Streamline Density Peak Clustering for Practical Adoptions

Shuai Yang
North Carolina State University
Raleigh, North Carolina
syang16@ncsu.edu

Xipeng Shen
North Carolina State University
Raleigh, North Carolina
xshen5@ncsu.edu

Min Chi
North Carolina State University
Raleigh, North Carolina
mchi@ncsu.edu

ABSTRACT

Since Density Peak Clustering (DPC) algorithm was proposed in 2014, it has drawn lots of interest in various domains. As a clustering method, DPC features superior generality, robustness, flexibility and simplicity. There are however two main roadblocks for its practical adoptions, both centered around the selection of *cutoff distance*, the single critical hyperparameter of DPC. This work proposes an improved algorithm named *Streamlined Density Peak Clustering (SDPC)*. SDPC speeds up DPC executions on a sequence of cutoff distances by 2.2-8.8X while at the same time reducing memory usage by a magnitude. As an algorithm preserving the original semantic of DPC, SDPC offers an efficient and scalable drop-in replacement of DPC for data clustering.

CCS CONCEPTS

• **Information systems → Clustering**; • **Theory of computation → Unsupervised learning and clustering**.

KEYWORDS

density clustering, algorithm optimization, hyperparameter tuning

ACM Reference Format:
Shuai Yang, Xipeng Shen, and Min Chi. 2019. Streamline Density Peak Clustering for Practical Adoptions. In *The 28th ACM International Conference on Information and Knowledge Management (CIKM '19), November 3–7, 2019, Beijing, China.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3357384.3358053

1 INTRODUCTION

Density Peak Clustering (DPC) is a general density-based clustering algorithm proposed in 2014 [23]. Compared to traditional clustering methods, DPC excels at multiple aspects: (1) Generality: Unlike some other clustering methods (e.g., K-Means), DPC applies to arbitrary data distributions and cluster shapes; (2) Robustness: it carries a built-in mechanism that detects and removes outliers; (3) Flexibility: no prior knowledge on the number of clusters is needed.

For these attractive properties,since its proposal, DPC has drawn interest in various domains, from facial images clustering [23], to image segmentation [5], text clustering [18] and so on. It has shown better clustering quality over other clustering algorithms [24], such as K-means [20], DBSCAN [11], Affinity Propagation [12].

Two main roadblocks however have been limiting practical adoptions of DPC; both center around *cutoff distance*, the single but critical hyperparameter in DPC. Cutoff distance, denoted as d_c, defines the range of neighborhood when the density of a point is computed. Data points within the d_c radius of the concerned data point are called its *neighbors*. The number of neighbors is the **density** of that data point. The value of d_c thus determines density computation—the key measure DPC relies on—and hence the quality of the final clusters. The influence is dramatic. As Figure 1 shows, a tiny change in d_c can cause large changes in the clustering results, which echoes the prior reported observations on the unstableness of clustering quality due to d_c values [25].

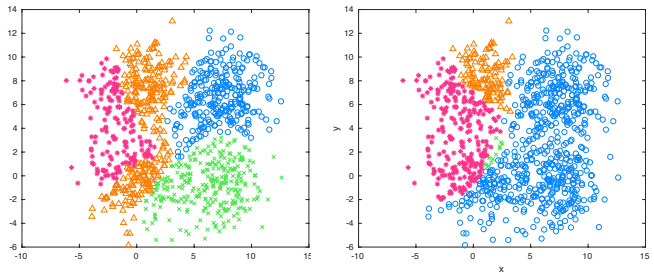
Roadblock-I: Temporal Delays. For the sensitivity of DPC on d_c, when applying DPC, users need to carefully select the appropriate values for d_c. As there are no well established methods for computing the best d_c value, iterative methods are typically used, in which, the practitioner runs DPC many times with a different d_c value used each time, and adopts the best among all the trials (as per Adjusted Rand Index(ARI), Davies-Bouldin index or other cluster quality measures) as the final clustering results. The delay in producing the clustering results is the first roadblock for practical DPC adoptions.

Roadblock-II: Spatial scalability. There can be many possible values of d_c; arbitrarily selecting the values to try may waste much time. The original DPC design [23], based on empirical observations, suggests to focus d_c choices to a range such that the average number of neighbors of a data point is 1–2% of the total number of data points. To determine that range, the design requires the use of $O(N^2)$ memory space to store all the pairwise distances (N is the number of data points), and keep them in memory throughout the whole process, because these distances could be requested by the subsequent stages of DPC. The large space overhead makes DPC hard to apply to large datasets, forming the other main roadblock for practical adoptions of DPC.

There have been some prior efforts trying to alleviate the issues. They fall into two main categories. (1) The first category skips the selection of d_c: Wang and others [25] propose to directly estimate the density via nonparametric multivariate kernel density estimation. Mehmood and others [21] further refine the kernel density estimation with heat diffusion method. Other work uses KNN instead of d_c to identify neighbors [10]. All of these approaches, however, change the definition of density in the original DPC and hence fall short in maintaining the semantic of DPC as well as the clustering results. (2) The second category directly estimates the optimal d_c: Field Theory in physics is introduced to model the density as potential in data field [26]. It estimates the optimal d_c through minimizing potential entropy of datasets. Gao and others [13] take

CIKM '19, November 3–7, 2019, Beijing, China

ACM ISBN 978-1-4503-6976-3/19/11…$15.00
https://doi.org/10.1145/3357384.3358053



(a) $d_c = 5.19$ (b) $d_c = 5.49$

Minor changes in d_c affect clustering quality (Adjusted Rand Index (ARI)) significantly

d_c	0.038	0.047	0.054	0.095	0.099	0.103	0.131	0.134	0.137
ARI	0.744	**0.578**	0.737	0.749	**0.621**	0.757	0.757	**0.579**	0.598

Figure 1: As the only hyperparameter, d_c strongly influences the clustering results. (Graphs: clustering results on dataset Squre4 [1] as d_c takes different values; Table: ARI changes as d_c value changes on dataset 2d-3c-no123[1])

into account the degree of dispersion of a dataset when determining the optimal d_c and derive a formula for d_c based on the standard deviations of each dimension. Zhang and others [30] propose to use distributed systems for DPC and estimate d_c via Reservoir Sampling. Although these efforts are valuable attempts, none of them can offer the guarantee that the resulting quality of the density and hence clustering results are not compromised compared to those by the original DPC.

The goal of this work is to provide a *drop-in replacement* of the original DPC with significantly enhanced speed and scalability, especially when the best cutoff distance needs to be empirically determined. We achieve the goal by distinctively focusing on computation reuse and stream-style data processing. The drop-in replacement is named *Streamlined DPC* (SDPC). SDPC reduces the memory cost of DPC by a magnitude through an incremental algorithm named *double-buffer incremental introselect*, and meanwhile, speeds up DPC executions on a sequence of d_c values by 2.2X–8.8X by harnessing information latent in previous iterations.

Inspired by studies for other algorithms [28, 29], unlike prior methods, our approach keeps DPC semantics unchanged — that is, keeping its clustering results the same as the original algorithm gives. This property guarantees no quality loss in whatever problem settings, and hence preserves all the appealing properties of the original DPC and all the confidence that users have built through the many applications of DPC. Although designed primarily to replace the original DPC, the techniques in SDPC can also benefit other variants of DPC by, for instance, speeding up the search for d_c in a range around the value estimated with previously proposed methods. To our knowledge, SDPC is the first drop-in replacement of the original DPC that preserves its semantic with substantially improved efficiency and scalability. By removing the two major roadblocks, it helps better materialize the full potential of DPC in practical usage.

2 BACKGROUND AND NOTATIONS

For reference, we put all notations used in this paper into Table 1.

Table 1: Notations

Notations	Definitions
d_{c_i}	The ith smallest cut-off distance
x_i	The ith data point
ρ_i	Density of x_i
δ_i	Distance from x_i to its nearest data point of higher ρ
$d_{i,j}$	Distance between x_i and x_j
N	Total number of data points
n	Total number of pairwise distances between data points
q_i	Index of data point with the ith highest ρ
q_i^c	Index of data point with the ith currently highest ρ
n_i	Index of x_i's nearest data point of higher ρ
C_t	The tth cluster
ρ_t^b	Density threshold for the border region of C_t
cl_i	Cluster label of x_i
d_i^s	The ith smallest distance among all pairwise distances between data points
p_i	Index of d_{c_i} in the sorted distance list
nb_i	Neighbor list of x_i
nbS_i	List of neighbor sizes of x_i under different d_c
$\bar{\rho}_{i,j}$	Average density of ρ_i and ρ_j

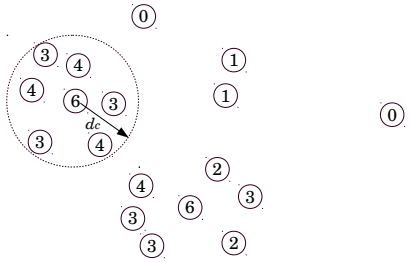


Figure 2: The density of a data point is the number of data points that are closer than d_c to it. The number on each data point shows the density of it.

DPC is based on the observation that every cluster center is surrounded by neighbors of lower density, while it has a relatively long distance to any other data points of higher density. There are two key concepts in DPC: (1) ρ: density and (2) δ: distance to the nearest data point of higher density.

2.1 Density (ρ)

Definition 2.1. The Density ρ_i is the number of data points that are closer than d_c to the data point x_i. As shown in Figure 2:

$$\rho_i = \sum_{j \in I_s \setminus \{i\}} \chi(d_{i,j} - d_c) \tag{1}$$

where if $x < 0$, $\chi(x) = 1$, otherwise, $\chi(x) = 0$. d_c is the cutoff distance and $d_{i,j}$ is the distance between data points x_i and x_j, I_S is the index set of all data points in the dataset.

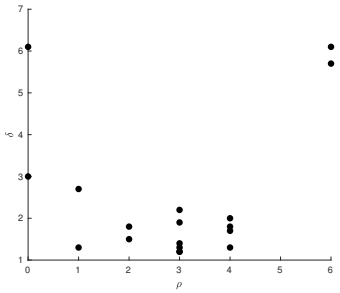


Figure 3: Decision graph for the example in Figure 2. Two cluster centers at upper-right corner are easily identified from other data points at the bottom of the graph.

2.2 Distance to the Nearest Data Point of Higher Density (δ)

Definition 2.2. δ_i is the distance from data point x_i to the nearest data point with a higher density:

$$\delta_{q_i} = \begin{cases} \min\limits_{j<i}\{d_{q_i,q_j}\}, & i \geqslant 2 \\ \max\limits_{j\geqslant 2}\{\delta_{q_j}\}, & i = 1 \end{cases} \tag{2}$$

where q_i is the index of the data point with the ith highest ρ. If two data points have the same ρ, they can be ordered by their original order in dataset. For the highest density point x_{q_1}, δ_{q_1} is defined as the maximum of all other δ's. That is to guarantee that x_{q_1} is one of the cluster centers, which we will elaborate later.

Since cluster centers are supposed to have both large ρ and δ, as the example shows in Figure 2, if we draw a graph using ρ as x-axis and δ as y-axis, the cluster centers are outliers located at the upper right of the graph. This graph is called *decision graph* which serves to identify cluster centers. Figure 3 is the decision graph for the example shown in Figure 2. It's easy to identify two cluster centers at upper right which correspond to the two data points with density 6 in Figure 2. Decision graph also helps unravel noise. Data points at the upper left corner are likely to be noise due to their low density and large distance to other high ρ data points. The cluster centers are hence picked as the K data points that have the largest products ($\rho \cdot \delta$). K can be determined automatically or specified by users[23].

2.3 Cluster Halo and Cluster Core

Besides the clustering results, DPC further divides all the data points into two categories: cluster core (robust assignment) and cluster halo (noise). For each cluster, DPC defines a border region, which includes data points in that cluster that are meanwhile less than d_c away from data points in other clusters. To tell cores from halos, a *density threshold* is computed for each cluster as follows:

$$\rho_t^b = \max_{x_i \in C_t}\{\max_{x_j \notin C_t \wedge d_{x_i,x_j} < d_c}\{\frac{\rho_i + \rho_j}{2}\}\} \tag{3}$$

Data points with density larger than the threshold form the cluster core, and other data points in the cluster form its halo.

3 IMPLEMENTATION OF DPC

Rodriguez and Laio have provided an implementation of DPC in the supplementary materials of their original DPC paper [23]. This section briefly describes this implementation as well as its speed and space bottlenecks. At a high level, the algorithm consists of three steps as outlined by the left side chart in Figure 4.

Stage I: Getting the value of d_c. The first stage is to determine the value of d_c. As a rule of thumb, d_c is chosen so that the average number of neighbors (i.e. data points with distance less than d_c to the concerned data point) is around 1-2% of the total number of data points in the dataset. To get such a d_c, DPC sorts all the pairwise distances of all data points in ascending order and picks the value at the position of $p \cdot n$, where p is specified by user and usually $\in [1\%, 2\%]$.

This implementation is straightforward, but it is both time and memory consuming. In this paper, we use N to represent the total number of data points in a dataset and n for the total number of pairwise distances. Hence, $n = (N(N-1))/2 \in O(N^2)$. Therefore, sorting all the distance takes $O(N^2 logN^2)$ time and the algorithm requires $O(N^2)$ memory space. When N is large, this stage becomes the bottleneck in both time and space. The limitation is severe. A machine with 40GB memory cannot run DPC to cluster a dataset with only 58000 data points (the Shuttle dataset [8]), as just storing the 1.68 billion pairwise distances would take 25GB memory. It seriously hampers the scalability of DPC.

Stage II: Computing ρ and δ of each data point. With the d_c value picked in Stage I, for each data point x_i, Stage II of DPC iterates all other data points and counts the number of data points with distance less than d_c to x_i, which takes $O(N^2)$ time. To compute δ, DPC sorts all the data points in terms of their density in descending order. Then, the process of computing δ is to locate the nearest data point in front of the concerned data point in that sorted list. The sorting takes $O(NlogN)$ and getting all δ in the sorted list costs $O(N^2)$. After that, computing $\rho \cdot \delta$ and pinpointing cluster centers with the k largest values take linear time.

Stage III: Cluster assignment and identifying cluster halo. Every data point is assigned to the same cluster as its nearest data point of higher density is, which takes $O(N)$ time. Let n_i ($i > 1$) be the index of x_i's nearest data point with a higher density. Thus x_{n_i} is x_i's nearest data point of higher density. The process follows descending order of density. Because only in that order, we can guarantee the cluster label of x_{n_i} has been already determined before x_i.

DPC adopts a straightforward approach to identify the cluster halo. It iterates all pairs of data points. As long as two data points come from different clusters and their distance is less than d_c, the average density of two data points are used to update the density threshold for border regions of both clusters. When the density threshold is determined, all data points with density less than the threshold of the cluster they belong to are marked as cluster halo. The time complexity of this stage is also $O(N^2)$.

To summarize, based on the preceding analysis of DPC, sorting all distances to get the value of d_c is the most computationally expensive. Since all distances remain unchanged during the tuning process, it's possible to reuse sorted distances so that sorting

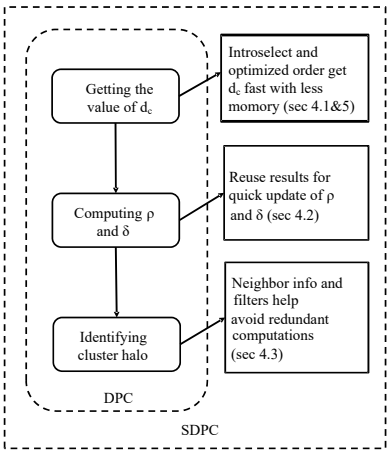


Figure 4: Main stages in DPC and the corresponding optimizations in SDPC

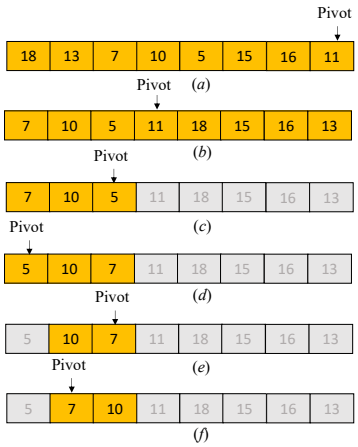


Figure 5: Example of Introselect algorithm searching the 2nd smallest element in an unsorted list. (a) To be concise, in this toy example we use the last element as the pivot; (b) Swap elements larger than the pivot to its right side. The pivot now is in its correct position, which is the 4th smallest element; (c) Since elements less than pivot are all in front (i.e., left side) of it, the algorithm works recursively on that side and considers no elements behind the pivot any more; (d) The pivot is the smallest element and the 2nd smallest element must be on its right side; (e) Recursively works on the new region; (f) Finally, the pivot is at the 2nd position and 7 is the 2nd smallest element that we are looking for.

only needs to be done once. It is however not the case for other parts of DPC: if d_c is changed, ρ, δ and cluster halo all need to be recomputed.

As there are no well established methods for directly calculating the best d_c value, practical uses of DPC run DPC many times with a different d_c value used each time, and select the best clustering results to use (assessed by ARI, DBI or other cluster quality measures). With m choices of d_c, the overhead for each of these parts would be as much as $O(m \cdot N^2)$. Next, we show how SDPC significantly reduces the time cost of the original DPC, and Section 5 explains how we remove the roadblock for spatial scalability of DPC.

4 REUSE-CENTERED OPTIMIZATIONS

The speed enhancement by SDPC centers around computation reuse and avoidance of unnecessary computations in the tuning process. It consists of six optimizations, which streamline all the three stages of DPC, as outlined on the right side of Figure 4. We next explain the design of each of the optimizations in SDPC.

4.1 Calculate d_c via Introselect and ordering

In the previous section, we have noticed the process of getting the value of d_c entails large overhead. It involves sorting all distances. Our target is to find the distance value at a specific position (e.g. $2\% \cdot n$) of the sorted distance list. This problem is equivalent to "find the kth smallest element in an unsorted list". It is not necessary to sort the entire dataset to get that value. There are some well-known algorithms that can solve this problem in linear time, such as Quickselect [15], Median of Medians [4] and Introselect [22].

Introselect is a hybrid of Quickselect and Median of Medians, combining their advantages. Quickselect has fast average performance, so Introselect starts with Quickselect: Every time it chooses a pivot and partitions elements into two parts, depending on whether their values are less than or greater than the pivot, then it works recursively on the side that has the element it looks for, as shown in Figure 5. But when it does not make sufficient progress, Introselect switches to Median of Medians which divides elements into sublists and gets the median of each sublist. Then it uses the median of these medians as the pivot to partition elements. Median of Medians guarantees linear worst-case performance.

SDPC adopts the Introselect to attain the value of d_c in linear time. If d_c corresponds to the $(p \cdot n)$th smallest value of all pairwise distances, after applying Introselect, d_c is at the position $p \cdot n$ in the list of all pairwise distances. Besides that, distances are partitioned into two parts by Introselect. Distances less than d_c are in front of d_c while distances larger than d_c are behind it, but distances in a partition are not sorted.

In the process of tuning, suppose that the range of interest for d_c is 1-2% of the total number of data points. If the step size is 0.1%, the choices of d_c are those where the average number of neighbors is 1%, 1.1%, 1.2%,...,2% of all data points. In another word, the values of d_c are $d^s_{\lfloor 1\% \cdot n \rfloor}$, $d^s_{\lfloor 1.1\% \cdot n \rfloor}$, $d^s_{\lfloor 1.2\% \cdot n \rfloor}$,...,$d^s_{\lfloor 2\% \cdot n \rfloor}$, respectively, where d^s_i is the ith element in the sorted distance list. The challenge is how to determine these values efficiently. One straightforward way is to sort all distances and then get the value of each d_c. However, sorting all the distances easily outweighs the benefits it may bring. As aforementioned, Introselect is applied to get the value of d_c fast. SDPC further optimizes the order in exploring d_c choices to reduce computations during repeated runs of Introselect for each d_c, as described in the following optimization.

Optimization Strategy 1: Assume there are m choices for d_c: $d_{c_1} < d_{c_2} < d_{c_3} <, ..., < d_{c_m}$, the execution order of Introselect takes the descending order of d_c.

Let's assume the corresponding position of d_{c_i} in the sorted distance list is p_i; based on the definition of d_{c_i}, it is clear that

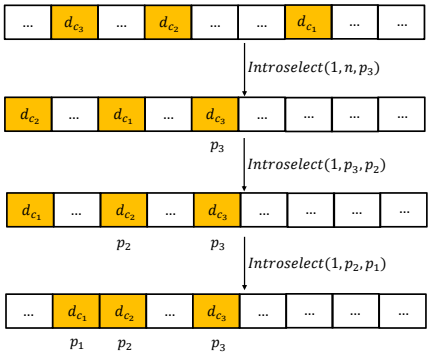


Figure 6: The process of performing Introselect in descending order of d_c. p_i is the corresponding position of d_{c_i} in sorted distance list. In this example, Introselect firstly places d_{c_3} into the correct place; at this point, d_{c_2} and d_{c_1} reside in front of d_{c_3} but are not in the correct places yet. The algorithm then works on d_{c_2} to put it into the correct place, and then fixes the location of d_{c_1}. Among the three parameters in Introselect(start, end, k), the first two specify the range it works on and the third parameter indicates that we are looking for the kth smallest element.

$p_1 < p_2 < p_3 <, ..., < p_m$. Figure 6 illustrates the descending order of d_c, in which, the Introselect works. The benefit comes from the property of the Introselect that it partitions the data into two parts based on d_c. All the elements less than d_c are in front of d_c while others greater than d_c are behind d_c in that distance list. Therefore, when SDPC finds d_{c_m} through Introselect, we know $d_{c_1}, d_{c_2}, ..., d_{c_{m-1}}$ are all located in front of d_{c_m} whose position is p_m. So for $d_{c_{m-1}}$, instead of searching the space of all the distances, Introselect only needs to focus on the subinterval $[1, p_m]$. Likewise, when $d_{c_{m-1}}$ is found, $d_{c_{m-2}}$ must be in the interval $[1, p_{m-1}]$. The total number of data points Introselect examines is $n + p_m + p_{m-1} + ... + p_2$. In contrast, if we explore all the d_c values in ascending order, every time the new d_c would be at the right side of the previous one. The subinterval to work on for $d_{c_{i+1}}$ is $[d_{c_i}, n]$. Therefore, the total number of data points to examine would be $n + (n - p_1) + (n - p_2) + ... + (n - p_{m-1})$. The theoretical speedup equals to :

$$\omega = \frac{mn - \Sigma_{i=1}^{m-1} p_i}{n + \Sigma_{i=2}^{m} p_i} \tag{4}$$

since usually $1\% \cdot n \leqslant p_i \leqslant 2\% \cdot n$ and $m >> 1$, utilizing Optimization Strategy 1 helps SDPC attain the values of all d_c fast. After getting all d_c values, we can then test each d_c in ascending order.

4.2 Update ρ and δ via computation reuse

This section describes optimizations involved in updating densities (ρ) and distances to the nearest neighbor with a higher density (δ). When d_c changes, ρ changes accordingly. The sorted list of data points (ordered on ρ) from the previous trial becomes out of order. Restoring the order is needed for updating δ. Therefore, this stage can be subdivided into two parts: (1) updating ρ; (2) restoring the order of ρ and updating δ. Optimization Strategy 2 is for the first part while Optimization Strategies 3 to 5 are all for the second part, among which, Optimization Strategy 3 proposes a new way to update δ during the process of restoring the order of ρ, which could largely reduce the number of recomputations of δ; Optimization Strategies 4 and 5 accelerate remaining recomputations.

1) Updating ρ: When a new d_c is applied, ρ of each data point could be changed accordingly. SDPC leverages the results that have already been computed in the trials of some earlier choices of d_c to avoid unnecessary computations. For two consecutive trials of SDPC on d_{c_k} and $d_{c_{k+1}}$, where $d_{c_k} < d_{c_{k+1}}$, densities of data point x_i under two trials are ρ_i^k and ρ_i^{k+1}, respectively. We can regard ρ_i^{k+1} as being composed of two parts of data points: data points whose distances from x_i are less than d_{c_k}, and those between d_{c_k} and $d_{c_{k+1}}$. Note that data points in the first part have already been counted in ρ_i^k, so only those with distances between d_{c_k} and $d_{c_{k+1}}$ are needed to be checked.

The crux of the problem becomes how to get the number of data points in the second part efficiently. Thanks to the use of Introselect, SDPC has divided all distances into a number of ordered partitions. Distances less than d_{c_k} and $d_{c_{k+1}}$ are all in front of p_k and p_{k+1}, respectively, where p_k and p_{k+1} are corresponding positions of d_{c_k} and $d_{c_{k+1}}$ in the sorted distance list. The distances belong to $[d_{c_k}, d_{c_{k+1}})$ are all located in the interval between p_k and p_{k+1} The increment of ρ relates with only the distances belonging to that interval. SDPC capitalizes on this observation to get speedups.

Optimization Strategy 2: After the exploration moves from d_{c_k} to $d_{c_{k+1}}$, when updating ρ, it needs to consider only the distances in the interval between p_k and p_{k+1} in the distance list. The operations are to increase the corresponding two end points' ρ value by one for each distance falling into that interval.

This strategy reduces the time complexity of updating ρ substantially. The number of distances between p_k and p_{k+1} relies on the step size of d_c. The setting of the step size in our experiments is $\frac{4}{100 \cdot T} \cdot n$, where $T = 8, 16, 32, 64$. So the number of distances to go through is $\frac{4}{100 \cdot T} \cdot n$ while the original one checks all n distances.

2) Restoring the order of ρ and updating δ: SDPC maintains a sorted list of data points based on ρ so that all data points with higher density than data point x_i's are in front of x_i in that list. It becomes easy to compute δ_i, since we need to go through only the data points in front of x_i and choose the shortest distance. However after updating ρ, data points in that list could become out of order. It is necessary to restore the order. Re-sorting ρ is a simple way to do that, but not the most efficient way: Although the change of ρ could mess up the order of a few data points, the list is still nearly sorted. So rather than resorting the whole list, we only need to adjust data points that are out of order, which can bring the time complexity down from $O(NlogN)$ to $O(N)$.

In addition, SDPC takes an optimized scheme to update δ. Rather than waiting to the end of restoring the order of data points, it updates δ while restoring the order of ρ, which allows for it to effectively leverage some latent information about δ contained in the order restoring process of ρ. The rationale is that δ of x_i hinges on the list of data points of higher ρ than itself. Although, for a new d_c, the relative positions of some data points could change, the relative positions of most data points do not. SDPC manages to reuse known information and only compute the changed part.

Specifically, SDPC goes from the beginning of the data points list (ordered on ρ in the previous trial of d_c). When it encounters an out-of-order data point, it swaps the involved data points to fix the order and at the same time updates δ. For instance, suppose that the data point it encounters is x_j, the data point before it is x_i, and $\rho_i < \rho_j$. SDPC swaps x_i and x_j to fix the order problem. After swapping, it uses Optimization Strategy 3 to update δ_i and δ_j. SDPC then continues to work on the rest of the list in this manner. Figure 7 illustrates how Optimization Strategy 3 works.

Optimization Strategy 3: Let x_i and x_j be two adjacent data points in the sorted list of data points with ρ mis-ordered ($\rho_i < \rho_j$). The updated δ_i and n_i (the index of x_i's nearest data point of higher ρ) are as follows:

$$\delta_i = \min(\delta_i, d_{i,j}) \tag{5}$$

$$n_i = \begin{cases} n_i, & d_{i,j} >= \delta_i \\ j, & otherwise \end{cases} \tag{6}$$

and the updated δ_j and n_j are as follows:

$$\delta_j = \begin{cases} \delta_j, & n_j \neq i \\ \min\limits_{t<s}\{d_{q_t^c, q_s^c}\}, & n_j = i \text{ and } s \geqslant 2 \end{cases} \tag{7}$$

$$n_j = \begin{cases} n_j, & n_j \neq i \\ \arg\min\limits_{q_t^c, t<s}\{d_{q_t^c, q_s^c}\}, & n_j = i \text{ and } s \geqslant 2 \end{cases} \tag{8}$$

where s is the determined position of x_j which we will explain later, q_i^c is the index of data point with the currently ith highest ρ. We say "currently" because the update process is still in progress. Note that $s = 1$ is a special case, we do not update it right now. Because there are two possibilities: (1) the final position of x_j is 1, δ_j is then assigned with the maximum value of all δ after all δ are updated; (2) the final position is not 1, it will be swapped with other data points and then Formula 5 and 6 are applied to update x_j.

This optimization strategy is easy to understand. For x_i, after the swapping, it now has one more data point of higher density. Therefore, comparing the current minimum with the distance to the new point of higher density leads to the new minimum. As for x_j, after swapping, it has one fewer point of higher density. There could be two different cases. (1) x_i is not x_{n_j}. In this case, we do not need to update δ_j and n_j. Because we know x_{n_j} is still in front of it. (2) x_i is x_{n_j}. In this case, because x_i is no longer a data point of higher density of x_j, recomputing the δ_j and n_j is unavoidable. But the recomputation is not executed immediately, as the position of x_j has not be determined and it could be further swapped with elements in front. Once the position of x_j is determined, we recompute δ_j and n_j.

Through Optimization Strategy 3, most of recomputations in DPC can be avoided, which brings significant speedups. But as we have mentioned, there are still remaining cases in which recomputations have to be done. For these cases, SDPC uses an early-stop scheme to accelerate recomputations, as explained next.

For every data point x_i, SDPC maintains a list of all its neighbors nb_i. Suppose that we have m choices of d_c, and $d_{c_1} < d_{c_2} < \ldots < d_{c_m}$. Firstly, SDPC executes with d_{c_1}, nb_i adds all neighbors within d_{c_1} and forms the first partition. When applying d_{c_2}, neighbors with distances between $[d_{c_1}, d_{c_2})$ are added to nb_i and form the second partition. The same goes with the other d_c choices. Hence, nb_i is divided into multiple partitions. These partitions are in order, even though the data points inside each partition are not. Based on this property, we propose the fourth Optimization Strategy.

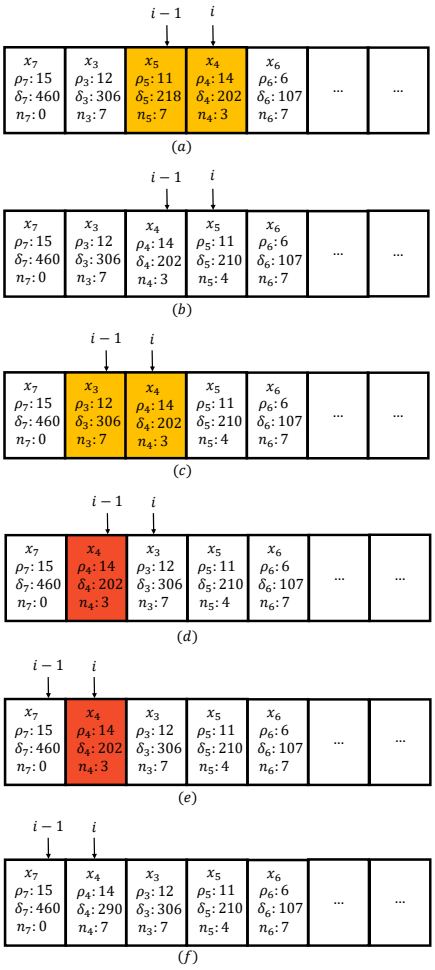


Figure 7: Example of the process of updating δ. (a) Go from the beginning to find the first out-of-order element, which is x_4 in this example; (b) Swap x_4 with x_5; assume $d_{4,5} = 210 < \delta_5$, δ_5 and n_5 are hence updated. On the other side, as $n_4 \neq 5$, no need to change n_4 or δ_4; (c) Continue to check x_4 with the element in front of it, since $\rho_3 < \rho_4$, they are out of order; (d) Swap x_4 with x_3; assume $d_{3,4} = 365 > \delta_3$, hence no update to δ_3 or n_3. However, as $n_4 = 3$, n_4 and δ_4 need to be recomputed. But the recomputation is not executed immediately, since the position of x_4 has not been determined. It's just marked (in red) to indicate that recomputation is needed; (e) Continue to check x_4 with x_7. They are in correct order; the position of x_4 is now determined. (f) Recomputation of n_4 and δ_4 is executed.

Optimization Strategy 4: For data point x_i, if one data point of a higher density is found in the jth partition of nb_i, then SDPC skips the subsequent partitions $k(k > j)$ in updating δ_i and n_i.

Proof. Let P_j and P_k be the set of indices of data points within the jth and kth partition of nb_i, respectively, where $k > j$.
$\forall y \in P_j, d_{x_i,x_y} \leqslant \min_{z \in P_k} d_{x_i,x_z}$.
If $\exists y' \in \{y | \rho_y > \rho_i, y \in P_j\}$,
$x_{y'}$ is a data point of higher density of x_i
$\because d_{x_i,x_{y'}} \leqslant \min_{z \in P_k} d_{x_i,x_z}$
$\therefore n_i$ must not be in P_k □

Note that SDPC can not stop immediately when it finds a data point of a higher density in partition j of nb_i, because data points in a partition are not ordered; some other data points in the partition may also meet the condition and are even closer to x_i.

Furthermore, it is not necessary to start searching from the very beginning of nb_i. The trial of a previous d_c value already provides some information that can help skip partitions that are impossible to contain new x_{n_i}, as the following optimization strategy exploits.

Optimization Strategy 5: For data point x_i, assume that x_{n_i}, in the previous trial of d_c, is in the jth partition of nb_i; in updating δ_i and n_i, SDPC skips data points in partition h, for all $h < j$, as they cannot be the new x_{n_i}. (Note: when updating δ_i and n_i, data points after x_i have not been processed. This new x_{n_i} may not be the final one, it could be further updated later)

Proof. Let P_j is the set of indices of data points within the jth partition of nb_i.
$\because$ In the previous trial of d_c, $n_i \in P_j$,
Assume $y \in P_h (h < j)$, we have $\rho_y < \rho_i$, otherwise $n_i = y \in P_h$
$\therefore x_y$ must be positioned after x_i in the list ordered by ρ.
Recall the Optimization Startegy 3 and Figure 7, the updating process starts from the beginning of that list.
When updating x_i, x_y is still untouched. New x_{n_i} comes from the data points currently in front of x_i
$\therefore x_y$ wil not be the new x_{n_i} □

Optimization Strategies 4 & 5 together can help accelerate the computations in updating ρ and δ. It's worth noting that if we keep nb_i sorted by their distance to x_i, it will be easier to identify the new nearest data point of higher ρ. We have explored this approach, but the overhead introduced by maintaining a sorted neighbor list outweighs the benefit.

4.3 Identify halos via computation skipping

Cluster core is the part with high confidence while cluster halo is the border region which DPC is less certain about. The separation provides users with more information about the clustering results. The key step in identifying cluster halo is to determine the density threshold which is defined in Formula 3. This step could be time-consuming since it involves enumerating all pairs of data points. SDPC manages to accelerate this step which is the performance bottleneck, thereby speeds up the whole process.

According to the definition of density threshold, the computation only happens between data points and their neighbors (data points within d_c distance to the concerned point). Since SDPC maintains a list of neighbors for each data point, it only needs to go through that list, reducing the number of computations from $\frac{N(N-1)}{2}$ to cN, where c is the average number of neighbors, typically falling in the range of $[1\% \cdot N, 2\% \cdot N]$.

If x_i and x_j are neighbors, their average density $\bar{\rho}_{i,j}$ is used to update the density thresholds of clusters that each of them belongs to. $\bar{\rho}_{i,j}$ will be computed when checking x_i with nb_i as well as checking x_j with nb_j. To avoid recomputing $\bar{\rho}_{i,j}$, SDPC employs a rule that $\bar{\rho}_{i,j}$ is computed only once when this current data point's density is less than or equal to the other one's. Based on that, SDPC employs optimization strategy 6 to further reduce computations.

Optimization Strategy 6: During updating density thresholds, for a data point x_i, if $\delta_i > d_c$, then SDPC skips x_i and continues to the next data point.

The rationale is straightforward. Condition $\delta_i > d_c$ means the distance to the nearest data point with higher density is larger than d_c. That is to say x_i has no neighbor whose density is higher than its own. Since computations of updating density thresholds happen only between a data point and its neighbors of a higher density, SDPC can hence skip x_i, avoiding examining its neighbors.

5 ENABLE SPATIAL SCALABILITY

Recall that the original DPC uses $O(N^2)$ to store all pairwise distances and sorts them in its calculation of a d_c value, which forms the roadblock for DPC scalability. In our optimized algorithm described in Section 4.1, by using Introselect, it circumvents sorting all pairwise distances. But it still needs to store all pairwise distances stored in memory. This section describes our solution to the problem. Our solution is a *double-buffer incremental introselect*. It employs two buffers. Suppose that we are looking for the kth smallest distance. The first buffer is of size k, and the second buffer is of size $\alpha k, (\alpha > 0)$. In our experiments, it is set to 1. The first buffer keeps the current k smallest distances; the second buffer stores new data.

Data elements are read into the buffers one after one. Every time when the buffer is full, Introselect is applied to the concatenation of the two buffers to find the current kth smallest distance. As a consequence of Introselect, all distances less than the current kth distance must reside in front of it, and hence the first buffer now has the current k smallest distances even though they are not in order. Distances in the second buffer are discarded, since none of them are possible to be part of the global k smallest distances. New distances are loaded into the second buffer to overwrite those discarded distances. When the buffer gets full, we run Introselect again. This procedure repeats until all distances have been processed. The distances in the the first buffer are now the global k smallest distances.

The memory consumption reduces from $\frac{N(N-1)}{2}$ units (one distance per unit) to $(1 + \alpha)k$. k is the position of the largest d_c in the sorted distance list, which equals to $\frac{2\%N(N-1)}{2}$ as the original DPC work [23] suggests. When $\alpha = 1$ (as used in our experiments), the memory consumption equals $2\%N(N-1)$. While it invokes Introselect $O(\frac{n}{k})$ times, every time the Introselect works on a much smaller buffer of size $(1 + \alpha)k$. The total time still remains linear.

One extra optimization is that each time when a new distance is loaded into the second buffer, SDPC compares the new distance d_{new} with the current kth smallest distance. If d_{new} is larger, it is

Table 2: Datasets Used in Experiments

Dataset	Dataset Properties			
	num. of points	num. of distances	num. of clusters	type
Elly_2d10c13a	2796	3907410	10	Artificial
Fig2_PanelB	4000	7998000	5	Artificial
Wine Quality	4898	11992753	7	RealWorld
Anuran Calls	7195	25880415	10	RealWorld
HTRU2	17898	160160253	2	RealWorld
Shuttle	58000	1681971000	7	RealWorld
Skin Seg.	245057	3.0×10^{10}	2	RealWorld

immediately discarded. Because it is impossible to be one of the global k smallest distances. As the algorithm proceeds, the current kth smallest distance becomes closer and closer to the global kth smallest distance, the optimization becomes increasingly effective.

Note that in the other stages of SDPC, likewise, rather than keeping all pairwise distances in memory like what DPC does, SDPC keeps only distances less than d_c in memory. They cover most of distances required in the following stages. The only exception is that when SDPC updates δ for a data point x, its nearest data point with a higher density could be more than d_c away from x. In that case, SDPC computes their distance on-the-fly. which are uncommon in our experiments, on average only 7% distances fall into this category.

6 EXPERIMENTS

We evaluate the efficiency of SDPC on a variety of benchmark datasets as shown in Table 2. Dataset Elly_2d10c13a [1], is an artificial dataset with 10 highly overlapping clusters. Fig2_PanelB is a dataset used in the DPC paper with artificially added noise from an uniform distribution. The following 5 real world datasets are collected from UCI Machine Learning Repository [8]. Wine Quality [7] aims to utilize physicochemical properties (e.g. citric acid and pH) to model the quality of wine. Anuran Calls [6] is a dataset of audio records for different frog calls. HTRU2 [19] collects emission signals of pulsars which are a type of star, of considerable scientific interest. Shuttle [8] dataset is used for extracting rules determining whether autolanding of a spacecraft is preferable. Skin Segmentation [2] contains RGB values of skin textures from images of people from diverse groups. The datasets are chosen to be diverse in both properties and the intended usage.

Clustering results. As SDPC does not change the semantic of DPC, it always produces the exactly same clustering results as DPC does, confirmed by our experimental results. Our experiments hence focus on the speedups it attains and the memory it saves. The machine we use for experiments is PowerEdge R620 equipped with 2 Xeon E5-2670 CPUs, 128GB memory, Ubuntu 16.04.

As mentioned in Section 1, small changes in d_c can result in large changes in clustering quality. To achieve a good coverage, we set the range of d_c such that the average number of neighbors is between 1% and 5% of all data points. To make it straightforward, we adopt equal step size in our experiments. To be specific: d_{c_i} is the distance at position p_i, where $p_i = 1 + 4\frac{i}{T}, (i = 1, 2, .., T)$ and T is the number of d_c choices varying among 8, 16, 32 and 64.

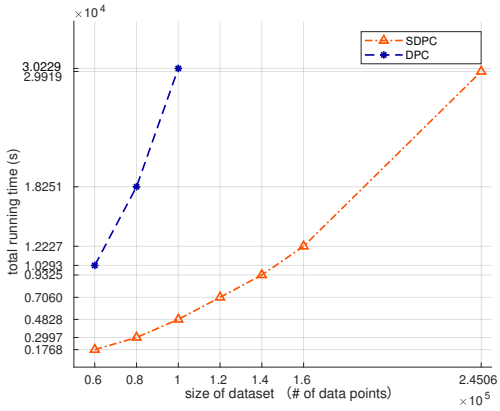


Figure 8: Running time of DPC and SDPC on Skin Segmentation dataset and its subsets with 64 d_c. Subsets are composed of 60000, 80000, 100000, 120000, 140000 and 160000 data points

In the original implementation of DPC, it sorts all the distances at the very beginning. Since distances are not changed during the process of tuning, there is no need to re-sort them every time. We hence revised the original DPC implementation so that sorted distances are reused. We take this revised implementation as our baseline which is faster than directly running original DPC. We next report the comparisons between the baseline and SDPC, in terms of both end-to-end time, the time of each of the major stages as well as the memory cost.

End-to-End Time. Table 3 reports the running times on each dataset in various settings. The numbers in the parentheses show the speedups (X) over the DPC. Note that the speedups are calculated with original full precision time so it may be a little different from the results calculated from the running time in Table 3 (only one decimal digital kept). It's clear to see the pattern that more d_c corresponds to larger speedups. This is intuitive: More times of executions leave larger space for optimization strategies to take effect. As shown in the 4th column, the best end-to-end performance we have observed is on the Shuttle dataset, which attains 8.8 times speedups. For the Skin Segmentation dataset, the original DPC algorithm runs out of the memory, so it is not listed in Table 3. By exploring the executions on different subsets of the Skin Segmentation dataset, we probe the upper bound of dataset size the original DPC can handle. Subsets of different numbers of data points are collected whose size are: 60000, 80000, 100000, 120000,140000 and 160000. Figure 8 shows that the largest number of data points DPC can deal with on the machine is 100000 while SDPC can process the whole dataset (245057 data points) with even less running time ($3.02 \times 10^7 s$ vs $2.99 \times 10^7 s$).

In our experiments, the largest number of d_c choices is 64, but in practice, with more d_c being included, even larger speedups will be achieved. Small datasets tend to show smaller speedups, as there are less room for improvement. Overall, the results show consistent superior performance of SDPC, across datasets and across the numbers of d_c choices. Next, we provide a detailed analysis of the results of each major stage and discuss some attained insights.

Table 3: Running Time Comparison of Different Algorithms

(The numbers in parentheses in column 4,6,8 and 10 are speedups (X) over DPC)

Dataset	d_c #	Running time (s)							
		Total		Getting d_c		Computing ρ and δ		Identifying cluster halo	
		DPC	SDPC	DPC	SDPC	DPC	SDPC	DPC	SDPC
Elly_2d10c13a	8	6.3	2.8 (2.3)	4.3	2.5 (1.7)	1.5	0.23 (6.6)	0.42	0.032(13.0)
	16	8.0	2.8 (2.9)	4.1	2.4 (1.7)	3.0	0.30 (10.2)	0.84	0.068 (12.4)
	32	12.1	3.0 (4.0)	4.1	2.4 (1.7)	6.2	0.41 (15.2)	1.7	0.14 (12.3)
	64	19.9	3.4 (5.8)	4.2	2.5 (1.7)	12.1	0.55 (21.8)	3.4	0.28 (12.0)
Fig2_PanelB	8	12.8	5.5 (2.3)	9.0	5.1 (1.8)	2.9	0.35 (8.2)	0.86	0.044(**19.5**)
	16	16.2	5.7 (2.9)	8.6	5.1 (1.7)	5.8	4.7 (12.3)	1.7	0.092 (19.0)
	32	23.8	6.1 (3.9)	8.6	5.3 (1.6)	11.5	0.58 (19.7)	3.5	0.19 (18.6)
	64	41.0	6.7 (6.1)	9.5	5.3 (1.8)	24.1	0.87 (27.5)	7.2	0.38 (18.9)
Wine Quality	8	31.5	8.6 (3.7)	24.5	7.2 (3.4)	5.2	1.2 (4.4)	1.6	0.11 (15.5)
	16	37.4	9.4 (4.0)	24.4	7.5 (3.2)	10.0	1.6 (6.4)	2.9	0.23 (12.8)
	32	52.2	10.2 (5.1)	25.5	7.6 (3.4)	20.5	2.1 (9.8)	5.9	0.46 (12.7)
	64	77.2	11.6 (6.7)	24.7	7.7 (3.2)	40.4	2.8 (14.3)	11.7	0.93 (12.7)
Anuran Calls	8	49.3	22.5 (2.2)	36.4	16.3 (2.2)	10.1	5.9 (1.7)	2.6	0.25 (10.4)
	16	59.1	23.7 (2.5)	31.9	16.4 (1.9)	21.3	6.7 (3.2)	5.5	0.53 (10.5)
	32	83.4	25.2 (3.3)	32.4	16.6 (1.9)	40.2	7.4 (5.4)	10.4	1.1 (9.5)
	64	132.7	27.6 (4.8)	30.3	17.0 (1.8)	80.8	8.2 (9.8)	20.9	2.2 (9.7)
HTRU2	8	305.2	108.7 (2.8)	214.2	98.4 (2.2)	70.6	9.1 (7.7)	19.1	1.1 (17.4)
	16	381.6	113.7 (3.4)	206.6	99.9 (2.1)	135.4	11.3 (12.0)	38.1	2.3 (16.4)
	32	573.4	119.8 (4.8)	210.4	100.5 (2.1)	280.9	14.1 (19.9)	80.4	4.8 (16.9)
	64	948.8	132.2 (7.2)	209.3	103.9 (2.0)	574.1	18.0 (31.9)	162.8	9.5 (17.1)
Shuttle	8	3597.3	1186.9 (3.0)	2445.4	1071.7 (2.3)	913.1	95.1 (9.6)	224.9	19.8 (11.4)
	16	5262.6	1203.7 (4.4)	2946.0	1056.2 (2.8)	1828.8	104.7 (17.5)	451.7	42.1 (10.7)
	32	7439.7	1278.1 (5.8)	3045.7	1075.4 (2.8)	3487.4	116.1 (30.1)	892.4	85.3 (10.5)
	64	12309.5	1404.5 (**8.8**)	3185.6	1094.7 (2.9)	7309.2	134.9 (**54.2**)	1795.7	172.3 (10.4)

Speedups on getting d_c. As Table 3 shows, more d_cs show slightly more speedups in this stage for a dataset, but the increment is not a lot. The reason is that this part includes the time spends on I/O operations. Even though SDPC does not need to keep all distances in the memory, it still needs to read all distances from the hard drive. The time spent on the I/O operations is a major part of the time. It is the same for the same dataset.

Avoided computations in updating ρ and δ. This stage updates ρ, the density, and δ, the distance to the nearest data point of higher ρ. As Table 3 shows, in this stage DPC achieves significant speedups, especially when the number of d_c gets larger. SDPC has proposed four Optimization Strategies for this process. Besides the running time speedups, Table 4 presents the number of recomputations of δ executed in DPC and SDPC with 64 d_c choices. Recomputing δ is the most computationally intensive in this stage, because for each data point, it needs to iterate all data points with higher density. Optimization Strategy 3 manages to avoid this recomputation. The second and third columns in Table 4 show the total number of recomputations needed by DPC and SDPC, respectively. The number of recomputations in SDPC is on average 5.7% of that in DPC; 94.3% recomputations are avoided. For the remaining recomputations, Optimization Strategies 4 & 5 take effects to accelerate recomputations. Column four shows the number of recomputations done with Optimization Strategies 4 & 5. When Optimization 4 & 5 are not suitable, the original approach is used for updating δ; the numbers of such recomputations are listed in column five. They are very small portions of the numbers in column 1. The synergy of all of these Optimization Strategies brings SDPC as much as 54.2X speedups (on Shuttle dataset) for this stage.

Table 4: Recomputations Comparison for Updating δ.

(The percentages in parentheses are the numbers of recomputations compared with that in DPC.)

Dataset	Number of Recomputations of δ			
	DPC	SDPC		
	Total	Total	Optimized	Remaining
Elly_2d10c13a	176148	19843 (11.3%)	18950	893 (0.5%)
Fig2_PanelB	252000	21707 (8.6%)	17823	3884 (1.5%)
Wine Quality	308574	14123 (4.6%)	13705	418 (0.1%)
Anuran Calls	453285	8624 (1.9%)	8161	463 (0.1%)
HTRU2	1127574	44298 (3.9%)	42284	2014 (0.2%)
Shuttle	3654000	86227 (2.4%)	84211	2016 (0.06%)

Speedups in identifying cluster halo. Speedups in this stage are only marginally affected by the change of the number of d_c. Our exploration shows that the speedups strongly correlate with the degree of overlap between clusters in the dataset (-0.83 Pearson correlation coefficient), which is intuitive given that this step mainly considers the overlapping areas.

Memory cost. Recall that our algorithm reduces the memory for storing distances from $\frac{N(N-1)}{2}$ units (one distance per unit) to $2\%N(N-1)$, independent of d_c. We hence report the memory cost in one d_c setting (d_c=64). Table 5 reports the measured maximum resident set size in memory. As the dataset increases, the savings increases, until it reaches about 90%. The savings corresponding to subsets of Skin Segmentation dataset with 60000, 80000 and 100000 data points are 88.6%, 89.5% and 89.1%. This trend meets our

Table 5: Memory Cost Comparison

(The percentages in parentheses are the memory cost of SDPC compared with that in DPC.)

Dataset	Memory Usage (MB)	
	DPC	SDPC
Elly_2d10c13a	96.4	17.1 (17.7%)
Fig2_PanelB	189.5	28.6 (15.1%)
Wine Quality	314.4	41.2 (13.1%)
Anuran Calls	654.4	82.5 (12.6%)
HTRU2	4500.7	460.7 (10.2%)
Shuttle	42217.7	4514.5 (10.7%)

expectation. In theory, the improved Introselet Algorithm reduces stored distances from $\frac{N(N-1)}{2}$ to $2\%N(N-1)$ which is 4% of original method. Also, for every data point, SDPC maintains a list of all its neighbors, it takes another 4%. There are some other data and intermediate memory usage which weight more for small dataset.

7 RELATED WORK

Section 1 has mentioned a set of work closely related with the selection of cutoff distances in DPC. Many efforts are made to improve DPC from other aspects. Xie and others [27] design a new cluster assignment strategy to improve the robustness of DPC. An adaptive cluster center selection approach is proposed for automatical center selection [3]. Li and others [17] view DPC from the perspective of tree structure and improve the decision graph by constructing a Δ-tree.

Accelerating clustering algorithms have been a popular research topic. Strategies proposed for other clustering algorithms have inspired our work. Bound-based strategies are designed to speedup the GMM training [29]. Ideas of computation-reuse has been applied to K-Means [9, 14]. Delaunay diagram is used to accelerate 2-D spatial clustering [16].

8 CONCLUSION

This work proposes a set of techniques to harness computation reuse opportunities and to reduce memory usage of DPC. The new algorithm, SDPC, gives significant speedups and large memory savings across all datasets of various sizes and dimensions, without compromising the clustering quality. By removing the two major roadblocks for practical adoptions, SDPC offers an efficient and scalable drop-in replacement of DPC. (The code and datasets are shared at https://github.com/PICTureRG/SDPC.)

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation (NSF) under Grant No. CCF-1525609 and CCF-1703487. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF.

REFERENCES

[1] Tomas Barton. 2015. clustering-benchmark. https://github.com/deric/clustering-benchmark.

[2] Rajen B Bhatt, Gaurav Sharma, Abhinav Dhall, and Santanu Chaudhury. 2009. Efficient skin region segmentation using low complexity fuzzy decision tree model. In *2009 Annual IEEE India Conference*. IEEE, 1–4.

[3] Rongfang Bie, Rashid Mehmood, Shanshan Ruan, Yunchuan Sun, and Hussain Dawood. 2016. Adaptive fuzzy clustering by fast search and find of density peaks. *Personal and Ubiquitous Computing* 20, 5 (2016), 785–793.

[4] Manuel Blum, Robert W. Floyd, Vaughan R. Pratt, Ronald L. Rivest, and Robert Endre Tarjan. 1973. Time bounds for selection. *J. Comput. Syst. Sci.* 7, 4 (1973), 448–461.

[5] Zhensong Chen, Zhiquan Qi, Fan Meng, Limeng Cui, and Yong Shi. 2015. Image segmentation via improving clustering algorithms with density and distance. *Procedia Computer Science* 55 (2015), 1015–1022.

[6] Juan Gabriel Colonna, Marco Cristo, Mario Salvatierra Júnior, and Eduardo Freire Nakamura. 2015. An incremental technique for real-time bioacoustic signal segmentation. *Expert Systems with Applications* 42, 21 (2015), 7367–7374.

[7] Paulo Cortez, António Cerdeira, Fernando Almeida, Telmo Matos, and José Reis. 2009. Modeling wine preferences by data mining from physicochemical properties. *Decision Support Systems* 47, 4 (2009), 547–553.

[8] Dua Dheeru and Efi Karra Taniskidou. 2017. UCI Machine Learning Repository. http://archive.ics.uci.edu/ml

[9] Yufei Ding, Yue Zhao, Xipeng Shen, Madanlal Musuvathi, and Todd Mytkowicz. 2015. Yinyang k-means: A drop-in replacement of the classic k-means with consistent speedup. In *International Conference on Machine Learning*. 579–587.

[10] Mingjing Du, Shifei Ding, and Hongjie Jia. 2016. Study on density peaks clustering based on k-nearest neighbors and principal component analysis. *Knowledge-Based Systems* 99 (2016), 135–145.

[11] Martin Ester, Hans-Peter Kriegel, Jörg Sander, Xiaowei Xu, et al. 1996. A density-based algorithm for discovering clusters in large spatial databases with noise.. In *Kdd*, Vol. 96. 226–231.

[12] Brendan J Frey and Delbert Dueck. 2007. Clustering by passing messages between data points. *science* 315, 5814 (2007), 972–976.

[13] Jing Gao, Liang Zhao, Zhikui Chen, Peng Li, Han Xu, and Yueming Hu. 2016. Icfs: An improved fast search and find of density peaks clustering algorithm. In *Dependable, Autonomic and Secure Computing, 14th Intl Conf on Pervasive Intelligence and Computing, 2nd Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress (DASC/PiCom/DataCom/CyberSciTech), 2016 IEEE 14th Intl C*. IEEE, 537–543.

[14] Hui Guan, Yufei Ding, Xipeng Shen, and Hamid Krim. 2018. Reuse-Centric K-Means Configuration. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. IEEE, 1224–1227.

[15] Charles AR Hoare. 1961. Algorithm 65: find. *Commun. ACM* 4, 7 (1961), 321–322.

[16] Ickjai Lee and Vladimir Estivill-Castro. 2006. Fast cluster polygonization and its applications in data-rich environments. *Geoinformatica* 10, 4 (2006), 399–422.

[17] Zejian Li and Yongchuan Tang. 2018. Comparative density peaks clustering. *Expert Systems with Applications* 95 (2018), 236–247.

[18] Peiyu Liu, Yingying Liu, Xiuyan Hou, Qingqing Li, and Zhenfang Zhu. 2015. A text clustering algorithm based on find of density peaks. In *Information Technology in Medicine and Education (ITME), 2015 7th International Conference on*. IEEE, 348–352.

[19] Robert J Lyon, BW Stappers, S Cooper, JM Brooke, and JD Knowles. 2016. Fifty years of pulsar candidate selection: from simple filters to a new principled real-time classification approach. *Monthly Notices of the Royal Astronomical Society* 459, 1 (2016), 1104–1123.

[20] James MacQueen et al. 1967. Some methods for classification and analysis of multivariate observations. In *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*, Vol. 1. Oakland, CA, USA, 281–297.

[21] Rashid Mehmood, Guangzhi Zhang, Rongfang Bie, Hassan Dawood, and Haseeb Ahmad. 2016. Clustering by fast search and find of density peaks via heat diffusion. *Neurocomputing* 208 (2016), 210–217.

[22] David R Musser. 1997. Introspective sorting and selection algorithms. *Software: Practice and Experience* 27, 8 (1997), 983–993.

[23] Alex Rodriguez and Alessandro Laio. 2014. Clustering by fast search and find of density peaks. *Science* 344, 6191 (2014), 1492–1496.

[24] Xianjin Shi, Wanwan Wang, and Chongsheng Zhang. 2017. An Empirical Comparison of Latest Data Clustering Algorithms with State-of-the-Art. *Indonesian Journal of Electrical Engineering and Computer Science* 5, 2 (2017), 410–415.

[25] Xiao-Feng Wang and Yifan Xu. 2017. Fast clustering using adaptive density peak detection. *Statistical methods in medical research* 26, 6 (2017), 2800–2811.

[26] Yi Wang, Qixin Chen, Chongqing Kang, and Qing Xia. 2016. Clustering of electricity consumption behavior dynamics toward big data applications. *IEEE transactions on smart grid* 7, 5 (2016), 2437–2447.

[27] Juanying Xie, Hongchao Gao, Weixin Xie, Xiaohui Liu, and Philip W Grant. 2016. Robust clustering by detecting density peaks and assigning points based on fuzzy weighted K-nearest neighbors. *Information Sciences* 354 (2016), 19–40.

[28] Shuai Yang and Xipeng Shen. 2018. Falcon: A fast drop-in replacement of citation knn for multiple instance learning. In *Proceedings of the 27th ACM International Conference on Information and Knowledge Management*. ACM, 67–76.

[29] Shuai Yang and Xipeng Shen. 2018. LEEM: Lean elastic EM for Gaussian mixture model via bounds-based filtering. In *2018 IEEE International Conference on Data Mining (ICDM)*. IEEE, 677–686.

[30] Yanfeng Zhang, Shimin Chen, and Ge Yu. 2016. Efficient distributed density peaks for clustering large data sets in mapreduce. *IEEE Transactions on Knowledge and Data Engineering* 28, 12 (2016), 3218–3230.