Real-Time Distance-Based Outlier Detection in Data Streams

Luan Tran
University of Southern California
Los Angeles, CA
luantran@usc.edu

Min Y. Mun
University of Southern California
Los Angeles, CA
mym_286@usc.edu

Cyrus Shahabi
University of Southern California
Los Angeles, CA
shahabi@usc.edu

ABSTRACT

Real-time outlier detection in data streams has drawn much attention recently as many applications need to be able to detect abnormal behaviors as soon as they occur. The arrival and departure of streaming data on edge devices impose new challenges to process the data quickly in real-time due to memory and CPU limitations of these devices. Existing methods are slow and not memory efficient as they mostly focus on quick detection of inliers and pay less attention to expediting neighbor searches for outlier candidates. In this study, we propose a new algorithm, CPOD, to improve the efficiency of outlier detections while reducing its memory requirements. CPOD uses a unique data structure called "core point" with multi-distance indexing to both quickly identify inliers and reduce neighbor search spaces for outlier candidates. We show that with six real-world and one synthetic dataset, CPOD is, on average, 10, 19, and 73 times faster than M_MCOD, NETS, and MCOD, respectively, while consuming low memory.

PVLDB Reference Format:
Luan Tran, Min Y. Mun, and Cyrus Shahabi. Real-Time Distance-Based Outlier Detection in Data Streams. PVLDB, 14(2): 141 - 153, 2021.
doi:10.14778/3425879.3425885

1 INTRODUCTION

Outlier detection is the task of finding data points that do not conform to an expected behavior in a dataset. With the expansion of data streaming across a broad range of applications, e.g., fraud detection in banking, defect detection in manufacturing, and abnormal vitals detection in healthcare, detecting outliers in data streams is receiving much attention. All these applications require unusual events to be recognized the moment they happen. For example, in real-time ECG monitors, abnormal heartbeats must be detected as soon as possible to reduce the mortality of patients. Furthermore, there is an increasing demand to conduct outlier detection on edge devices such as security cameras, routers, smartphones, and wearable devices [2, 7, 14] that have limited memory capacity as well as low processing power. This paper focuses on Distance-based Outlier Detection in Data Streams (DODDS) with low CPU and memory requirements. In any given dataset, a distance-based outlier is any data point that has less than K neighbors within a distance of R.

Distance-based outlier detection was initially studied for static datasets [10]. Later, it was shown that it can discover abnormalities in data streams with high accuracy, such as abnormal heartbeats from ECG streams [13].

Because the size of data streams is potentially unbounded, outlier detection is conducted for a sliding window, i.e., the set of active data objects. This setting ensures maximum efficiency in computation time and memory usage, as well as detecting the outliers within the most recent context. In this setting, the outlier status of each data point can change when a new slide arrives. More specifically, an inlier can become an outlier if its neighbors expire, while an outlier can become an inlier if it gains new neighbors. This characteristic of DODDS presents a challenge in terms of monitoring the status of every active data point for each sliding window. A number of algorithms have been proposed for DODDS, such as ExactStorm [1], LEAP [4], AbstractC [17], MCOD [12], LUE [12], DUE [12], NETS [18], and M_MCOD [16]. Among these studies, MCOD, M_MCOD, and NETS have performed best in processing times and memory requirements.

The central idea of these methods is to quickly identify inliers using unique data structures such as micro-cluster (MCOD, M_MCOD) and inlier cell (NETS). These data structures store data points that are neighbors of each other and in the range R/2 from their centers. They maintain at least K+1 data points to guarantee that all members are inliers. This approach eliminates the need for explicit neighbor searches of the members. However, for the remaining data points that are not in any micro-cluster or inlier cell, linear neighbor searches are performed, which can be very inefficient. For example, in Figure 1, no outlier exists when K = 3; however, these methods require neighbor searches for all data points because for which neither micro-cluster nor inlier cell exists. Furthermore, NETS requires much memory for high dimensional data to maintain cells using a grid-based index structure.

To address these challenges, we propose the Core Point-based Outlier Detection (CPOD) algorithm, which employs a new data structure called "core point." Core points use multi-distance indexing to both identify inliers quickly and reduce neighbor search spaces for outlier candidates. Additionally, CPOD employs the minimal probing principle [4] to find optimal neighbor sets for data points, further reducing unnecessary neighbor searches. The number of core points is relatively insignificant compared to the size of datasets (see Table 1), and thus CPOD consumes low memory.

Performance Improvement. With extensive experiments on synthetic and real-world datasets, we observe that, on average, CPOD is approximately 10, 19, and 73 times faster than M_MCOD, NETS, and MCOD, respectively. CPOD also requires about three times less memory than NETS and comparable memory to MCOD and M_MCOD. For all the datasets with default settings, CPOD requires less than 0.05 seconds to process one sliding window, whereas in some cases other methods require more than 0.43 seconds. In particular, for one of our datasets (EM), CPOD takes 0.02 seconds to process one sliding window, while other methods take 30 to 200 times longer.

Proceedings of the VLDB Endowment, Vol. 14, No. 2 ISSN 2150-8097.
doi:10.14778/3425879.3425885

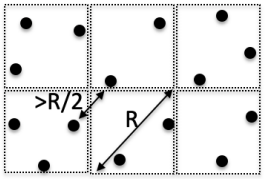


Figure 1: Example data with K = 3. No micro-cluster exists because the distance between any pair of data points is larger than R/2. No inlier cell exists because all the cells contain at most 3 data points.

The remainder of this paper is organized as follows. In Section 2, we formally define the DODDS problem and provide an overview of the state-of-the-art algorithms. In Section 3, we introduce our proposed algorithm, CPOD. In Section 4, we present our evaluation results in detail. In Section 5, we discuss different problem settings. We conclude the paper with discussion and future research directions in Section 6.

2 BACKGROUND

In this section, we present the formal definition of DODDS and an overview of the state-of-the-art algorithms.

2.1 Problem Definition

To define the problem, we first explain important concepts in distance-based outlier detection and data streams. Given a distance function which is defined in a metric space, neighbors and outliers are defined as follows.

Definition 2.1 (Neighbor). Two data points are neighbors of each other if their distance is not greater than R. A data point is not considered a neighbor of itself.

Definition 2.2 (Distance-based Outlier). Given a dataset $\mathbb{D}$, a distance threshold R, and a neighbor count threshold K, a distance-based outlier in $\mathbb{D}$ is a data point that has fewer than K neighbors in $\mathbb{D}$.

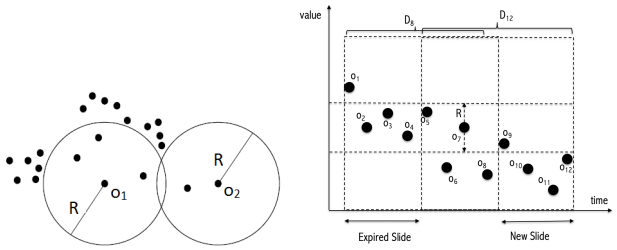
A data point that has at least K neighbors is called an *inlier*. Figure 2(a) shows an example of a dataset from [12, 15] that has two outliers with K = 4. Data points o_1 and o_2 are outliers since they have three and one neighbors, respectively.

Definition 2.3 (Data Stream). A data stream is a possibly infinite series of data points $..., o_{n-2}, o_{n-1}, o_n, ...$, which are sorted in increasing order of time.

Since the size of data streams is potentially unbounded, data streams are typically processed in a *sliding window*, i.e., a set of active data points. In this study, we adopt the *count-based window* setting in data streams, as in the previous work [1, 4, 12, 15, 18], which is defined as follows.

Definition 2.4 (Count-based Window). Given a positive number n and a fixed window size W, the count-based window $\mathbb{D}_n$ is the set of W data points: $o_{n-W+1}, o_{n-W+2}, ..., o_n$.

Given the window size W, all count-based windows have the same number of data points. Therefore, we can control the volume of data that we process at a time and fairly evaluate the *scalability* of the DODDS algorithms. We use the term *window* to refer to the *count-based window*. In this setting, *a slide* represents the set of data points that expire and arrive in the same batch for processing. Every slide contains the same number of data points. The slide size S, which is fixed, characterizes the speed of the data streams. Every time a new slide of S data points arrives, it is added into the window, while the slide of the oldest S data points *expires* and is discarded. Figure 2(b) shows an example of two consecutive windows with W = 8 and S = 4. The x-axis reports the arrival time of data points and the y-axis reports the data values. When the window $\mathbb{D}_8$ slides, four data points $\{o_9, o_{10}, o_{11}, o_{12}\}$ arrive in a new slide, and four data points $\{o_1, o_2, o_3, o_4\}$ expire and are removed from the current window.



(a) Outlier detection in static dataset with K = 4.

(b) Example of DODDS with K = 4, W = 8, and S = 4.

Figure 2: Outlier detection in data stream and static dataset.

As the data points in a stream arrive and expire in slides, it is important to distinguish between the following two concepts: *preceding neighbor* and *succeeding neighbor*. For each data point o, its *preceding neighbors* are neighbors that expire before it. On the other hand, its *succeeding neighbors* are neighbors that expire in the same slide or after it. Figure 2(b) illustrates the sliding window concept and how it affects the outlier statuses of the data points. For example, in this figure, o_7 has one succeeding neighbor o_5 and three preceding neighbors, i.e., o_2, o_3, and o_4.

The Distance-based Outlier Detection in Data Streams (DODDS) is defined as follows.

PROBLEM 1 (DODDS). *Given the window size W, the slide size S, the distance threshold R, and the neighbor count threshold K, DODDS detects the distance-based outliers in every sliding window* $..., \mathbb{D}_n, \mathbb{D}_{n+S}, ...$

The challenge of DODDS is that the expired and newly arrived neighbors can affect the outlier statuses of existing data points. For example, in Figure 2(b), in $\mathbb{D}_8$, o_7 is an inlier as it has 4 neighbors, i.e., o_2, o_3, o_4 and o_5. In $\mathbb{D}_{12}$, o_7 becomes an outlier because o_2, o_3, and o_4 expired and o_7 has only two neighbors, i.e., o_5 and o_9. Note that an inlier that has at least K succeeding neighbors never becomes an outlier in the future. Those inliers are thus called *safe inliers*. On the other hand, the inliers that have less than K succeeding neighbors are *unsafe inliers*, as they may become outliers when the context changes and their preceding neighbors expire.

Several fundamental approaches, i.e., Exact-Storm [1], LUE [12], and DUE [12], use an M-Tree [5] for indexing data points in sliding windows to reduce the time of finding neighbors. M-Tree is efficient in neighbor search, however, inserting and removing data points from M-Tree is time-consuming. In [15, 16, 18], MCOD, M_COD, and NETS with their unique data structures were shown to outperform Exact-Storm, LUE, DUE, and LEAP [3] in CPU time and memory requirement.

2.2 Micro-cluster based Algorithms - MCOD and M_MCOD

2.2.1 MCOD. In order to find if a data point is an outlier, we need to check if it has K neighbors. Neighbor searches can be significantly expensive when carried out for every data point. MCOD [11] introduced the concept of micro-clusters to reduce the need for neighbor searches. A micro-cluster is composed of at least K + 1 data points. It is centered at one data point and has a radius of R/2. According to the triangular inequality, the distance between every pair of data points in a micro-cluster is not greater than R. Therefore, every data point in a micro-cluster is an inlier. Figure 3(a) shows an example of three micro-clusters, i.e., MC_1, MC_2, and MC_3, and different symbols represent data points in each micro-cluster. Some data points may not fall into any micro-clusters and are stored in a potential outlier (PD) list.

For any data point *o*, it can be added to an existing micro-cluster, become the center of its micro-cluster, or be added to PD. For instance, if *o* is within the distance of R/2 to the center of a micro-cluster *MC*, MCOD adds *o* to *MC*. Otherwise, MCOD searches for the neighbors of *o* in PD with the range of R/2. If MCOD finds at least K neighbors, it forms a new micro-cluster with *o* as the cluster center. Otherwise, MCOD adds *o* to PD and finds all neighbors for *o*. MCOD uses a linear search to find neighbors in PD. To find neighbors in micro-clusters, according to the triangular inequality, MCOD only searches in micro-clusters whose centers are in the range of 3R/2 from *o*. Since MCOD does not store the neighbor information of the data points in micro-clusters, it is efficient in memory usage.

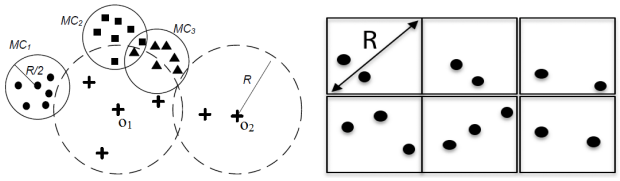
Drawbacks of MCOD: 1) MCOD performs poorly when most data points do not have K neighbors in the range of R/2 and are stored in PD because a large PD list is inefficient for neighbor searches. 2) MCOD uses a linear search in all existing micro-clusters when finding a micro-cluster for a data point, which is also inefficient. 3) MCOD finds all the neighbors for each data point in PD, which can incur unnecessary neighbor searches.

2.2.2 M_MCOD. M_MCOD improves MCOD by incorporating the *minimal probing principle* [4]. More specifically, with M_MCOD, the neighbor search for a data point *o* stops when it finds K neighbors. Moreover, it utilizes the continuity characteristics of data streams by first finding neighbors in the data points whose arrival times are close to the arrival time of *o*. The succeeding slides of *o* are checked first, and then the preceding slides to ensure the neighbors are most recent. When some neighbors of *o* expire and *o* does not have enough neighbors, M_MCOD refinds the neighbors for *o* in the slides that have not been searched. To accommodate these processes, for PD and each micro-cluster, M_MCOD employs a map of slides and their corresponding data points to retrieve data points given a slide index quickly.

Drawbacks of M_MCOD: M_MCOD improves MCOD by solving its third drawback. However, M_MCOD still has the first two drawbacks of MCOD.

2.3 Cell-based Algorithm - NETS

NETS employs a grid-based index structure to form cells that are hyper-cubes in high dimensional space. The diagonal of each cell is R, which forces the data points in a cell to be neighbors of each other. This grid-based index structure is also used to monitor the data points having at least a pre-defined number of neighbors in dynamic density-based clustering problem [9]. Figure 3(b) illustrates 6 example cells in 2-dimensional data. Every data point is placed in a cell. By using the grid-based index structure, NETS can quickly assign a data point to a corresponding cell and efficiently monitor the number of data points in every cell. If a cell has at least K + 1 data points, it is called *inlier cell* because all the data points inside it are inliers. NETS also provides a technique to determine *outlier cells* that contain all outliers quickly. More specifically, if the total number of data points in neighboring cells within the range 2R from the current cell is not greater than K, the current cell is an *outlier cell*. For the data points in *undetermined cells*, NETS performs neighbor searches to find all their neighbors.



(a) Three micro-clusters with K = 4. (b) Two *inlier cells* and four *undetermined cells* with K = 2.

Figure 3: Example micro-clusters and cells.

Drawbacks of NETS: 1) Cells are placed on a grid-based index structure, and the diagonal of each cell is R. Hence, they do not cover all neighbors in the range of R/2 from their centers and there is a higher chance that a cell does not contain all inliers. 2) In cell-level (detecting inlier and outlier cells) and point-level (detecting outlier data points) outlier detection steps, NETS uses a linear search over cells to find neighboring cells, which is inefficient. Therefore, when data has a *low concentration ratio* [18], i.e., data points do not have many neighbors in the range R/2, NETS generates few inlier cells and incurs high CPU running time. 3) For each data point *d* in *undetermined cells*, NETS does not apply the *minimal probing principle* and finds all neighbors for *d*, which can incur unnecessary neighbor searches. 4) With high dimensional data, NETS requires much memory for maintaining cells using a grid-based index structure.

3 CORE POINT-BASED OUTLIER DETECTION - CPOD

Motivated by the drawbacks of micro-cluster based (MCOD, M_MCOD) and cell-based (NETS) algorithms, we propose CPOD that employs a new data structure called core point to expedite neighbor searches for outlier candidates. A core point is a special data point that stores its distances to other data points in multiple ranges. In CPOD, every sliding window has a set of core points to expedite the neighbor search. Using core points, similar to MCOD, CPOD can also quickly identify inliers, which are at least $K + 1$ data points in the range $R/2$ from a core point. But different from MCOD, for the remaining data points which are outlier candidates, CPOD uses core points within each slide to perform neighbor searches. More specifically, for each outlier candidate, CPOD searches for neighbors in surrounding slides to find enough neighbors. To find neighbors in each slide, CPOD uses its corresponding core points with some pruning techniques to reduce neighbor search spaces. In this section, we first introduce the core point concept

and related pruning techniques, which are the essential parts of CPOD. Subsequently, we discuss the corresponding data structures and the complete algorithm.

3.1 Core Point Overview

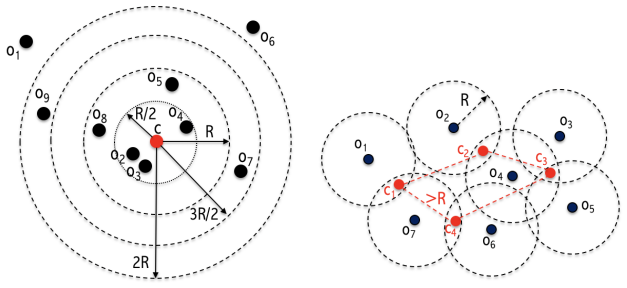
3.1.1 Core Point Definition. The triangular inequality is commonly used in reducing the neighbor search space. To simplify the discussion, let us focus on a single slide and denote its dataset as $\mathbb{S}$. Moreover, let us call the data point for which we are finding neighbors in a slide *a query point* q. Given $q, \mathbb{S}$ and two other data points in $\mathbb{S}$, i.e., $c, p \in \mathbb{S}$, we have $d(c,p) \leq d(p,q) + d(q,c)$ and $d(c,p) \geq d(q,c) - d(p,q)$. Therefore, if p is a neighbor of q, we have $d(c,p) \leq R + d(q,c)$, and $d(c,p) \geq d(q,c) - R$. In other words, if $d(q,c)$ is known, the neighbor p of q is in the range $(d(q,c) - R)$ to $(R + d(q,c))$ from c. Hence, data point c with its corresponding distances to other data points in $\mathbb{S}$ can be used to produce the reduced neighbor search for q. We define such a data point c as a core point.

However, computing and storing the distances to all other data points are time and memory consuming. Note that by the definition of outlier, most of the data points have at least K neighbors and are inliers. For example, if the outlier rate is 1%, 99% of data points have at least K neighbors, which can be set to be core points. Therefore, for each core point c, CPOD stores only the data points in the range $2R$ from c, which is sufficient to provide the reduced neighbor search spaces for the neighbors of c. Consequently, given a dataset $\mathbb{S}$, we formally define a core point as follows.

Definition 3.1 (Core Point). A core point c is a data point that stores the lists of data points, $\mathbb{E}(c)$, in different distance ranges from it as follows.

$$\mathbb{E}(c) = \mathbb{E}_0(c) \cup \mathbb{E}_1(c) \cup \mathbb{E}_2(c) \cup \mathbb{E}_3(c)$$

where $\mathbb{E}_k(c) = \{p \in \mathbb{S} | kR/2 < d(c,p) \leq (k+1)R/2\}$, $k \in \{0,1,2,3\}$. $\mathbb{E}_k(c)$ is the list of the data points in $\mathbb{S}$ whose distances to c are larger than $kR/2$ and less than or equal to $(k+1)R/2$.



(a) Example core point c.

(b) Example supporting core point set $\mathbb{C} = \{c_1, c_2, c_3, c_4\}$.

Figure 4: Example core point and supporting core point set.

Figure 4(a) illustrates an example of a core point with its $\mathbb{E}$ lists in different levels, $\mathbb{E}_0(c) = \{o_2, o_3, o_4\}$, $\mathbb{E}_1(c) = \{o_5, o_8\}$, $\mathbb{E}_2(c) = \{o_7\}$, $\mathbb{E}_3(c) = \{o_9\}$.

Let us denote *d-associate* of a core point c as the list of data points in $\mathbb{S}$ whose distances to c are not greater than d. We say that ***core point*** c ***supports the data point*** p ***or*** p ***is linked to*** c if p is in the *R-associate* of c, or in other words, $p \in \mathbb{E}_0(c) \cup \mathbb{E}_1(c)$. For example, in Figure 4(a), core point c supports data points o_2, o_3, o_4, o_5 and o_8.

Every data point should be supported in neighbor search. Therefore, we define the supporting core point set of a data set as follows.

Definition 3.2 (Supporting Core Point Set). The supporting core point set $\mathbb{C}$ of a data set $\mathbb{S}$ consists of core points such that:

- The distance between any pair of core points is greater than R. In other words, $\forall c_1 \neq c_2 \in \mathbb{C}$, $d(c_1, c_2) > R$.
- Each data point $p \in \mathbb{S}$ is linked to at least one core point c. In other words, $\forall p \in \mathbb{S}, \exists c \in \mathbb{C} : p \in \mathbb{E}_0(c) \cup \mathbb{E}_1(c)$.

The first constraint limits the size of the core point set $\mathbb{C}$ while the second constraint guarantees that every data point in $\mathbb{S}$ is supported. Figure 4(b) illustrates an example of supporting core point set (red points) of a data set (black points). In this figure, for each data point o_i, $1 \leq i \leq 7$, there exists at least one core point within a distance of R. Table 1 shows the number of supporting core points for all sliding windows in the real-world datasets we examine in this study. We observed that for each dataset, a small number of core points, i.e., less than 0.5% of the size of data set, can support all the data points. With the window size set to 10,000 for small data sets, i.e., FC and TAO, and 100,000 for large datasets, i.e., GAS, EM, HPC, and STK, we observed that the average number of supporting core points for each window is less than 1% the window size.

3.1.2 Pruning Techniques using Core Point. CPOD performs neighbor searches for outlier candidates in individual slides to verify their outlier statuses. Therefore, with the supporting core point set for each slide, CPOD utilizes the following pruning techniques, which are applicable for any set of data, to obtain reduced search spaces. Given a supporting core point set $\mathbb{C}$ for a single slide with its data set $\mathbb{S}$, let q be a query point that is a data point for which we are finding neighbors, and $\mathbb{N}(q)$ be the neighbor set of q in $\mathbb{S}$. Based on an arbitrary distance from q to the core points in $\mathbb{C}$, CPOD has a corresponding reduced neighbor search space.

Theorem 3.3 (Instant Neighbor Confirmation). *If the distance between the query point q and a core point $c \in \mathbb{C}$ is less than or equal to $R/2$, all the data points in $\mathbb{E}_0(c)$ are neighbors of q. In other words, if $\exists c \in \mathbb{C}$ such that $d(c,q) \leq R/2$, we have $\mathbb{E}_0(c) \subseteq \mathbb{N}(q)$.*

Proof. According to the triangular inequality, for any data point $p \in \mathbb{E}_0(c)$, we have $d(p,q) \leq d(p,c) + d(q,c) \leq R/2 + R/2 = R$. Therefore, p and q are neighbors of each other. This completes our proof. □

Using Theorem 3.3, for each data point q, which has a core point in the range R/2, we simply have to count the data points in $\mathbb{E}_0(c)$ to get the number of neighbors of q in $\mathbb{E}_0(c)$. This characteristic is similar to counting neighbors in a cell in NETS or a micro-cluster in MCOD and M_MCOD.

Theorem 3.4 (Search Space Reduction 1). *If the distance between the query point q and a core point $c \in \mathbb{C}$ is not greater than $R/2$, all the neighbors of q are in the $3R/2$-associate of c. In other words, if $\exists c \in \mathbb{C}$ such that $d(c,q) \leq R/2$, we have $\mathbb{N}(q) \subseteq \mathbb{E}_0(c) \cup \mathbb{E}_1(c) \cup \mathbb{E}_2(c)$.*

Proof. According to the triangular inequality, for any neighbor $p \in \mathbb{S}$ of q, we have $d(p,c) \leq d(p,q) + d(q,c) \leq R + R/2 = 3R/2$. Therefore $\mathbb{N}(q) \subseteq \mathbb{E}_0(c) \cup \mathbb{E}_1(c) \cup \mathbb{E}_2(c)$. This completes our proof. □

Table 1: The number of core points for all sliding windows in real-world datasets.

Name	Description	Size	Dimensions	No. Core Points	Core Points/Dataset	Core Points/Window
GAS	Household gas sensors	0.9M	10	1626	0.18 %	0.31 %
FC	Forest cover types	0.55M	55	404	0.07 %	0.71 %
TAO	Oceanographic sensors	0.6M	3	258	0.04 %	0.31 %
EM	Gas sensor array	1.0M	16	3469	0.34 %	0.41 %
HPC	Electric power consumption	1.0M	7	381	0.04 %	0.32 %
STK	Stock trading records	1.1M	1	232	0.02 %	0.11 %

Theorem 3.4 provides the superset of the neighbors of q in $\mathbb{S}$ when q has a core point within a distance of $R/2$, $d(q,c) \leq R/2$.

Theorem 3.5 (Search Space Reduction 2). *If the distance between the query point q and a core point $c \in \mathbb{C}$ is not greater than R, all the neighbors of q are in $\mathbb{E}(c)$. In other words, if $\exists c \in \mathbb{C}$ such that $d(c,q) \leq R$, we have $\mathbb{N}(q) \subseteq \mathbb{E}(c)$.*

Proof. According to the triangular inequality, for any neighbor $p \in \mathbb{S}$ of q, we have $d(p,c) \leq d(p,q)+d(q,c) \leq R+R = 2R$. Therefore $p \in \mathbb{E}(c)$. This completes our proof. □

Theorem 3.5 provides the superset of the neighbors of the query point q if q has a core point c within a distance of R, $d(q,c) \leq R$.

Theorem 3.6 (No Neighbor Confirmation). *Let $\mathbb{C}_r^q = \{c \in \mathbb{C} | d(q,c) \leq r\}$ be the set of core points whose distances to q are not greater than r. All the neighbors of the query point q are in the R-associates of core points in $\mathbb{C}_{2R}^q$. In other words, $\mathbb{N}(q) \subseteq \bigcup_{c_i \in \mathbb{C}_{2R}^q} (\mathbb{E}_0(c_i) \cup \mathbb{E}_1(c_i))$. If the distance from q to any core point $c \in \mathbb{C}$ is greater than 2R, q has no neighbor in $\mathbb{S}$.*

Proof. Assume a data point $p \in \mathbb{S}$ is a neighbor of q, $d(p,q) \leq R$. Because each data point in $\mathbb{S}$ is linked to at least a core point, there exists a core point $c \in \mathbb{C}$ such that $d(c,p) \leq R$. Therefore, $d(q,c) \leq d(q,p) + d(c,p) \leq R + R = 2R \Rightarrow c \in \mathbb{C}_{2R}^q$. Also, we have $p \in \mathbb{E}_0(c) \cup \mathbb{E}_1(c)$. Therefore, $N(q) \subseteq \bigcup_{c_i \in \mathbb{C}_{2R}^q} (\mathbb{E}_0(c_i) \cup \mathbb{E}_1(c_i))$. Hence, if there is not any core point in the range $2R$ from q, $C_{2R}^q = \emptyset$, $\mathbb{N}(q) = \emptyset$, q does not have any neighbor in $\mathbb{S}$. This completes our proof. □

Theorem 3.6 provides the superset of the neighbors of q when q does not have any neighbor, which is a core point in $\mathbb{S}$, but there exist core points in the distance range 2R from q. On the other hand, if the distance from q to any core point in $\mathbb{C}$ is greater than 2R, q has no neighbor in $\mathbb{S}$. By using this theorem, CPOD can avoid unnecessary distance computations if q does not have any neighbor in $\mathbb{S}$.

Theorem 3.7 (Core Point Formation). *For any core point c, its 2R-associate is a subset of the union of its R-associate and the R-associates of other core points in the range 3R from it. In other words, $\mathbb{E}(c) \subseteq \bigcup_{c_i \in \mathbb{C}_{3R}^c \cup \{c\}} (\mathbb{E}_0(c_i) \cup \mathbb{E}_1(c_i))$.*

Proof. According to the definition of supporting core point set, every data point $p \in \mathbb{E}(c_1)$ is linked to at least one core point c_2 such that $d(p,c_2) \leq R$, core point c_2 can be the same as or different from core point c_1. Therefore, $d(c_1,c_2) \leq d(c_1,p) + d(p,c_2) \leq 2R + R = 3R$. Also, p is in the *R-associate* of c_2. Therefore, $\mathbb{E}(c) \subseteq \bigcup_{c_i \in \mathbb{C}_{3R}^c \cup \{c\}} (\mathbb{E}_0(c_i) \cup \mathbb{E}_1(c_i))$. This completes our proof. □

Core Point Formation. Theorem 3.7 is used to form the *2R-associate* of a core point. For a new set of data $\mathbb{S}$, the process of forming the set of core points supporting $\mathbb{S}$ can be separated into two steps. In the first step, we start with an empty core point set, $\mathbb{C} = \emptyset$. For every data point $p \in \mathbb{S}$, if p can be linked to an existing core point $c \in \mathbb{C}$, we add it to the *R-associate* of c. Otherwise, we create a new core point c with the same values as p then link p to c and add c to $\mathbb{C}$. In the second step, after every data point is linked to at least one core point, we find the *2R-associate* of every core point c in the *R-associates* of other core points in the range 3R from c. Each data point candidate is added to the corresponding $\mathbb{E}$ list of c based on its distance to c. Finally, $\mathbb{C}$ is outputted as the supporting core point set of $\mathbb{S}$.

3.2 Data Structure

In this section, we present the data structure for the two most important objects in CPOD, data point and core point.

Data Point. In each data point p, $p.neighborCount$, $p.predNeighborMap$, and $p.numSucNeighbors$ represent the total number of found neighbors, the counts of neighbors in preceding slides, and the number of succeeding neighbors, respectively. We use $p.closeCore$ to store the core point in the range of R/2 from p and $p.coreList$ to store the list of the core points that are linked to p. It helps CPOD to instantly access the reduced neighbor search spaces for p. Note that for each data point p, there is at most one core point in the range R/2 from it. This is because if there are at least two core points c_1 and c_2 in the range R/2 from p, according to the triangular inequality, the distance between c_1 and c_2 is not greater than R, and violates the definition of supporting core point set (Definition 3.2). CPOD also stores the last searched succeeding and preceding slides in $p.lastRight$ and $p.lastLeft$, respectively. They are used when CPOD refinds neighbors for p.

Core Point. In each core point c, each variable $c.\mathbb{E}_0, c.\mathbb{E}_1, c.\mathbb{E}_2, c.\mathbb{E}_3$ is a map of slide indices and the corresponding data points. For the example core point c in Figure 4(a), if (o_1, o_2, o_3), (o_4, o_5, o_6), and (o_7, o_8, o_9) are in slide $\mathbb{S}_1$, $\mathbb{S}_2$, and $\mathbb{S}_3$, respectively, we have $c.\mathbb{E}_0 = \{1 : (o_2, o_3),\ 2 : (o_4)\}$, $c.\mathbb{E}_1 = \{2 : (o_5),\ 3 : (o_8)\}$, $c.\mathbb{E}_2 = \{3 : (o_7)\}$, $c.\mathbb{E}_3 = \{3 : (o_9)\}$. CPOD also maintains a map of slides and their corresponding supporting core point sets for quickly retrieving core points during the neighbor search in

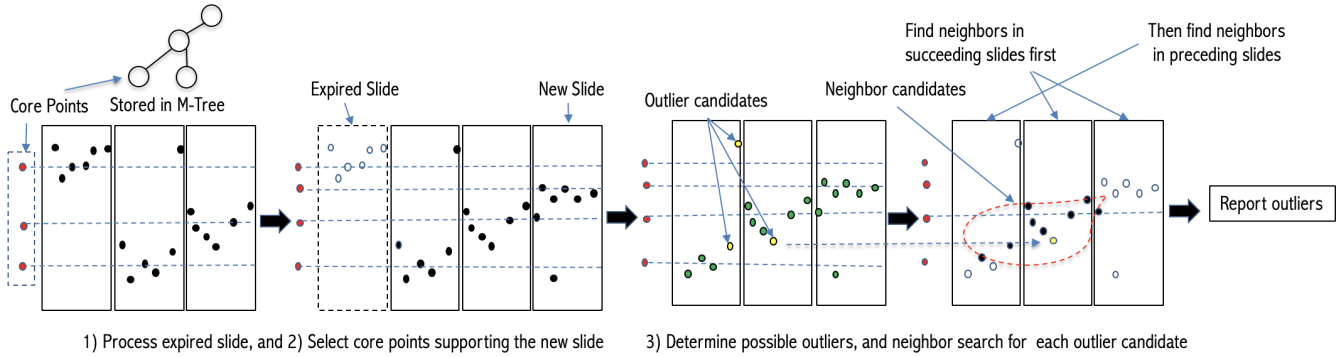


Figure 5: CPOD Algorithm Overview.

each slide. To accelerate the process of core point search, CPOD maintains distinct core points in an M-Tree [5].

3.3 Algorithm Details

Our proposed algorithm CPOD is illustrated in Figure 5 and Algorithm 1. It consists of three main procedures: processing expired slide, finding a supporting core point set for new data, and neighbor search for outlier candidates.

Expired Slide Processing. Every time a new slide $\mathbb{S}_{new}$ arrives, a slide $\mathbb{S}_{expired}$ expires. In function ProcessExpiredSlide(), line 5, Algorithm 1, CPOD removes the expired slide from the current window, decreases the neighbor counts of the data points by the number of expired neighbors, and eliminates the expired data points from the $\mathbb{E}$ lists of the existing core points.

Slide Indexing. CPOD selects a supporting core point set (Definition 3.2) for every slide. That is illustrated by function SelectCore() in Algorithm 2, which is called when a new slide arrives, line 6, Algorithm 1. This function is also called by function InitCore(), Algorithm 1, for initializing the supporting core points for the first window. To select core points supporting a slide, CPOD first links every data point p to a core point in the range R from it by conducting range queries on the M-Tree [5] which stores all the core points. For each data point p, if there exists a core point c in the range R from p, p is linked to c, line 5 and 9, Algorithm 2. If no core point is found, a new core point c is created with the same values of p, then p is linked to c, line 12, Algorithm 2. All the new core points are also added to the M-Tree for efficient range queries. Note that with this strategy, one core point can support data points in multiple slides. For each core point c supporting a slide, CPOD finds its *2R-associates* in the *R-associates* of other core points which are in the range $3R$ from c, according to Theorem 3.7. For each candidate p, based on its distance to c, CPOD adds it to the corresponding $\mathbb{E}$ list of c.

Fast inliers identification: After selecting core points that support the new slide, CPOD updates the number of data points in the $\mathbb{E}_0$ list of each core point, function UpdateHalfRCount(), line 7, Algorithm 1. For each core point c, if $c.\mathbb{E}_0$ has at least K + 1 data points, all the data points in $c.\mathbb{E}_0$ are inliers, according to Theorem 3.3.

Neighbor Search. For the data points that have been not determined as inliers in the previous step, CPOD runs neighbor searches to find their neighbors, function FindNeighbor(), Algorithm 3. In this function, CPOD first finds in the succeeding slides, line 1 to 6, then the preceding slides, line 7 to 12. During the neighbor search for data point q, by using $q.lastLeft$ and $q.lastRight$, CPOD only searches for neighbors in the slides that have not been checked before. In each slide s, CPOD first finds the core points that can be used for reducing neighbor search space, function SelectCore(), Algorithm 2. There are four possible scenarios: 1) There exists one core point in the range R/2 from q,

Algorithm 1: The Overall Procedure of CPOD

input : A data stream Σ, First window $\mathbb{D}$
output : Outliers in every sliding window
parameters : Window size W, Slide size S, Distance threshold R, Neighbor threshold K

```
1  allCores ← InitCore(𝔻)                                   // Init core points
   /* New slide 𝕊_new arrives with index s */
2  while (s, 𝕊_new) do
3      outliers ← ∅
4      𝕊_expired ← GetOldestSlide(𝔻, W, S)
5      ProcessExpiredSlide(𝕊_expired)
       /* Select core points for new slide */
6      allCores[s] ← SelectCore(𝕊_new)
7      UpdateHalfRCount(allCores[s])
8      for p ∈ 𝕊_new do
           /* Continue if closeCore has more than K data points
              in the range R/2. Theorem 3.3  */
9          if p.closeCore.countHalfR > K then
10             continue
           /* Neighbor search for outlier candidates in the new
              slide */
11         FindNeighbor(p)
12         if p.neighborCount < K then
13             outliers.add(p)
       /* re-find neighbors for outlier candidates in other
          slides */
14     for p ∈ W \ 𝕊_new do
15         if p.neighborCount < K then
16             FindNeighbor(p)
17             if p.neighborCount < K then
18                 outliers.add(p)
19     yield(outliers)
```

Algorithm 2: SelectCore($\mathbb{S}$)

```
input  : A slide 𝕊
output : Corresponding core points
static variables : Existing core points allCores
corePoints ← ∅                                   // Init core points
for p ∈ 𝕊 do
    /* Find core point for p in current core points */
    distance, c ← FindCore(p, corePoints)
    if c then // linked to a current core point
        AddToE(c, p, distance)
    else // find in the previous core points
        distance, c ← FindCore(p, allCores)
        if c then
            AddToE(c, p, distance)
            corePoints.add(c)
        else // create a new core point from p
            c ← NewCore(p)
            AddToE(c, p, 0)
            corePoints.add(c)
for c ∈ corePoints do
    for c2 ≠ c ∈ corePoints do
        distance ← ComputeDistance(c, c2)
        if distance ≤ 3R then
            /* Only need to check with data points linked to
               c2. Theorem 3.7 */
            CheckCoreWList(c, c2.E0)
            CheckCoreWList(c, c2.E1)
return corePoints
```

Algorithm 3: FindNeighbor(q)

```
input  : Data point q
/* First, search for neighbors in succeeding slides that q has
   not checked yet */
s_slides ← GetSucceedingSlides(q, checked = false)
for 𝕊 ∈ s_slides do
    FindNeighborInSlide(q, 𝕊)
    q.lastRight ← 𝕊
    if q.neighborCount ≥ K then
        return
/* Then, search for neighbors in preceding slides that q has
   not checked yet */
p_slides ← GetPrecedingSlides(q, checked =
  false, reversedOrder = true)
for 𝕊 ∈ p_slides do
    FindNeighborInSlide(q, 𝕊)
    q.lastLeft ← 𝕊
    if q.neighborCount ≥ K then
        return
return
```

Algorithm 4: FindNeighborInSlide(q, $\mathbb{S}$)

```
/* Find core points in 𝕊 to get reduced the search space */
distance, cores ← FindCorePoint(q, allCores[𝕊.sIdx])
if distance ≤ R/2 then
    /* Using Theorem 3.4 */
    c ← cores[0]
    q.neighborCount ←
      q.neighborCount + size(c.E0[𝕊.sIdx])
    returnIfEnoughNB(q)
    FindNbInList(q, c.E1); returnIfEnoughNB(q)
    FindNbInList(q, c.E2); returnIfEnoughNB(q)
else if distance ≤ R then
    /* Using Theorem 3.5 */
    c ← cores[0]
    FindNbInList(q, c.E0); returnIfEnoughNB(q)
    FindNbInList(q, c.E1); returnIfEnoughNB(q)
    FindNbInList(q, c.E2); returnIfEnoughNB(q)
    FindNbInList(q, c.E3); returnIfEnoughNB(q)
else if distance ≤ 2R then
    /* Using Theorem 3.6 */
    for c ∈ cores do
        FindNbInList(q, c.E0)
        returnIfEnoughNB(q)
        FindNbInList(q, c.E1)
        returnIfEnoughNB(q)
return
```

2) There exists at least one core point in the range R/2 to R from q, 3) There exists at least one core point in the range R to 2R from q, and 4)There is no core point in the range 2R from q.

In each scenario, CPOD uses the corresponding reduced search space, according to Theorem 3.4, 3.5, and 3.6.

Instant neighbor confirmation: Specifically, in the first scenario, if there exists one core point c in the range R/2 from q, all the data points in $c.\mathbb{E}_0[s]$ are neighbors of q without computing any distance, according to Theorem 3.3. The neighbor count of q is updated by the size of $c.\mathbb{E}_0[s]$. If q still does not have enough neighbors, the neighbor search space for q is reduced to $c.\mathbb{E}_1[s] \cup c.\mathbb{E}_2[s]$.

Neighbor search space reductions: In the second scenario, there exists one core point c in range R/2 to R from q, the reduced neighbor search space is $c.\mathbb{E}_0[s] \cup c.\mathbb{E}_1[s] \cup c.\mathbb{E}_2[s] \cup c.\mathbb{E}_3[s]$, according to Theorem 3.5. In the third scenario, there exists a list C of core points in the range R to $2R$ from q, the reduced neighbor search space is $\bigcup_{c_i \in C}(c_i.\mathbb{E}_0[s] \cup c_i.\mathbb{E}_1[s])$, according to Theorem 3.6.

Instant no neighbor confirmation: In the fourth scenario, there is no core point in the range $2R$ from q, CPOD instantly confirms that q has no neighbors and stops neighbor search in slide s, according to Theorem 3.6.

During the neighbor search, if q has found enough neighbors, i.e., at least K neighbors when checking preceding slides or at least K succeeding neighbors when checking succeeding slides, the process is stopped, function returnIfEnoughNB(), Algorithm 4. The numbers of succeeding neighbors and preceding neighbors of q are updated with the count of found neighbors accordingly. Eventually, if q

does not have K neighbors, it is added to the outlier list. After the neighbor search step is complete, the outlier list is outputted.

3.4 Complexity Analysis

For each sliding window, we denote N_c as the number of core points, N_f as the number of outlier candidates, and N_r as the number of distance computations for a possible outlier. In the worst case, the time for indexing a new slide is $O(N_c S)$, the time for neighbor search is $N_f N_r$. Therefore, the total time complexity is $O(N_c S + N_f N_r)$. In most cases, $N_c << W$ as depicted in Table 1 and $N_f < W$. Because CPOD utilizes the *minimal probing principle* to stop neighbor search when finding enough neighbors, in most cases, N_r is small as reported in Table 7. Therefore, in most cases, the time complexity can be reduced to $O(S + WN_r)$.

Regarding the memory requirements, storing every window requires $O(W)$ space, and storing N_c core points requires $O(N_c T_{2r})$ space, where T_{2r} is the number of data points in the range $2R$ of core points. Therefore, the total space complexity is $O(W + N_c T_{2r})$. In most cases, $N_c << W$ and $T_{2r} < W$, and hence the total space complexity can be approximated as $O(W)$.

4 EXPERIMENTS

We compared our proposed algorithm CPOD with MCOD [15], M_MCOD [16], and NETS [18], which are current, state-of-the-art algorithms. For a fair evaluation, all the algorithms were implemented in Java. We obtained the NETS source codes, which were public in [18]. Our experiments were conducted on a Linux machine with a 3.47 GHz processor and 10 GB Java heap space.

Datasets. We used the six real-world data sets and a synthetic data set listed in Table 2, similar to [15, 16, 18]. The number of dimensions of the datasets ranges from 1 to 55. Datasets GAU, STK, and TAO are low dimensional (1 to 3), where GAU [15] is generated by a Gaussian mixture model with three distributions, STK [15] contains stock trading records, and TAO [1, 12] contains oceanographic data provided by the Tropical Atmosphere Ocean project. Datasets HPC and GAS are mid-dimensional (7 to 10), where HPC contains electric power consumption data, and GAS contains household gas sensor data. Datasets EM and FC are high-dimensional (16 to 55), where EM contains chemical sensor data, and FC contains forest cover type data. They are all available at UCI Machine Learning Repository [6].

Default Parameter Setting. To derive comparable outlier rates across datasets, which are approximately 1%, we set the default parameter values as in Table 2, similar to [1, 12, 15, 18]. Unless specified otherwise, all the parameters take on their default values in our experiments. For NETS, we set the number of sub-dimensions parameters, as suggested in [18], to be 3 and 4 for FC and EM, respectively.

Performance Measurement. We measured the CPU time of all the algorithms for processing each sliding window with ThreadMXBean in Java and used a separate thread to monitor the Java Virtual Machine memory. The CPU running time and peak memory measurements were averaged over all sliding windows.

4.1 Highlights of Results

We compared all the algorithms using all the datasets with the default value parameters. Figures 6 and 7 show the CPU running time and peak memory of the methods. Regarding CPU running time, CPOD ran much faster than the other algorithms with GAS, FC, TAO, EM, HPC datasets, and was comparable to NETS with GAU and STK datasets. As reported in Table 3, on average, CPOD was approximately 10, 19, and 73 times faster than M_MCOD, NETS, and MCOD, respectively. Especially with the EM dataset, CPOD required only 0.02 seconds to process one sliding window, which was 30, 77, and 201 times faster than M_MCOD, NETS, and MCOD, respectively, all required more than 0.6 seconds. The reason was that the *concentration ratio* of EM is low, i.e., many data points do not have K neighbors in the range R/2. There were few micro-clusters in MCOD and M_MCOD or inlier cells in NETS. With GAU and STK datasets, which are one-dimensional, CPOD had a comparable total CPU running time to NETS. This remarkable performance of CPOD demonstrated the merits of pruning techniques using core points and the minimal probing principle in reducing neighbor search space. Table 4 shows the average number of performed distance computations for processing each sliding window in all methods. They include the distance computations in indexing data points and neighbor searches. As depicted, on average, CPOD required 5.3, 8.6, and 34.1 times fewer distance computations than NETS, M_MCOD, and MCOD, respectively. Especially for the high-dimensional datasets, i.e., EM, FC, in which the distance computation is significantly expensive, CPOD required at least 8 times fewer distance computations than the other methods.

Table 2: Datasets and Default Parameter Values.

Dataset	Dim	Size	W	S	R	K
GAU	1	1.0M	100,000	5,000	0.028	50
STK	1	1.1M	100,000	5,000	0.45	50
TAO	3	0.6M	10,000	500	1.9	50
HPC	7	1.0M	100,000	5,000	6.5	50
GAS	10	0.9M	100,000	5,000	2.75	50
EM	16	1.0M	100,000	5,000	115	50
FC	55	0.6M	10,000	500	525	50

Table 3: The Speedups of CPOD compared to Other Methods: Others' CPU Running Time / CPOD's CPU Running Time.

Dataset	GAS	FC	TAO	EM	HPC	GAU	STK	Average
MCOD	98	25	18	201	24	80	67	73
M_MCOD	7	6	4	30	6	14	7	10
NETS	25	7	6	77	17	1	1	19

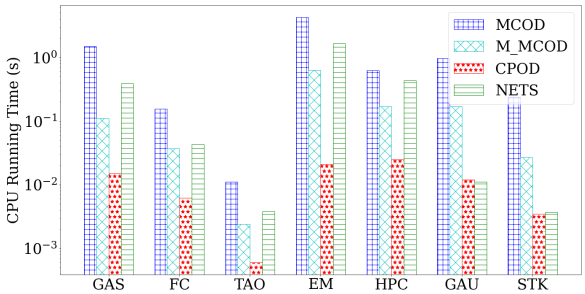


Figure 6: Overall CPU Running Time Comparison.

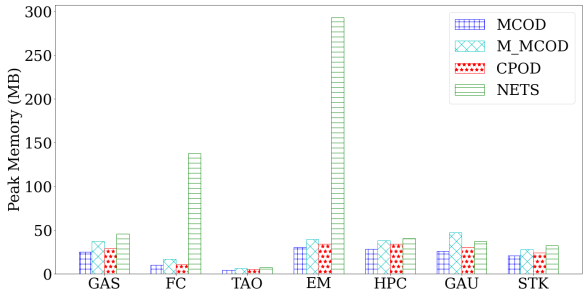


Figure 7: Overall Peak Memory Comparison.

Table 4: Average Number of Distance Computations Per Sliding Window (millions).

Dataset	EM	FC	HPC	GAS	TAO	STK	GAU	Average
MCOD	56.0	1.35	4.47	17.29	0.17	0.8	5.89	12.3
M_MCOD	12.0	0.6	2.2	2.6	0.03	0.8	3.56	3.1
CPOD	**0.27**	**0.07**	**0.72**	**0.43**	**0.02**	0.32	0.7	**0.36**
NETS	4.6	0.26	5.1	3.3	**0.02**	**0.02**	**0.2**	1.94

Table 5: Memory Comparison: Others' Peak Memory / CPOD's Peak Memory.

Dataset	GAS	FC	TAO	EM	HPC	GAU	STK	Average
MCOD	0.86	0.96	0.83	0.88	0.93	0.83	0.87	0.88
M_MCOD	1.27	1.55	1.15	1.14	1.11	1.56	1.17	1.28
NETS	1.34	12.71	1.29	8.59	1.18	1.22	1.35	3.95

Regarding memory requirements, as shown in Figure 7, CPOD required comparable memory to MCOD, M_MCOD, and much less than NETS. We report the ratio between the peak memory of the other algorithms and CPOD in Table 5. As reported, on average, CPOD required comparable memory to MCOD and M_MCOD and three times less memory than NETS. Especially with the FC dataset, the peak memory of CPOD was 12.7 times less than NETS and 1.5 times less than M_MCOD. CPOD required low memory because the number of core points is relatively small compared to the window size. CPOD required high memory for the EM dataset than the other datasets because there were more core points (see Table 1). Note that in return, for the EM dataset, CPOD achieved the highest speedup compared to the other methods. We observed a high peak memory in NETS with the EM and FC datasets. That is because the EM and FC datasets are high dimensional, and the grid-based index structure in NETS requires high memory.

4.2 Effects of Parameters on Performance

We varied the parameter values to verify the robustness of the performance of the algorithms. Due to the lack of space, we present the results for the selected datasets, i.e., FC, GAS, EM, and HPC. The results with the other datasets showed similar patterns. When varying one parameter, the other parameters took the default values.

4.2.1 Varying Window Size W. The window size determines the volume of the amount of workload on the algorithms. In this experiment, we varied the window size *W* from 1K to 20K with FC, from 10K to 200K with GAS, EM, and HPC.

Figure 8 shows the CPU running time of the algorithms. While the CPU time increased along with W for all the algorithms in most cases, the increase in CPU time for CPOD is mainly due to an increase in the number of data points for neighbor search. Notably, the increase in CPU time of CPOD was tiny compared to that of the other algorithms. The gap between CPOD and the others increased along with W. In all cases, CPOD was much faster than the other algorithms.

Figure 9 reports the peak memory when W increases. The peak memory increased because of the increase in the number of data points in a window. In most cases, the peak memory of CPOD was always lower than NETS and is comparable to MCOD and M_MCOD. With the EM dataset, the peak memory of CPOD increased more than with the other data sets as the number of core points grew but still less than 100 MB. Note that, in return, for this dataset, CPOD achieved the highest speedups compared to the other algorithms.

4.2.2 Varying Slide Size S. The slide size controls the speed of data streams and determines the number of data points arriving and expiring in each update. In this experiment, we varied the slide size S from 5% to 100% of the default value of W for each dataset.

Figure 10 reports the CPU running time of the methods. The CPU time of all the methods increased in most cases because when S is larger, more data points in a window are affected by expired or new neighbors. Therefore, the time for indexing data points and monitoring outlier statuses of data points increases. Again, CPOD achieved the fastest CPU running time in all cases. Especially for the EM dataset, the CPU running time of CPOD was always only 1% that of the other methods. For the FC, GAS, and HPC datasets, the gap between the CPU running time of CPOD and NETS was smaller along with S/W. That was because as S increases, the time for indexing a new slide in CPOD increased while NETS uses a grid-based index structure, which is less affected by the increase in the slide size. In return, CPOD was always faster in neighbor search and more efficient in quickly determining inliers. Therefore, even in the extreme case, S = W, CPOD was also faster than NETS with FC and EM, while comparable to NETS with GAS and HPC datasets.

Figure 11 shows the peak memory of all the methods. We observed that CPOD peak memory was comparable to MCOD and M_MCOD with FC and EM in all cases, and with GAS and HPC when $S/W \leq 10\%$. CPOD peak memory was always much smaller than that of NETS. For the GAS dataset, the peak memory of CPOD increased starting when $S/W = 20\%$. However, even in the extreme case, i.e., S = W, CPOD still required less than 50MB of memory.

4.2.3 Varying Radius Threshold R. The radius threshold R determines the area of neighborhood. When R increases, each data point can have more neighbors. In this experiment, we vary R from 10% to 1000% of the default value of R for each dataset.

Figure 12 shows the CPU running time of all the methods. When R increases, all the methods ran faster because all data points can have more neighbors and have smaller chances to become outliers. Also, more data points can be quickly confirmed as inliers because of more data points in micro-clusters or cells or $\mathbb{E}_0$ lists of core points. CPOD achieved the fastest CPU running time in most cases.

Figure 13 shows the peak memory of all the methods. We observed similar trends with the FC, GAS, and HPC datasets in which the peak memory of the methods decreased when R increased. In CPOD, the reason was that there are fewer core points.

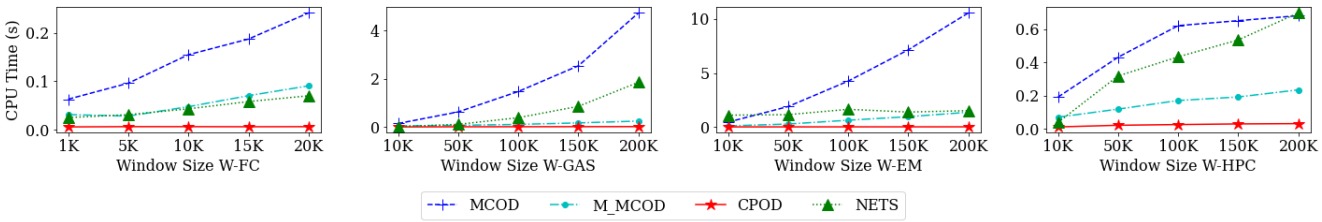


Figure 8: Varying Window Size - CPU Time Comparison.

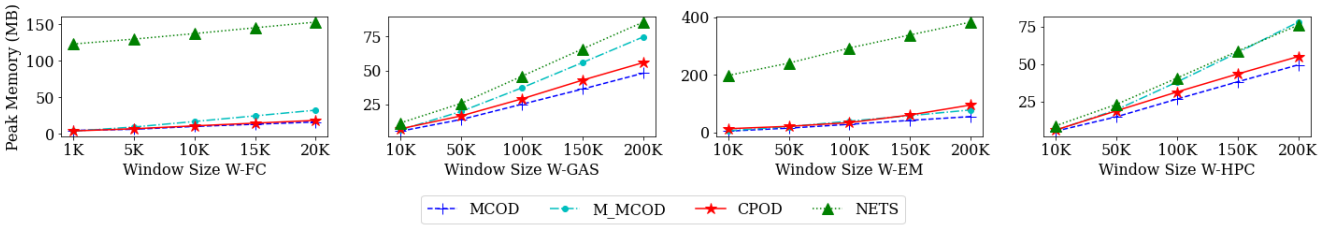


Figure 9: Varying Window Size - Peak Memory Comparison.

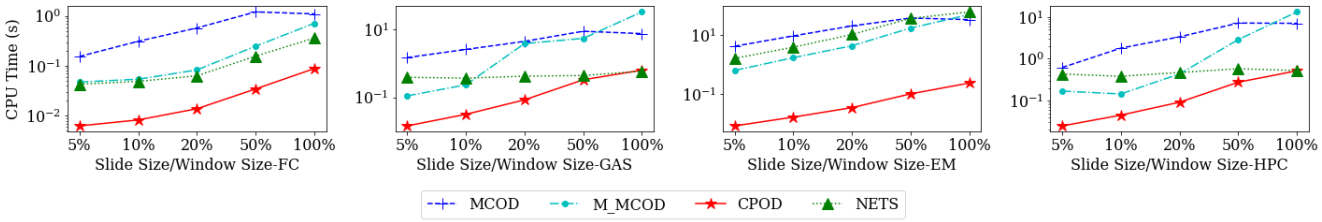


Figure 10: Varying Slide Size - CPU Time Comparison.

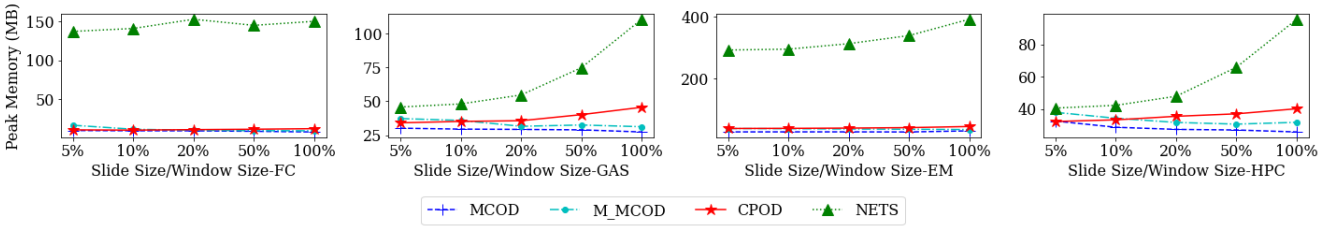


Figure 11: Varying Slide Size - Peak Memory Comparison.

Meanwhile, there are more data points in micro-clusters that do not need to store preceding neighbor list in MCOD and M_MCOD, and there are fewer cells in NETS. For the EM dataset, we observed that the peak memory of CPOD first increased when R was increased from 10% to 50%. This was because of the increase in the number of data points in the $\mathbb{E}$ lists of core points. When R was further increased, although the length of $\mathbb{E}$ lists also increased, the number of core points decreased, thus decreasing the peak memory. In most cases, the peak memory of CPOD was comparable to that of MCOD and M_MCOD, while much smaller than that of NETS.

4.2.4 Varying Neighbor Threshold K. The neighbor threshold K determines the number of neighbors required for a data point to be an inlier. Therefore, when K is increased, the number of outliers increases. We vary K from 10 to 100 for all the datasets. Figure 14 depicts the CPU running time of all the methods. In general, the CPU running time of the methods increased along with K. The reason is that there were fewer data points that were quickly confirmed as inliers. Here again, CPOD was much faster than the other methods. In all cases, CPOD was at least 2.5, 8.7, and 43.5 times faster than M_MCOD, NETS, and MCOD, respectively.

Figure 15 shows the peak memory of all the methods. The peak memory of CPOD was comparable to M_MCOD with the EM dataset, while much smaller than M_MCOD with the FC, GAS, and HPC datasets. In all cases, CPOD required less memory than NETS. Additionally, the peak memory of CPOD was quite stable because the increase in K does not affect the $\mathbb{E}$ lists of core points.

4.3 Online Updating Radius Threshold

In an online outlier detection system, users may want to adjust parameters, e.g., radius threshold R, to retrieve their expected outliers. Micro-clusters in MCOD and M_MCOD, cells in NETS, and core points in CPOD are built based on the radius threshold R.

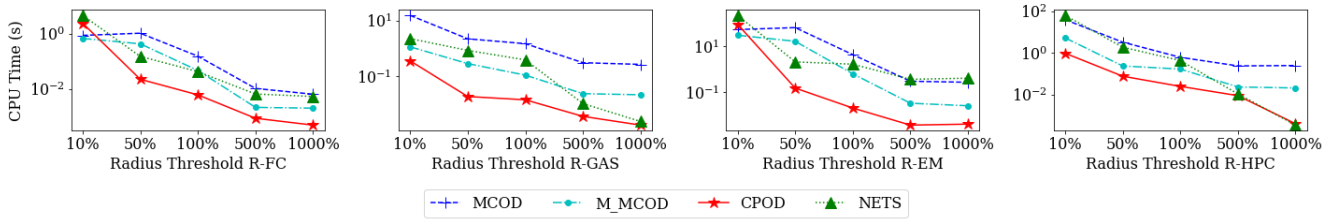


Figure 12: Varying Radius Threshold R - CPU Time Comparison.

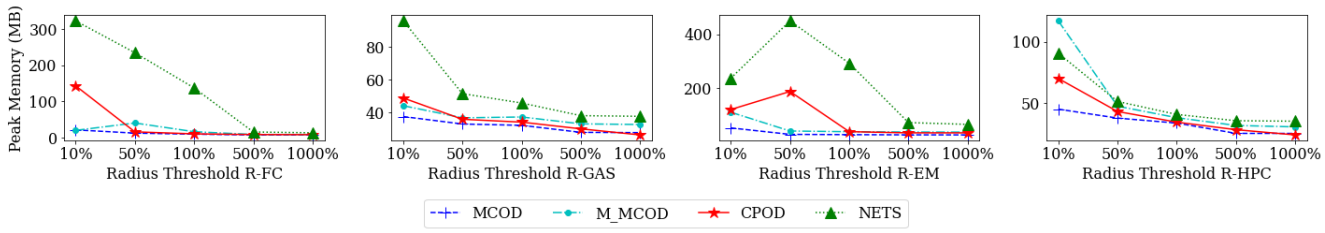


Figure 13: Varying Radius Threshold R - Peak Memory Comparison.

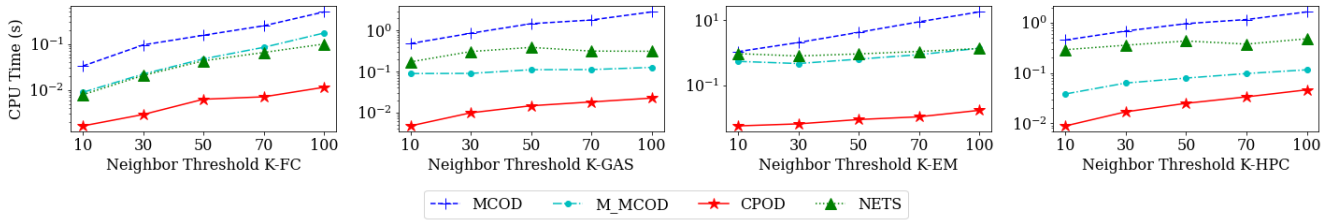


Figure 14: Varying Neighbor Threshold K - CPU Time Comparison.

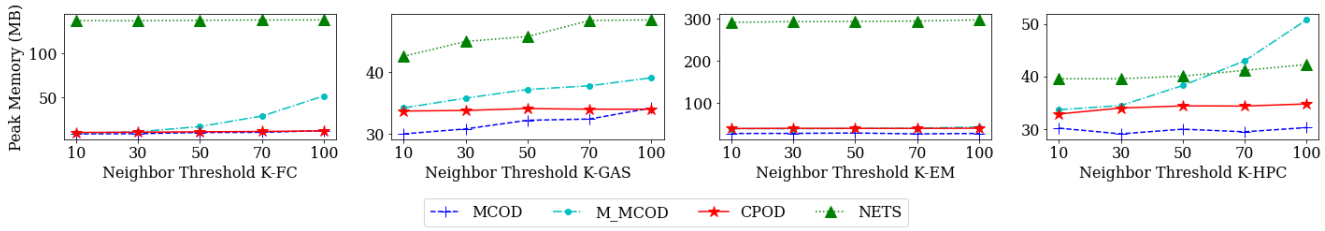


Figure 15: Varying Neighbor Threshold K - Peak Memory Comparison.

Consequently, the parameter R has a significant impact on all the algorithms' performance. When R is updated online, all the methods need to rebuild their corresponding index structures and re-find neighbors for the current data points. Table 6 shows the CPU time of all the methods for rebuilding the index structures and re-finding neighbors for the current data when R is updated online with the HPC dataset. We observed similar trends with the other datasets. As shown in this table, CPOD required a much shorter CPU time than the other algorithms. Especially in the extreme case, when R was updated to 10% of the default value, CPOD required only 21 seconds. In contrast, the other methods took approximately 5, 18, and 26 times longer for re-indexing data points and re-finding neighbors. This is because CPOD efficiently utilizes pruning techniques in selecting supporting core points and finding neighbors.

Table 6: CPU Time (seconds) for rebuilding index structure and re-finding neighbors when changing R - HPC Dataset.

R/Default R	10%	50%	100%	500%	1000%
CPOD	**21.14**	**1.33**	**0.58**	**0.24**	**0.08**
MCOD	545.06	52.31	10.95	1.48	0.71
M_MCOD	108.37	4.46	1.47	0.59	0.51
NETS	388.02	17.24	3.14	0.27	0.14

4.4 Running on Low-Resource Devices

We examined the performance of the algorithms when running on low-resource devices. Specifically, we used a Raspberry Pi emulator in which we varied the CPU clock speed from 700 MHz to 1500 MHz to simulate different real-world products. Raspberry Pi is commonly used to manage streaming data in IoT systems. Its memory was set to 1 GB. Figure 16 shows the running time of all the methods with the EM and HPC datasets. As shown in this figure, CPOD always

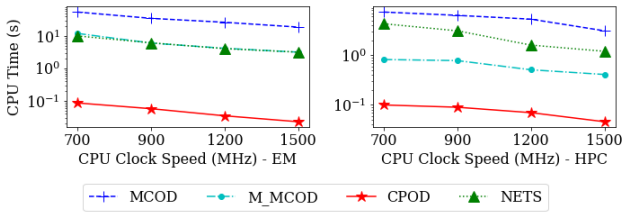


Figure 16: Varying CPU Clock Speed of Raspberry Emulator.

required less than 0.1 seconds, while the other methods required at least approximately 1 and 10 seconds for HPC and EM datasets.

4.5 The Breakdown of CPOD CPU Time

Figure 17 shows the breakdown of CPOD CPU time, which is separated into three main steps, i.e., expired slide processing, indexing new slide, finding neighbors for new and existing data. As shown in this figure, most of the CPU time was spent on indexing new slides and neighbor search. We observed the highest percentage of time for finding neighbors for new data in high dimensional data, e.g., FC, EM, HPC, and GAS. That was because of the large numbers of outlier candidates requiring neighbor searches, which are reported in Table 7. As depicted in this table, with the EM dataset, there were 3,360 outlier candidates on average, which were approximately 67% of a slide requiring neighbor searches. That is, most data points do not have K neighbors in the range R/2. That explains why the cell-based and micro-cluster based methods do not perform well with EM. The third row of Table 7 shows the average number of distance computations for each outlier candidate in the neighbor search. Interestingly, with EM and GAU, on average, each outlier candidate only needs less than K = 50 distance computations. That is because CPOD efficiently utilizes Theorems 3.3, 3.4, 3.5, and 3.6 for reducing neighbor search spaces.

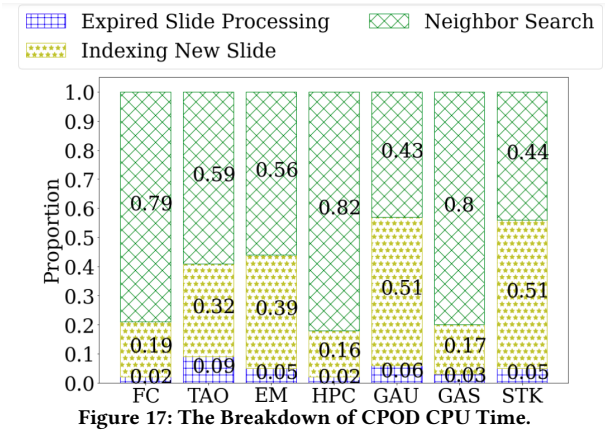


Figure 17: The Breakdown of CPOD CPU Time.

Table 7: Outlier Candidates Per Sliding Window.

Dataset	FC	TAO	EM	HPC	GAU	GAS	STK
No. Outlier Candidates	428	212	3360	1434	678	1919	711
Distance Computations Per Outlier Candidate	191	72	45	354	39	71	51

4.6 Core Point Selection Process

CPOD uses a heuristic, Algorithm 2, to select core points supporting each slide. It is equivalent to the min set cover problem [8], which finds the fewest data points such that the union of their neighbors covers the data set. This problem is NP-hard. We compared the number of core points selected by CPOD with the minimum core points selected by the integer programming approach, which requires finding all neighbors for each data point and searching for the best combination of core points. Although our heuristic does not return the minimal core point set, it is much faster than the integer programming approach because it neither finds all neighbors for each data point nor conducts a core point combination search. We report the average ratio between the number of CPOD's selected core points and the optimal core points for each slide, as in Table 8. As reported in this table, the number of CPOD's selected core points was less than twice the optimal core points for all the datasets.

Table 8: Average Ratio Between the Number of CPOD Core Points and Minimal Core Point Selection.

Dataset	FC	TAO	GAU	GAS	EM	HPC	STK
CPOD/Optimal	1.9	1.61	1.39	1.98	1.99	1.54	1.32

5 DISCUSSIONS

With the sliding window setting, depending on the application, users might be interested in the aggregated outlier status of every data point from its outlier status in sliding windows. For example, an aggregated outlier can be defined as a data point that is an outlier in at least $n \geq 1$ sliding windows. To achieve that, with CPOD, for each data point, we can add a variable that monitors the number of sliding windows in which it is an outlier. This study does not consider the aggregation process after detecting outliers in every sliding window to keep the problem more generic.

6 CONCLUSIONS

In this paper, we proposed CPOD, a real-time distance-based outlier detection algorithm in data streams. With CPOD, we proposed a core point data structure, along with multi-distance pruning techniques, to both quickly identify inliers and reduce neighbor search space. We showed that CPOD performs efficiently with various types of datasets and outperforms the state-of-the-art algorithms in CPU running time while consuming low memory. An algorithm optimizing the memory usage of core points, e.g., minimizing the size of supporting core point set, or compressing the $\mathbb{E}$ lists of a core point, can be explored in the future work.

ACKNOWLEDGMENTS

This work has been supported in part by NSF grants IIS-1910950 and CNS-2027794, the USC Integrated Media Systems Center, and unrestricted cash gifts from Microsoft and Google. The opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the sponsors.

REFERENCES

[1] Fabrizio Angiulli and Fabio Fassetti. 2007. Detecting distance-based outliers in streams of data. In *Proceedings of the sixteenth ACM conference on Conference on information and knowledge management*. ACM, 811–820.

[2] Dominik Breitenbacher, Ivan Homoliak, Yan Lin Aung, Nils Ole Tippenhauer, and Yuval Elovici. 2019. HADES-IoT: A Practical Host-Based Anomaly Detection System for IoT Devices. In *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security*. 479–484.

[3] Lei Cao, Di Yang, Qingyang Wang, Yanwei Yu, Jiayuan Wang, and E.A. Rundensteiner. 2014. Scalable distance-based outlier detection over high-volume data streams. In *Data Engineering (ICDE), 2014 IEEE 30th International Conference on Data Engineering*. 76–87.

[4] Lei Cao, Di Yang, Qingyang Wang, Yanwei Yu, Jiayuan Wang, and Elke A Rundensteiner. 2014. Scalable distance-based outlier detection over high-volume data streams. In *2014 IEEE 30th International Conference on Data Engineering*. IEEE, 76–87.

[5] Paolo Ciaccia, Marco Patella, and Pavel Zezula. 1997. M-tree: An E cient Access Method for Similarity Search in Metric Spaces. In *Proceedings of the 23rd VLDB conference, Athens, Greece*. Citeseer, 426–435.

[6] Dheeru Dua and Casey Graff. 2017. UCI Machine Learning Repository. http://archive.ics.uci.edu/ml

[7] Nicholas Duffield, Patrick Haffner, Balachander Krishnamurthy, and Haakon Andreas Ringberg. 2016. Systems and methods for rule-based anomaly detection on IP network flow. US Patent 9,258,217.

[8] Uriel Feige. 1998. A threshold of ln n for approximating set cover. *Journal of the ACM (JACM)* 45, 4 (1998), 634–652.

[9] Junhao Gan and Yufei Tao. 2017. Dynamic density based clustering. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 1493–1507.

[10] Edwin M. Knorr and Raymond T. Ng. 1998. Algorithms for Mining Distance-Based Outliers in Large Datasets. In *Proceedings of the 24rd International Conference on Very Large Data Bases (VLDB '98)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 392–403.

[11] M. Kontaki, A. Gounaris, A.N. Papadopoulos, K. Tsichlas, and Y. Manolopoulos. 2011. Continuous monitoring of distance-based outliers over data streams. In *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*. 135–146. https://doi.org/10.1109/ICDE.2011.5767923

[12] Maria Kontaki, Anastasios Gounaris, Apostolos N Papadopoulos, Kostas Tsichlas, and Yannis Manolopoulos. 2011. Continuous monitoring of distance-based outliers over data streams. In *2011 IEEE 27th International Conference on Data Engineering*. IEEE, 135–146.

[13] Yuhang Lin, Byung Suk Lee, and Daniel Lustgarten. 2018. Continuous detection of abnormal heartbeats from ECG using online outlier detection. In *Annual International Symposium on Information Management and Big Data*. Springer, 349–366.

[14] Marina Thottan and Chuanyi Ji. 2003. Anomaly detection in IP networks. *IEEE Transactions on signal processing* 51, 8 (2003), 2191–2204.

[15] Luan Tran, Liyue Fan, and Cyrus Shahabi. 2016. Distance-based outlier detection in data streams. *Proceedings of the VLDB Endowment* 9, 12 (2016), 1089–1100.

[16] Luan Tran, Liyue Fan, and Cyrus Shahabi. 2019. Fast Distance-based Outlier Detection in Data Streams based on Micro-clusters. In *Proceedings of the Tenth International Symposium on Information and Communication Technology*. 162–169.

[17] Di Yang, Elke A. Rundensteiner, and Matthew O. Ward. 2009. Neighbor-based Pattern Detection for Windows over Streaming Data. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology* (Saint Petersburg, Russia) *(EDBT '09)*. ACM, New York, NY, USA, 529–540.

[18] Susik Yoon, Jae-Gil Lee, and Byung Suk Lee. 2019. NETS: extremely fast outlier detection from a data stream via set-based processing. *Proceedings of the VLDB Endowment* 12, 11 (2019), 1303–1315.