efficiency that provides a small f', by using the observations from real datasets and AKNN graph. Figure 7 depicts the distributions of distances to the k-th NN in some real datasets used in our experiments. (We used random $\min\{0.3n, 1000000\}$ objects to obtain the distributions.) The distances follow Gaussian (Glove and MNIST) or Zipf-like (PAMAP2 and Words) distributions. That is, only a small number of objects have a (much) longer distance to their k-th NN than those of the others, as can be seen from the right side of each figure. This observation suggests that, as long as N is not too large, if (1) we can obtain a tight threshold τ, and (2) $udist_k(p) \approx dist_k(p)$ for each $p \in P$, then we can filter most objects. In these conditions, it is trivial that f', i.e., the number of objects p having $udist_k(p) > \tau$ is small.

From Fig. 7a–d, for different k, the distributions do not change. (We omit the distributions of distances to the 50th NN of MNIST and Words, because they are similar to the ones to the 20th NN.) This observation suggests that, given k and $k'(\neq k)$, we have similar rank orders of each object in the $dist_k(\cdot)$- and $dist_{k'}(\cdot)$-based rankings defined in Problem 3. Then, if we have $udist_k(p) \approx dist_k(p)$ and $udist_K(p) \approx dist_K(p)$ for an arbitrary p, the first N objects in P tend to place high ranks in Problem 3. — ($\star$)

As stated in Sect. 3, approximate k-NN searches based on proximity graphs provide highly accurate results. This derives $udist_k(p) \approx dist_k(p)$ and condition 2. Actually, the accuracy of NNDESCENT+ is almost perfect in practice (see Sect. 6.1.1), so $\frac{udist_K(p)}{dist_K(p)} \approx 1$. Now we have ($\star$), which satisfies condition 1, i.e., a tight threshold τ. Consequently, we practically have conditions 1 and 2, which renders a small f', regardless of n. (This is also confirmed empirically, see Sect. 6.2.) It can be seen that the above discussion also justifies our approach of utilizing a proximity graph.

Optimization Recall that NNDESCENT+ computes the exact K'-NNs for a constant number of objects. Recall further that Prograd utilizes the first N objects to initialize the threshold τ. If $k \leq K'$, Prograd can utilize the objects, whose exact K'-NNs have been computed, to obtain τ. In this case, Prograd needs only $\Theta(N)$ time to obtain τ. Then, the total cost of Prograd is $O((k + f')n)$. This optimization makes Prograd (much) faster. (As noted above, this optimization is available only when $k \leq K'$, and the setting of K' is an application matter. So, Prograd without this optimization is assumed to be the default setting.)

6 Experiments

Our experiments were conducted on a Ubuntu machine with dual 12-core 3.0GHz Intel Xeon E5-2687w v4 processors 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.

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

Datasets We used the following seven real datasets.

- Deep: a set of image features obtained by a deep neural network.
- Glove: a set of dense word embedding vectors (words are from Twitter).
- HEPMASS: a set of learned feature vectors of exotic particles.
- MNIST: a set of gray-scaled digit images.
- PAMAP2: a set of sensor readings obtained from human activities.
- SIFT: a set of SIFT feature vectors of images.
- Words: a set of English words (i.e, strings).

Table 3 summarizes the statistics of the above datasets and the distance functions we used. Because the original domains of PAMAP2 are different from each other, we normalized PAMAP2 so that the domain of each dimension was $[0, 10^5]$.

Table 3 Statistics of datasets and distance functions used

Dataset	n	Dim.	Distance function
Deep [12]	10,000,000	96	L_2-norm
Glove [43]	1,193,514	25	Angular distance
HEPMASS [13]	7,000,000	27	L_1-norm
MNIST [2]	3,000,000	784	L_4-norm
PAMAP2 [46]	2,844,868	51	L_2-norm
SIFT [3]	1,000,000	128	L_2-norm
Words [4]	466,551	1–45	Edit distance

Table 4 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

6.1 Evaluation of (r, k)-DOD algorithms

Algorithms We evaluated the following algorithms:

- State-of-the-art ones: *Nested-loop* [15], *SNIF* [48], *DOLPHIN* [9], and *VP-tree* [52].
- Proximity graph-based ones: *NSW* [40], *KGraph* [23], *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. 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 algorithms. For KGraph, MRPG-basic, and MRPG on PAMAP2, we set $K = 40$, and we set $K = 25$ on the other datasets. The number of links for each object in NSW is set so that its memory usage is almost the same as that of KGraph. For MRPG, we set $K' = 4 \times K$.

Parameters Table 4 shows the default parameters. They were specified so that the outlier ratio is small [53] or clear outliers are identified (see Sect. 2). 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, due to their large n or dimensionality.

6.1.1 Evaluation of NNDescent+

To confirm that NNDescent+ builds an accurate AKNN graph, we measured recall and distance rate $\hat{dist}_K/dist_K$, where $\hat{dist}_K$ is the distance to an approximate K-th NN obtained by NNDescent+ (or NNDescent). We used Glove, SIFT, and Words to evaluate the accuracy of NNDescent and NNDescent+. (To obtain the exact K-NNs, we need $O(n^2)$ time, so we did not evaluate accuracy on the other datasets.)

Table 5 Accuracy of NNDescent (NNDescent+)

Dataset	Recall	$\hat{dist}_K/dist_K$
Glove	0.94 (0.95)	1.00 (1.00)
SIFT	0.96 (0.96)	1.00 (1.00)
Words	0.99 (0.99)	1.00 (1.01)

Table 5 shows the experimental result. Clearly, the accuracy of NNDescent+ is competitive with that of NNDescent and is almost perfect. This result suggests that each object in P can traverse its neighbors efficiently. In the next experiments, we show that NNDescent+ is faster than NNDescent.

6.1.2 Evaluation of offline processing

We evaluated the pre-processing efficiency of proximity graphs: 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 s for each dataset.

MRPG(-basic) vs. KGraph Table 6 presents the pre-processing time of each proximity graph under 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 may be surprising, because MRPG has additional operations after running NNDescent+ whereas KGraph simply computes an AKNN graph. Actually, this result is derived from the efficiency of NNDescent+. We depict the decomposed times of building a KGraph, MRPG-basic, and MRPG on Glove in Table 7, as an example. This table shows that NNDescent+ is faster than NNDescent, demonstrating the effectiveness of partitioning based on VP-tree and skipping similar object lists. Also, the other functions for building a MRPG do not incur significant costs. These provide high performance for building a MRPG.

Table 6 Pre-processing time (s)

Dataset	NSW	KGraph	MRPG(-basic)
Deep	NA	20,079.80	17,230.30 (13417.40)
Glove	2333.47	923.83	791.53 (755.54)
HEPMASS	NA	7935.25	5221.86 (4345.63)
MNIST	33368.60	2972.96	2281.55 (1566.05)
PAMAP2	4522.14	1325.40	888.61 (729.54)
SIFT	4910.94	929.48	817.33 (723.75)
Words	871.27	455.15	868.62 (707.08)

Table 7 Decomposed time of pre-processing on Glove (s)

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

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 the building of a MRPG. That is, NNDescent+ for MRPG incurs longer time than that for MRPG-basic, as Table 7 presents.

NSW vs. the other proximity graphs Building a NSW consistently needs longer time than building the other proximity graphs, which can be seen from Table 6. Because the NSW building algorithm is based on iterative object insertion, this property lacks scalability to large datasets. Therefore, NSW cannot be built on Deep and HEPMASS within a half day.

Scalability test Figure 8 illustrates the scalability to n for building the proximity graphs. We varied n by random sampling (i.e., we varied the sampling rate). NSW basically needs (much) longer time for building. This algorithm is competitive only in the case of Words, because its cardinality is smaller than those of the other datasets. KGraph, MRPG-basic, and MRPG have linear scalability to n, due to Theorems 2 and 4.

6.1.3 Evaluation of online processing

We evaluated (r, k)-DOD algorithms. Tables 8 and 9 describe the running time and index size of each algorithm, respectively. Algorithms whose index could not be built within the time limit were not tested.

Our approach vs. state − of − the − art Let us compare our approach, proximity graph-based solution (NSW, KGraph, MRPG-basic, and MRPG), with the state-of-the-art (Nested-loop, SNIF, DOLPHIN, and VP-tree). Table 8 shows that our approach is clearly faster than the state-of-the-art, demonstrating its robustness to the distance functions listed in Table 3. This result is derived from the reduction of unnecessary distance computations. Specifically, in our approach (or a proximity graph), each object tends to have links (or paths) to its neighbors. This yields efficient filtering, 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 spending 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 Sect. 4.2.5).

Table 9 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 vs. the other proximity graphs We next focus on the performances of the proximity graphs. Table 8 reports that MRPG is the 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 10 shows that MRPG and MRPG-basic reduce f more compared with KGraph and NSW. This fact demonstrates the effectiveness of monotonic paths, i.e., MRPG and MRPG-basic have better reachability than the others. We notice that the performance difference between MRPG and KGraph is not significant on Deep 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 in the verification 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.[7] The verification cost of outliers and false positives thus remains on them, as with the other proximity graphs.

Recall that each dataset follows a power law distribution w.r.t. the number of neighbors. If a dataset has many objects

[7] Although NSW has more f than 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.

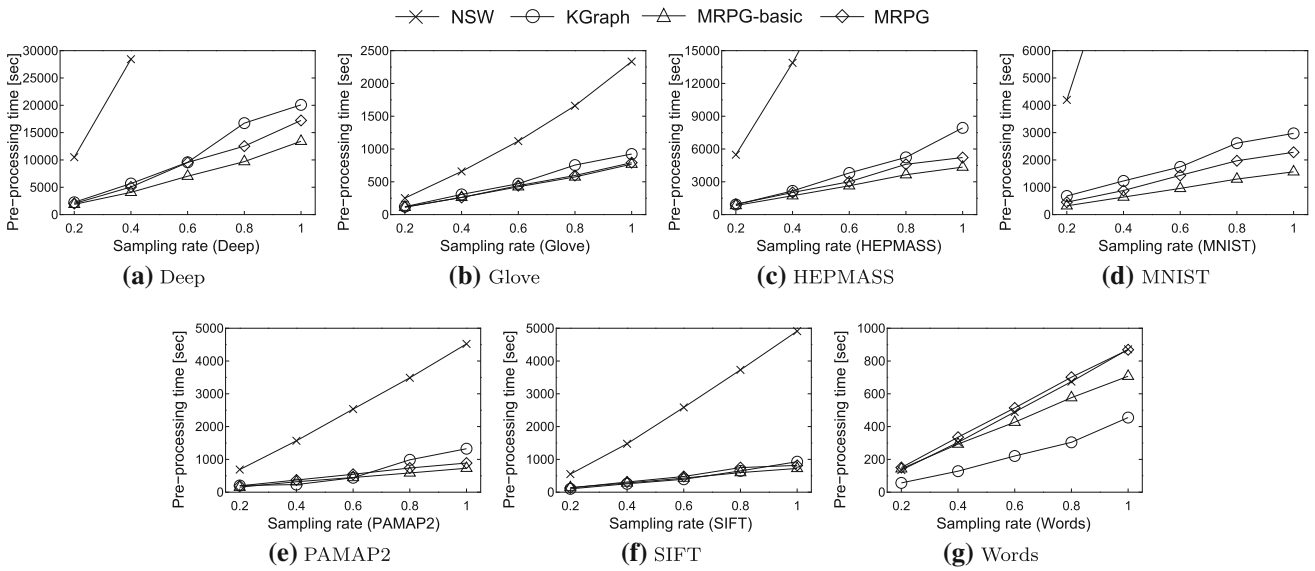


Fig. 8 Impact of n (pre-processing time) on the (r, k)-DOD problem

Table 8 Running time (s)

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	17,295.40	20,360.80	NA	8597.23	NA	780.19	638.83	**38.40**
MNIST	15,494.00	22,579.80	NA	13,836.60	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**

Bold shows the winner

Table 9 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

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 the data distributions, so f between the datasets are different, as shown in Table 10. For example, we observed that Deep is sparser than the other datasets,[8] so its f is large.

Table 11 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.)

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

Table 10 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	2363	1,743
MNIST	7079	4698	2509	438
PAMAP2	18,346	22,543	4290	3986
SIFT	4899	2513	585	51
Words	9569	989	120	4

Table 11 Decomposed time of outlier detection on Glove (s)

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

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 additional 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 Algorithms 4 and 5 . 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 report the numbers of false positives provided by the first, second, and third variants on PAMAP2: they are respectively 11937, 4712, and 9720. They are less than those of NSW and KGraph, see Table 10. From this result, we see that Connect- SubGraphs is useful and Remove- Detours is important to provide fewer false positives.

Varying n Figure 9 studies the scalability of each proximity graph in the same way as in Fig. 8. (Parameters were fixed as the default ones.) Since Table 8 confirms the superiority of our approach over the state-of-the-art, we omit their results.

As n 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 our monotonic path creation provides their scalability. Figure 9h confirms that $f+t$ of MRPG is constant or almost not affected by n. As stated earlier, we found that t is usually not dependent on n, and MRPG yields $f \ll t$ or at least $f < t$. This observation provides the (almost) linear scalability of our (r, k)-DOD algorithm with MRPG.

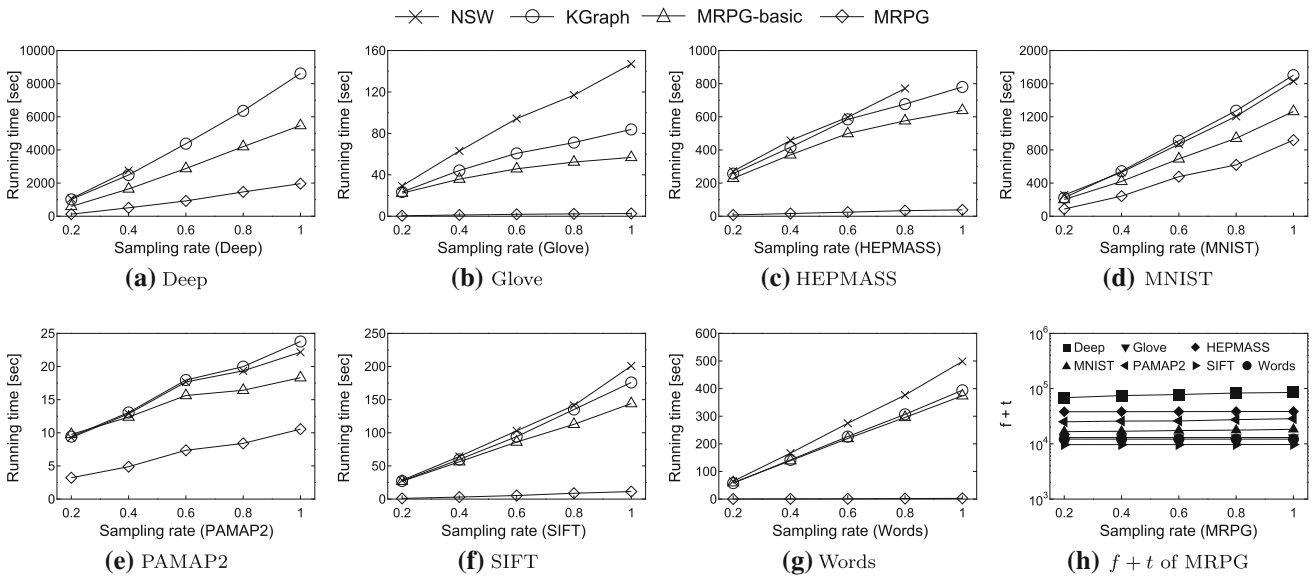
In the Words case, MRPG-basic and KGraph show similar performances. We observed that the outliers in Words have large dimensionality. Because computing edit distance needs a quadratic cost to dimensionality, the verification of outliers incurs a large computational cost. For example, with the default parameters, MRPG-basic (KGraph) took 2.43 (0.73) and 371.65 (393.23) s for filtering and verification, respectively. From the result in Table 10, we see that false positives in Words are verified quickly (by early termination) and the verification of outliers dominates the running time.

Varying k. We investigated the influence of outlier ratio by varying k. Figure 10 presents the results. (The omitted results appear in [8].) As k increases, our approach needs to traverse more objects, rendering a larger filtering cost. In addition, as k increases, the outlier ratio increases. Our approach hence needs a 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 Fig. 11 (k is fixed at the default value). Recall that the time of Algorithm 1 depends on the output size, as its time complexity is $O((f+t)n)$. As r increases, the output size decreases thus its running time also decreases. Similar to the results in Fig. 10, MRPG keeps outperforming KGraph and NSW when the outlier ratio is either high or low.

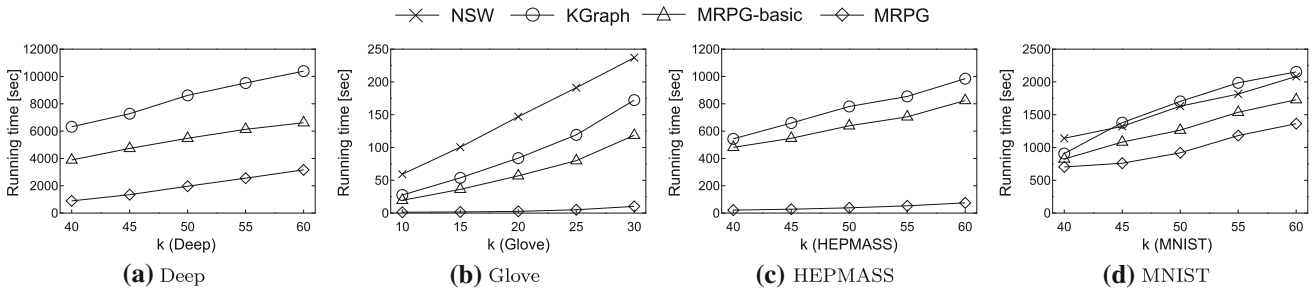
From Fig. 11, we observe that the output size tends to converge by increasing r. The objects returned by this problem with such r are clearly isolated ones. Applications of the (r, k)-DOD problem follow the concept: objects with a small number of neighbors within sufficiently large distances are outliers. Running our algorithm by varying r (from small to large) is a feasible way of finding such objects, i.e., outliers required by the applications. After doing this, the applications may vary k and again run our algorithm while varying r. By iterating this, they can obtain the result sets of multiple pairs of r and k. Although how to get the final result from these sets is not our scope, a possible method is to import



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

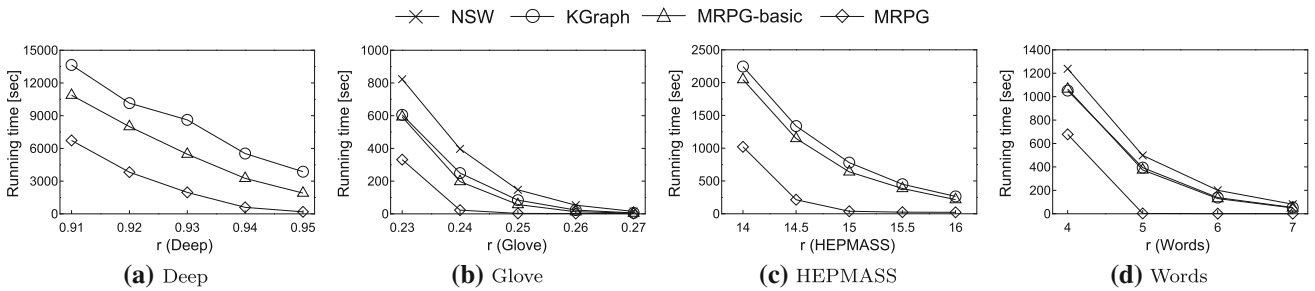
(e) PAMAP2 **(f)** SIFT **(g)** Words **(h)** $f + t$ of MRPG

Fig. 9 Impact of n on the (r, k)-DOD problem



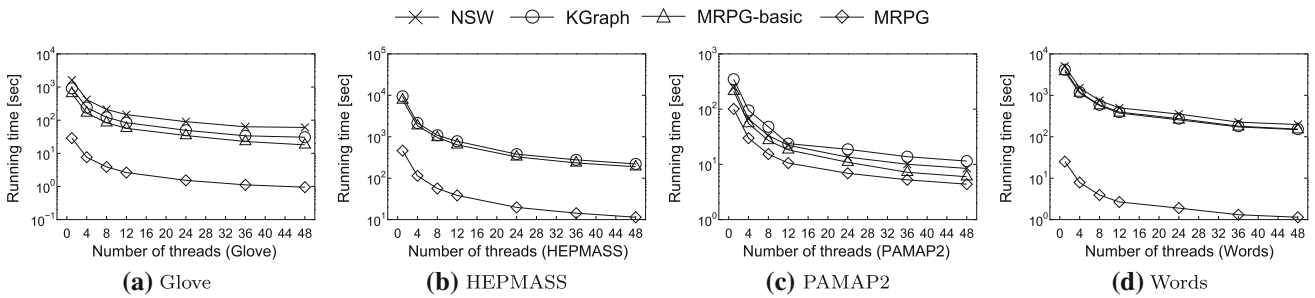
(a) Deep **(b)** Glove **(c)** HEPMASS **(d)** MNIST

Fig. 10 Impact of k on the (r, k)-DOD problem



(a) Deep **(b)** Glove **(c)** HEPMASS **(d)** Words

Fig. 11 Impact of r



(a) Glove **(b)** HEPMASS **(c)** PAMAP2 **(d)** Words

Fig. 12 Impact of the number of threads on the (r, k)-DOD problem

Table 12 Running times (s) of Prograng and Prograng-opt

Algorithm	Prograng	Prograng-opt
Deep	356.44	261.90
Glove	19.49	10.91
HEPMASS	71.09	32.80
MNIST	266.17	56.10
PAMAP2	18.14	12.92
SIFT	17.25	5.54
Words	70.44	8.41

Table 13 Decomposed time (s) of Prograng

	Computing τ	Filtering	Verification
Deep	94.54	54.65	207.208
Glove	8.59	4.63	6.27
HEPMASS	38.29	27.52	5.28
MNIST	202.64	14.26	48.41
PAMAP2	5.22	10.83	2.09
SIFT	11.71	3.91	1.64
Words	62.03	1.90	6.51

only objects that frequently appear in the above result sets. This is because these objects tend to be evaluated as outliers even for comparatively small k and large r among the specified values of r and k. (Recall that objects are easy to be inliers for small k and large r.)

Varying the number of threads We clarify that our approach exploits multi-threading empirically. Figure 12 shows the results on Glove, HEPMASS, PAMAP2, and Words. We see that our solution exploits the available threads and has almost linear scalability to the number of threads. For example, compared with a single thread case, MRPG with 12 threads obtains about 10–12 times speed-up in the datasets. Also, the superiority among the proximity graphs does not change.

6.2 Evaluation of (N, k)-DOD algorithms

We turn our attention to the (N, k)-DOD problem.

Algorithms We evaluated ORCA [15], RBRP [25,26], DIODE [42], Prograng, and Prograng-opt (Prograng with optimization in Sect. 5.3). Note that the three state-of-the-art algorithms were also parallelized.

The inner parameters of RBRP and DIODE were set by following the original papers. The setting of the proximity graph of Prograng(-opt) is the same as that in Sect. 6.1. Section 6.1.2 has already shown that we can build a strongly connected AKNN graph efficiently, so this section focuses on online time.[9]

Parameters We set $N = 1000$ and $k = 20$ by default. The default number of threads is 12.

Effectiveness of our approach We demonstrate the effectiveness of our approach to Problem 3 by showing the number of verified objects (i.e., we show f'). On Deep, Glove, HEPMASS, MNIST, PAMAP2, SIFT, and Words, f' of Prograng is respectively 3448, 920, 130, 196, 308, 146, and 139. We see that $f' \ll n$, and Prograng filters *more than* 99.9% of the objects in P.

Effectiveness of optimization To see how the optimization contributes to efficiency improvement, we compared Prograng-opt with Prograng. Table 12 shows the comparison result. Prograng-opt obtains at least 1.4x speed-up. Table 13 details the decomposed time of Prograng. Recall that Prograng-opt optimizes computing τ, and we observed that Prograng-opt computes τ within 0.001 [sec]. We then see that Prograng-opt functions well, when computing τ is a bottleneck, e.g., HEPMASS, MNIST, SIFT, and Words cases.

Varying n. Next, we investigated the scalability of each algorithm in the same way as Sect. 6.1. Figure 13 depicts the result (plots are log-scale). We have three main observations. First, Prograng(-opt) scales (almost) linearly to n. This is because, as shown in Fig. 13h, f' of Prograng(-opt) is almost constant. (Words does not seem to have this case, but still has $f' \ll n$.) Second, Prograng-opt is the clear winner in all tests and Prograng also outperforms the state-of-the-art with a large margin (with the exception of the Words case). Last, in Words case, RBRP, DIODE, and Prograng have similar performances. We found that RBRP and DIODE also obtain a tight τ quickly on Words. As shown in Table 13, the bottleneck of Prograng on Words is τ computation, thus the last observation is reasonable.

Varying N Figure 14 shows the impact of the number of outliers N. (We do not show the result if a given algorithm could not terminate (N, k)-DOD within the time limit.) As N increases, the running time of each algorithm increases, except for Prograng-opt on Words. This is reasonable, because τ decreases as N increases. Notice from Fig. 14a that Prograng(-opt) is the only algorithm that can deal with large n and N.

The Words case, i.e., the result in Fig. 14g, has a different observation from those with the other datasets. First, DIODE is a little better than Prograng. We found that DIODE obtains a more accurate threshold than Prograng on Words. However, the difference is small, and DIODE is much slower than Prograng on the other datasets. Second, the performance of Prograng-opt is not stable. Recall that the performance of

[9] Actually, we can employ a strongly connected NSW as an index of Prograng, but Sect. 6.1.2 has demonstrated that building an NSW does not scale well. Therefore, we do not consider this index. In addition, Sect. 6.1 demonstrated that triangle inequality-based pruning, i.e., SNIF, does not improve efficiency compared with the simple nested-loop. Prograng with the batch filtering discussed in Sect. 5.1 has the same case, thus we do not test it.

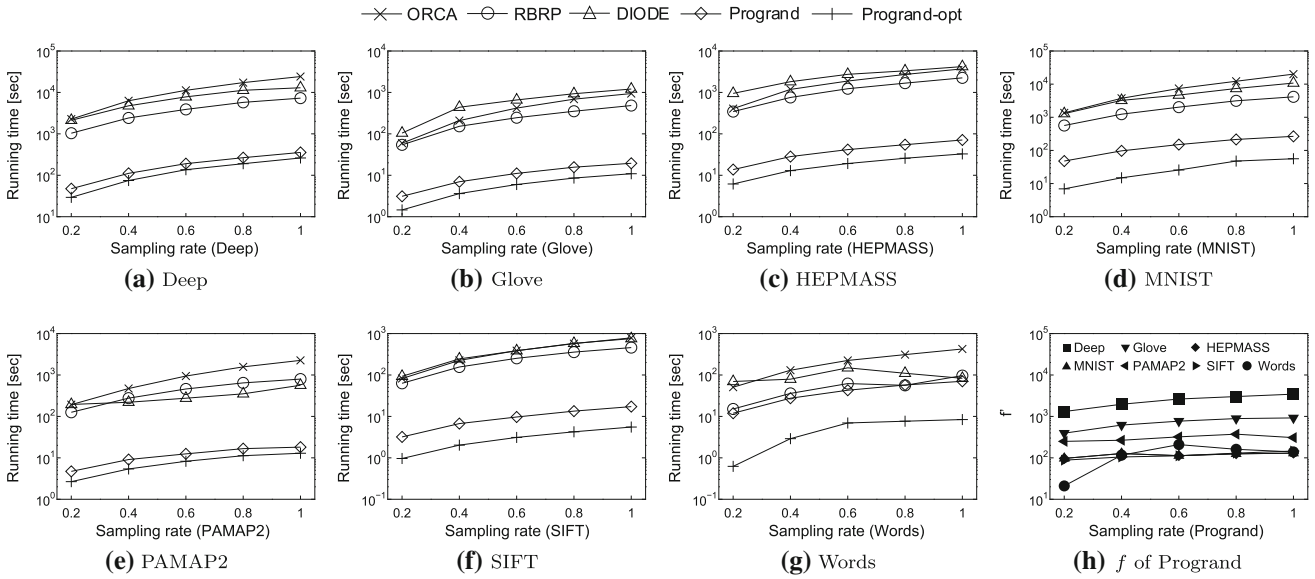


Fig. 13 Impact of n on the (N, k)-DOD problem

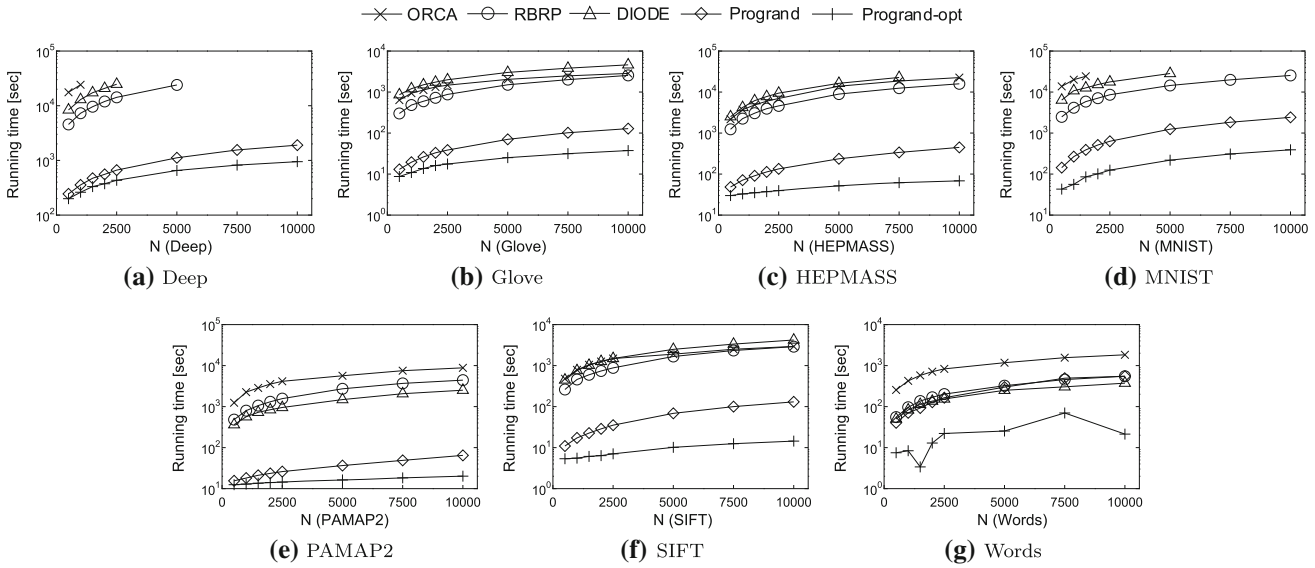


Fig. 14 Impact of N

Prograng-opt depends on f', i.e., the initialized τ. This is also affected by N, thus, when τ is very accurate (or not so accurate), its running time becomes faster (see the result of $N = 1500$) or a bit slower (see the result of $N = 7500$).

Varying k Figure 15 studies the impact of k. First, we see that the state-of-the-art algorithms are not affected much by k. Since they rely on nested-loop, computing $udist_k(\cdot)$ incurs $O(n)$ time, so this result is understandable. On the other hand, the performance of Prograng(-opt) is affected by k. For fixed N and n, the time complexity of Prograng(-opt) is $O(k + f')$. However, the empirical results show that the running time of Prograng normally grows sub-linearly to k. This suggests that the impact of k is practically not so strong, compared with the theoretical result.

Varying the number of threads Last, we study the impact of the number of threads. Figure 16 shows the results on Glove, HEPMASS, PAMAP2, and Words. It can be seen that, as well as the nested-loop-based state-of-the-art algorithms, Prograng(-opt) receives benefit from multi-threading and its running time decreases with increasing the number of available threads. Thanks to this property, Prograng(-opt) keeps outperforming the state-of-the-art algorithms. Notice that Prograng(-opt) can detect outliers quickly even with a single thread, whereas the other algorithms with a single

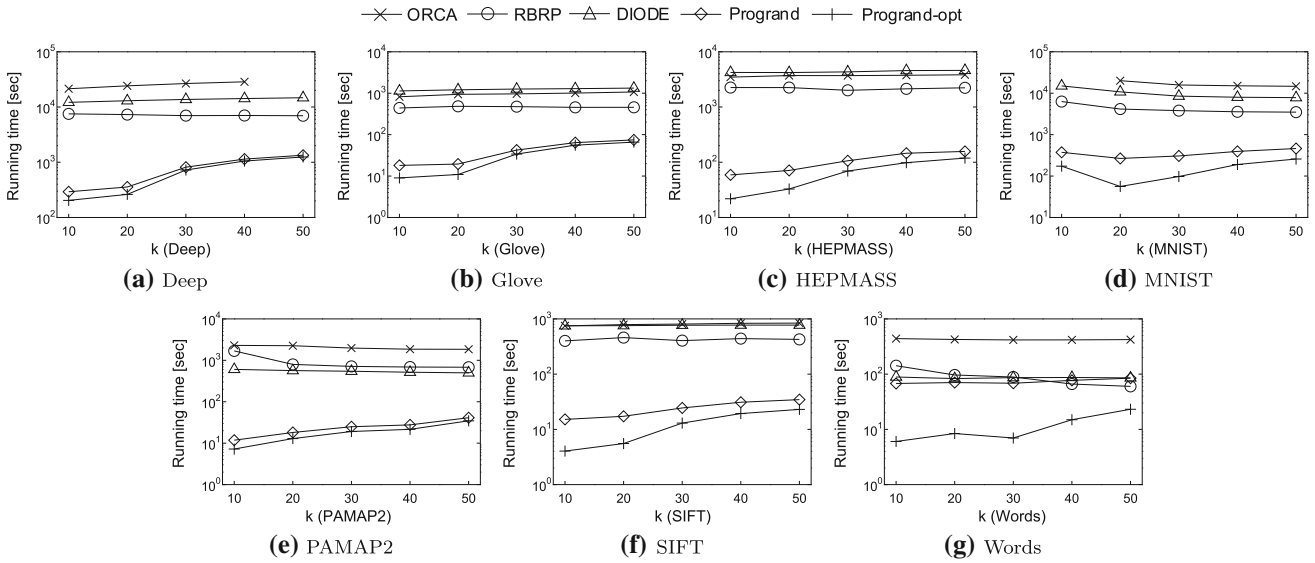


Fig. 15 Impact of k on the (N, k)-DOD problem

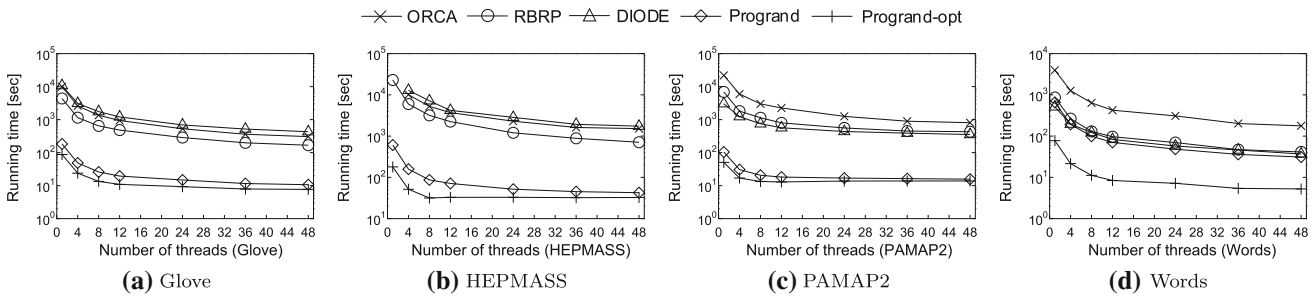


Fig. 16 Impact of the number of threads on the (N, k)-DOD problem

thread need much longer time or cannot terminate (N, k)-DOD within the time limit.

7 Conclusion

In this article, we addressed the problem of distance-based outlier detection in metric spaces. We proposed proximity graph-based algorithms. For the (r, k)-DOD problem, we proposed a proximity graph-based algorithm. To optimize this algorithm, we devised MRPG (Metric Randomized Proximity Graph), which improves reachability to neighbors and reduces the verification cost. For the (N, k)-DOD problem, we proposed Prograng, which is also a proximity graph-based algorithm, and this algorithm needs nearly $O(Nn)$ time in practice. Our experiments on real datasets confirm that our algorithms are much faster than state-of-the-art.

Some DOD works, e.g., [54], consider that multiple parameters are given in a batch. How to process (r, k)-DOD or (N, k)-DOD with multiple parameters efficiently (in a batch) is an open problem to be worth investigating.

Acknowledgements This research is partially supported by JST PRESTO Grant Number JPMJPR1931, JSPS Grant-in-Aid for Scientific Research (A) Grant Number 18H04095, and JST CREST Grant Number JPMJCR21F2.

References

1. https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.IsolationForest.html
2. https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/multiclass.html

3. http://corpus-texmex.irisa.fr/
4. https://github.com/dwyl/english-words
5. Aggarwal, C.C.: Outlier analysis. In: Data Mining, pp. 237–263 (2015)
6. Amagata, D., Hara, T.: Fast density-peaks clustering: multicore-based parallelization approach. In: SIGMOD, pp. 49–61 (2021)
7. Amagata, D., Onizuka, M., Hara, T.: Fast and exact outlier detection in metric spaces: a proximity graph-based approach. In: SIGMOD, pp. 36–48 (2021)
8. Amagata, D., Onizuka, M., Hara, T.: Fast and exact outlier detection in metric spaces: a proximity graph-based approach (full version) (2021). arXiv:2110.08959
9. Angiulli, F., Fassetti, F.: Dolphin: an efficient algorithm for mining distance-based outliers in very large datasets. ACM Trans. Knowl. Data Discov. **3**(1), 4 (2009)
10. Angiulli, F., Pizzuti, C.: Fast outlier detection in high dimensional spaces. In: European Conference on Principles of Data Mining and Knowledge Discovery, pp. 15–27 (2002)
11. Arya, S., Mount, D.M.: Approximate nearest neighbor queries in fixed dimensions. In: SODA, vol. 93, pp. 271–280 (1993)
12. Babenko, A., Lempitsky, V.: Efficient indexing of billion-scale datasets of deep descriptors. In: CVPR, pp. 2055–2063 (2016)
13. Baldi, P., Cranmer, K., Faucett, T., Sadowski, P., Whiteson, D.: Parameterized machine learning for high-energy physics (2016). arXiv preprint arXiv:1601.07913
14. Batista, G.E., Prati, R.C., Monard, M.C.: A study of the behavior of several methods for balancing machine learning training data. SIGKDD Explor. Newslett. **6**(1), 20–29 (2004)
15. Bay, S.D., Schwabacher, M.: Mining distance-based outliers in near linear time with randomization and a simple pruning rule. In: KDD, pp. 29–38 (2003)
16. Boguna, M., Krioukov, D., Claffy, K.C.: Navigability of complex networks. Nat. Phys. **5**(1), 74 (2009)
17. Boukerche, A., Zheng, L., Alfandi, O.: Outlier detection: methods, models, and classification. ACM Comput. Surv. **53**(3), 1–37 (2020)
18. Breunig, M.M., Kriegel, H.P., Ng, R.T., Sander, J.: Lof: identifying density-based local outliers. In: SIGMOD, pp. 93–104 (2000)
19. Campos, G.O., Zimek, A., Sander, J., Campello, R.J., Micenková, B., Schubert, E., Assent, I., Houle, M.E.: On the evaluation of unsupervised outlier detection: measures, datasets, and an empirical study. Data Min. Knowl. Disc. **30**(4), 891–927 (2016)
20. Chen, L., Gao, Y., Zheng, B., Jensen, C.S., Yang, H., Yang, K.: Pivot-based metric indexing. PVLDB **10**(10), 1058–1069 (2017)
21. Cui, B., Coi, B.C., Su, J., Tan, K.L.: Indexing high-dimensional data for efficient in-memory similarity search. IEEE Trans. Knowl. Data Eng. **17**(3), 339–353 (2005)
22. Dearholt, D., Gonzales, N., Kurup, G.: Monotonic search networks for computer vision databases. In: ACSSC, vol. 2, pp. 548–553 (1988)
23. Dong, W., Moses, C., Li, K.: Efficient k-nearest neighbor graph construction for generic similarity measures. In: WWW, pp. 577–586 (2011)
24. Fu, C., Xiang, C., Wang, C., Cai, D.: Fast approximate nearest neighbor search with the navigating spreading-out graph. PVLDB **12**(5), 461–474 (2019)
25. Ghoting, A., Parthasarathy, S., Otey, M.E.: Fast mining of distance-based outliers in high-dimensional datasets. In: SDM, pp. 609–613 (2006)
26. Ghoting, A., Parthasarathy, S., Otey, M.E.: Fast mining of distance-based outliers in high-dimensional datasets. Data Min. Knowl. Disc. **16**(3), 349–364 (2008)
27. Gu, X., Akoglu, L., Rinaldo, A.: Statistical analysis of nearest neighbor methods for anomaly detection. In: NeurIPS, pp. 10923–10933 (2019)
28. Harwood, B., Drummond, T.: Fanng: fast approximate nearest neighbour graphs. In: CVPR, pp. 5713–5722 (2016)
29. Hodge, V., Austin, J.: A survey of outlier detection methodologies. Artif. Intell. Rev. **22**(2), 85–126 (2004)
30. Ilyas, I.F., Chu, X.: Data Cleaning. Morgan & Claypool (2019)
31. Kim, J., Scott, C.D.: Robust kernel density estimation. J. Mach. Learn. Res. **13**(1), 2529–2565 (2012)
32. Knorr, E.M., Ng, R.T.: Algorithms for mining distance-based outliers in large datasets. In: VLDB, vol. 98, pp. 392–403 (1998)
33. Kontaki, M., Gounaris, A., Papadopoulos, A.N., Tsichlas, K., Manolopoulos, Y.: Continuous monitoring of distance-based outliers over data streams. In: ICDE, pp. 135–146 (2011)
34. Krauthgamer, R., Lee, J.R.: Navigating nets: Simple algorithms for proximity search. In: SODA, pp. 798–807 (2004)
35. Kriegel, H.P., Schubert, M., Zimek, A.: Angle-based outlier detection in high-dimensional data. In: KDD, pp. 444–452 (2008)
36. Larson, S., Mahendran, A., Lee, A., Kummerfeld, J.K., Hill, P., Laurenzano, M.A., Hauswald, J., Tang, L., Mars, J.: Outlier detection for improved data quality and diversity in dialog systems. In: NAACL-HLT, pp. 517–527 (2019)
37. Lerman, G., Maunu, T.: An overview of robust subspace recovery. Proc. IEEE **106**(8), 1380–1410 (2018)
38. Li, W., Zhang, Y., Sun, Y., Wang, W., Li, M., Zhang, W., Lin, X.: Approximate nearest neighbor search on high dimensional data—experiments, analyses, and improvement. IEEE Trans. Knowl. Data Eng. **32**(8), 1475–1488 (2019)
39. Liu, F.T., Ting, K.M., Zhou, Z.H.: Isolation forest. In: ICDM, pp. 413–422 (2008)
40. Malkov, Y., Ponomarenko, A., Logvinov, A., Krylov, V.: Approximate nearest neighbor algorithm based on navigable small world graphs. Inf. Syst. **45**, 61–68 (2014)
41. Malkov, Y.A., Yashunin, D.: Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. IEEE Trans. Pattern Anal. Mach. Intell. **42**(4), 824–836 (2020)
42. Orair, G.H., Teixeira, C.H., Meira, W., Jr., Wang, Y., Parthasarathy, S.: Distance-based outlier detection: consolidation and renewed bearing. PVLDB **3**(1–2), 1469–1480 (2010)
43. Pennington, J., Socher, R., Manning, C.D.: Glove: Global vectors for word representation. In: EMNLP, pp. 1532–1543 (2014)
44. Perdacher, M., Plant, C., Böhm, C.: Cache-oblivious high-performance similarity join. In: SIGMOD, pp. 87–104 (2019)
45. Ramaswamy, S., Rastogi, R., Shim, K.: Efficient algorithms for mining outliers from large data sets. In: SIGMOD, pp. 427–438 (2000)
46. Reiss, A., Stricker, D.: Introducing a new benchmarked dataset for activity monitoring. In: ISWC, pp. 108–109 (2012)
47. Sun, Y., Wang, W., Qin, J., Zhang, Y., Lin, X.: Srs: solving c-approximate nearest neighbor queries in high dimensional euclidean space with a tiny index. PVLDB **8**(1), 1–12 (2014)
48. Tao, Y., Xiao, X., Zhou, S.: Mining distance-based outliers from large databases in any metric space. In: KDD, pp. 394–403 (2006)
49. Tran, L., Fan, L., Shahabi, C.: Distance-based outlier detection indata streams. PVLDB **9**(12), 1089–1100 (2016)
50. Wang, H., Bah, M.J., Hammad, M.: Progress in outlier detection techniques: a survey. IEEE Access **7**, 107964–108000 (2019)
51. Wang, Y., Liu, W., Ma, X., Bailey, J., Zha, H., Song, L., Xia, S.T.: Iterative learning with open-set noisy labels. In: CVPR, pp. 8688–8696 (2018)
52. Yianilos, P.N.: Data structures and algorithms for nearest neighbor search in general metric spaces. In: SODA, pp. 311–321 (1993)
53. Yoon, S., Lee, J.G., Lee, B.S.: Nets: Extremely fast outlier detection from a data stream via set-based processing. PVLDB **12**(11), 1303–1315 (2019)
54. Yoon, S., Shin, Y., Lee, J.G., Lee, B.S.: Multiple dynamic outlier-detection from a data stream by exploiting duality of data and queries. In: SIGMOD, pp. 2063–2075 (2021)

55. Zhang, H., Chen, G., Ooi, B.C., Tan, K.L., Zhang, M.: In-memory big data management and processing: a survey. IEEE Trans. Knowl. Data Eng. **27**(7), 1920–1948 (2015)
56. Zois, V., Tsotras, V.J., Najjar, W.A.: Efficient main-memory top-k selection for multicore architectures. PVLDB **13**(2), 114–127 (2019)