THE UNIVERSITY OF QUEENSLAND

AUSTRALIA

High Performance Density-based Clustering on Massive Data

Junhao Gan

Master of Engineering

A thesis submitted for the degree of Doctor of Philosophy at

The University of Queensland in 2017

School of Information Technology & Electrical Engineering

Abstract

DBSCAN, a density-based clustering method for multi-dimensional points, was proposed in 1996. Since then, it has received extensive applications, while its computational hardness is still unsolved to this date. The original KDD'96 paper claimed an algorithm of $O(n \log n)$ "average run time complexity" (where n is the number of data points) without a rigorous proof. In 2013, a genuine $O(n \log n)$-time algorithm was found in 2D space under the Euclidean distance. The hardness of dimensionality $d \geq 3$ has remained open ever since.

This thesis considers the problem of computing DBSCAN clusters from scratch (assuming no existing indexes) under the Euclidean distance. We prove that, for $d \geq 3$, the problem requires $\Omega(n^{4/3})$ time to solve, unless very significant breakthroughs—ones widely believed to be impossible—could be made in theoretical computer science. Motivated by this, we propose a relaxed version of the problem called *ρ-approximate DBSCAN*, which returns the same clusters as DBSCAN, unless the clusters are "unstable" (i.e., they change once the input parameters are slightly perturbed). The ρ-approximate problem can be settled in $O(n)$ expected time regardless of the constant dimensionality d.

The thesis also enhances the previous result on the exact DBSCAN problem in 2D space. We show that, if the n data points have been pre-sorted on each dimension (i.e., one sorted list per dimension), the problem can be settled in $O(n)$ worst-case time. As a corollary, when all the coordinates are integers, the 2D DBSCAN problem can be solved in $O(n \log \log n)$ time deterministically, improving the existing $O(n \log n)$ bound.

Given the popular usage of density-based clustering approach in many applications demanding data updates, this thesis further investigates the algorithmic principles for dynamic clustering by DBSCAN. Surprisingly, we prove that the ρ-approximate version *suffers from the very same hardness when the dataset is fully dynamic*, namely, when both insertions and deletions are allowed. We also show that this issue goes away as soon as tiny further relaxation is applied, yet still ensuring the same quality of ρ-approximate DBSCAN. Our algorithms guarantee near-constant update processing, and outperform existing approaches by a factor over two orders of magnitude.

The last part of the thesis targets the scenario that the dataset cannot fit in main memory. The core contribution of this part is to show that, for any *d-dimensional grid graph* with n vertices, we

can always compute in $O(sort(n))$ I/Os—where $sort(n)$ is the I/O complexity of sorting n elements—a multiway vertex separator that serves the same algorithmic purposes as the well-known separator of [Maheshwari and Zeh, SICOMP'08] for a planar graph. This finding leads to (i) an algorithm that performs L_∞ *density-based clustering* (and hence, approximate L_p density-based clustering for any constant $p > 0$) with near-linear I/Os in any fixed dimensionality, and (ii) improved algorithms for several fundamental problems on 2D grid graphs: connected components (CC), single source shortest path (SSSP), and breadth-first search (BFS). In particular, the improvement on the CC problem owes also to *disproving* a common belief that 2D grid graphs were sparse under edge contractions.

Declaration by Author

This thesis is composed of my original work, and contains no material previously published or written by another person except where due reference has been made in the text. I have clearly stated the contribution by others to jointly-authored works that I have included in my thesis.

I have clearly stated the contribution of others to my thesis as a whole, including statistical assistance, survey design, data analysis, significant technical procedures, professional editorial advice, and any other original research work used or reported in my thesis. The content of my thesis is the result of work I have carried out since the commencement of my research higher degree candidature and does not include a substantial part of work that has been submitted to qualify for the award of any other degree or diploma in any university or other tertiary institution. I have clearly stated which parts of my thesis, if any, have been submitted to qualify for another award.

I acknowledge that an electronic copy of my thesis must be lodged with the University Library and, subject to the policy and procedures of The University of Queensland, the thesis be made available for research and study in accordance with the Copyright Act 1968 unless a period of embargo has been approved by the Dean of the Graduate School.

I acknowledge that copyright of all material contained in my thesis resides with the copyright holder(s) of that material. Where appropriate I have obtained copyright permission from the copyright holder to reproduce material in this thesis.

Publications during candidature

Journal papers:

- Junhao Gan and Yufei Tao. On the Hardness and Approximation of Euclidean DBSCAN. Accepted and to appear in ACM Transactions on Database Systems (TODS). (Best papers of SIGMOD 2015)

Conference papers:

- Junhao Gan and Yufei Tao. Dynamic Density Based Clustering. In *Proceedings of ACM Conference on Management of Data* (*SIGMOD*), pages 1493-1507, 2017.
- Miao Qiao, Junhao Gan and Yufei Tao. Range Thresholding on Streams. In *Proceedings of ACM Conference on Management of Data* (*SIGMOD*), pages 571-582, 2016.
- Junhao Gan and Yufei Tao. DBSCAN Revisited: Mis-Claim, Un-Fixability, and Approximation. In *Proceedings of ACM Conference on Management of Data* (*SIGMOD*), pages 519-530, 2015. (Winner of the Best Paper Award)

Publications included in this thesis

Junhao Gan and Yufei Tao. On the Hardness and Approximation of Euclidean DBSCAN. Accepted and to appear in ACM Transactions on Database Systems (TODS). (Best papers of SIGMOD 2015) - incorporated as Chapter 2.

Contributor	Statement of contribution
Junhao Gan (Candidate)	Designed algorithm (50%) Wrote the paper (40%) Designed experiments (70 %) Proofreading the paper (50 %)
Yufei Tao	Designed algorithm (50%) Wrote the paper (60%) Designed experiments (30 %) Proofreading the paper (50 %)

Junhao Gan and Yufei Tao. Dynamic Density Based Clustering. In *Proceedings of ACM Conference on Management of Data* (*SIGMOD*), pages 1493-1507, 2017. - incorporated as Chapter 3.

Contributor	Statement of contribution
Junhao Gan (Candidate)	Designed algorithm (50%) Wrote the paper (40%) Designed experiments (80 %) Proofreading the paper (50 %)
Yufei Tao	Designed algorithm (50%) Wrote the paper (60%) Designed experiments (20 %) Proofreading the paper (50 %)

Contributions by others to the thesis

In all of the presented research in this thesis, Prof. Yufei Tao, as my principal advisor, has provided technical guidance for formulating the problems, refinement of ideas as well as reviewing and polishing the presentation.

Statement of parts of the thesis submitted to qualify for the award of another degree

None.

Acknowledgments:

This thesis is not only a milestone of my Ph.D study, but also a summary of an important chapter of my life. I would like to give my sincere thanks to people who have supported and helped me during my Ph.D study. It is a great pleasure to express my gratitude and thankfulness to them all in my humble acknowledgment.

First and foremost, I would like to give my earnest gratitude to my advisor Prof. Yufei Tao for the continuous support in my research and my life. His strictness and high standard in research have helped me learn a lot, and his passion and fortitude in research have taught me to never give up no matter what situation is. Without his careful guidance, all my papers that have been published or submitted, as well as this thesis, would never have been completed or written. I could not have imagined having a better advisor and mentor for my Ph.D study. Being a Ph.D student under his supervision is one of the most wonderful things in my life.

I thank Prof. Hong Cheng from the bottom of my heart for her generous help since I was a research assistant in her group before my Ph.D study. I still remember the words she said to me when I was confused about the future, and her words have encouraged me to choose a tougher but better path in my research. Furthermore, I could not exactly remember how many times she has written recommendation letters for me.

I appreciate Dr. Xiaocheng Hu and Dr. Miao Qiao, who have provided me numerous of insightful and fruitful discussions. Thank you to my good friends Dr. GuangYong Chen and Dr. Junjie Ye, and my dear roommates Dr. Yu Rong and Mr. David Tong for bringing me such an enjoyable and unforgettable life.

My great thankfulness also goes to Prof. Xiaofang Zhou, who is always nice and supportive, and he has given me lots of constructive comments not only in research but also in life. I also thank the DKE group very much, which has provided me a warm and delightful atmosphere just like a big family. I acknowledge Prof. Xue Li for often sharing his study and life experience with me during lunch time. An especial big thank you to Mr. Lei Li, Dr. Bolong Zheng, Dr. XingZhong Du, Dr. Weiqing Wang and Ms. Zoe Zhang for our “Eating Group”, in which we often shared thoughts and ideas. I also thank Ms. Dan He and Mr. PingFu Chao for their generous help in my daily life.

Last but not the least, my deepest gratitude goes to my dear family: my parents, my sister and my wife. I profoundly thank my parents for giving birth to me and raising me with all the best that

they could offer to me. I sincerely appreciate my sister, who has taken the responsibility to take care of our parents these years, and thanks to her, I can pursue my dream without worrying too much about them. My greatest thankfulness is owed to my wife, who has quit her job in China and come to Australia with me. Without her, I would never be able to have delicious meals when I am back to home after school everyday. Most importantly, it is my wife who lets me know that someone is waiting for me at home and lets me realize that I owe someone a good future. My family is my most precious possession in my life. Without my family's endless support, I would never have been able to complete my Ph.D study and write this thesis.

Keywords

dbscan, density-based clustering, dynamic, updates, hopcroft hard, algorithms, computational geometry, multi-dimensional grid graphs, orthogonal separator, external memory

Australian and New Zealand Standard Research Classifications (ANZSRC)

ANZSRC code: 080604, Database Management, 100%

Fields of Research (FoR) Classification

FoR code: 0806, Information Systems, 100%

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Density-based clustering is one of the most fundamental topics in data mining. Given a set P of n points in d-dimensional space $\mathbb{R}^d$, the objective is to group the points of P into subsets—called *clusters*—such that any two clusters are separated by "sparse regions". Figure 1.1 shows two classic examples taken from [27]: the left one contains 4 snake-shaped clusters, while the right one contains 3 clusters together with some noise. The main advantage of density-based clustering (over methods such as *k-means*) is its capability of discovering clusters with arbitrary shapes (while k-means typically returns ball-like clusters).

Density-based clustering can be achieved using a variety of approaches, which differ mainly in their (i) definitions of "dense/sparse regions", and (ii) criteria of how dense regions should be connected to form clusters. In this thesis, we concentrate on DBSCAN, which is an approach invented by [27], and received the test-of-time award in KDD'14. DBSCAN characterizes "density/sparsity" by resorting to two parameters:

- ϵ: a positive real value;
- *MinPts*: a small positive constant integer.

Let $B(p, \epsilon)$ be the d-dimensional ball centered at point p with radius ϵ, where the distance metric is the Euclidean distance. $B(p, \epsilon)$ is "dense" if it covers at least *MinPts* points of P.

DBSCAN forms clusters based on the following rationale. *If $B(p, \epsilon)$ is dense, all the points in $B(p, \epsilon)$ should be added to the same cluster as p.* This creates a "chained effect": whenever a new

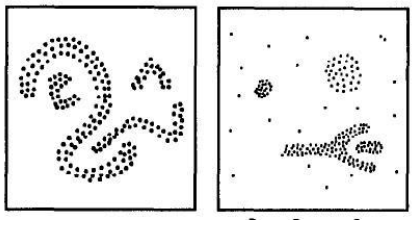


FIGURE 1.1: Examples of density-based clustering from [27]

point p' with a dense $B(p', \epsilon)$ is added to the cluster of p, all the points in $B(p', \epsilon)$ should also join the same cluster. The cluster of p continues to grow in this manner to the effect's fullest extent.

1.1 Euclidean DBSCAN Revisited: Hardness and Approximation

Previous Description of DBSCAN's Running Time. The original DBSCAN algorithm of [27] performs a *region query* for each point $p \in P$, which retrieves $B(p, \epsilon)$. Regarding the running time, Ester et al. [27] wrote:

> "*The height an R*-tree is $O(\log n)$ for a database of n points in the worst case and a query with a "small" query region has to traverse only a limited number of paths in the R*-tree. Since the Eps-Neighborhoods are expected to be small compared to the size of the whole data space, the average run time complexity of a single region query is $O(\log n)$. For each of the points of the database, we have at most one region query. Thus, the average run time complexity of DBSCAN is $O(n \log n)$.*"

The underlined statement lacks scientific rigor:

- Consider a dataset where $\Omega(n)$ points coincide at the same location. No matter how small is ϵ, for every such point p, $B(p, \epsilon)$ always covers $\Omega(n)$ points. Even just reporting the points inside $B(p, \epsilon)$ for all such p already takes $\Theta(n^2)$ time—this is true regardless of how good is the underlying R*-tree or any other index deployed.

- The notion of "average run time complexity" in the statement does not seem to follow any of the standard definitions in computer science (see, for example, Wikipedia[1]). There was no clarification on the mathematical meaning of this notion in [27], and neither was there a proof on the claimed complexity. In fact, it would have been a great result if an $O(n \log n)$ bound could indeed be proved under any of those definitions.

The "$O(n \log n)$ average run time complexity" has often been re-stated with fuzzy or even no description of the accompanying conditions. A popular textbook [33], for example, comments in Chapter 10.4.1:

> *If a spatial index is used, the computational complexity of DBSCAN is $O(n \log n)$, where n is the number of database objects. Otherwise, the complexity is $O(n^2)$.*

Similar statements have appeared in many papers: [15] (Sec 3.1), [19] (Sec 2), [25] (Chapter 5, Sec 2), [41] (Sec 2), [45] (Sec 5.4), [50] (Sec 1), [56] (Sec 2), [61] (Sec 3.3), [70] (Sec 2.2.3), [71] (Sec 5.2), mentioning just 10 papers. Several works have even utilized the $O(n \log n)$ bound as a building-brick lemma to derive new "results" incorrectly: see Sec D.1 of [42], Sec 3.2 of [57], and Sec 5.2 of [59]).

Gunawan [31] also showed that all of the subsequently improved versions of the original DBSCAN algorithm either do not compute the precise DBSCAN result (e.g., see [16, 43, 68]), or still suffer from $O(n^2)$ running time [47]. As a partial remedy, he developed a new 2D algorithm which truly runs in $O(n \log n)$ time, without assuming any indexes.

Hardness and Approximation of DBSCAN. The first part of our results was motivated by two questions:

- For $d \geq 3$, is it possible to design an algorithm that genuinely has $O(n \log n)$ time complexity? To make things easier, is it possible to achieve time complexity $O(n \log^c n)$ even for some *very large* constant c?

- If the answer to the previous question is no, is it possible to achieve linear or near-linear running time by sacrificing the quality of clusters slightly, while still being able to give a strong guarantee on the quality?

[1] *Https://en.wikipedia.org/wiki/Average-case_complexity*

We answer the above questions with the following results:

- We prove that the DBSCAN problem (computing the clusters from scratch, without assuming an existing index) requires $\Omega(n^{4/3})$ time to solve in $d \geq 3$, unless very significant breakthroughs—ones widely believed to be impossible—can be made in theoretical computer science. Note that $n^{4/3}$ is arbitrarily larger than $n \log^c n$, regardless of constant c.

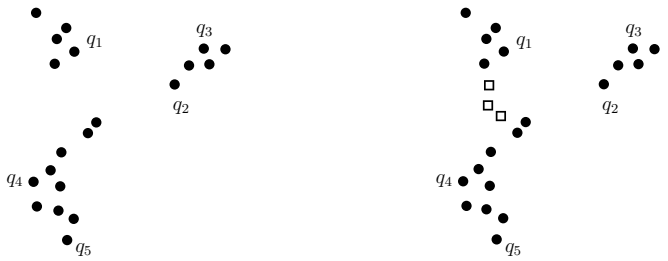
- We introduce a new concept called *ρ-approximate DBSCAN* which comes with strong assurances in both *quality* and *efficiency*. For quality, its clustering result is guaranteed to be "sandwiched" between the results of DBSCAN obtained with parameters $(\epsilon, MinPts)$ and $(\epsilon(1 + \rho), MinPts)$, respectively. For efficiency, we prove that ρ-approximate DBSCAN can be solved in linear $O(n)$ expected time, for any ϵ, arbitrarily small constant ρ, and in any fixed dimensionality d.

New 2D Algorithms. We also give a new algorithm that solves the exact DBSCAN problem in 2D space using $O(n \log n)$ time, but in a way substantially simpler than the solution of [31]. The algorithm reveals an inherent geometric connection between (exact) DBSCAN and *Delaunay graphs*. The connection is of independent interests.

Furthermore, we prove that the 2D exact DBSCAN problem can actually be settled in $O(n)$ time, provided that the n data points have been sorted along each dimension. In other words, the "hardest" component of the problem turns out to be sorting the coordinates, whereas the clustering part is easy. Immediately, this implies that 2D DBSCAN can be settled in $o(n \log n)$ time when the coordinates are integers, by utilizing fast integer-sorting algorithms [5, 34]: (i) deterministically, we achieve $O(n \log \log n)$ time—improving the $O(n \log n)$ bound of [31]; (ii) randomly, we achieve $O(n \sqrt{\log \log n})$ time in expectation.

1.2 Dynamic Euclidean DBSCAN: Hardness and Approximation

Maintaining clustering results on a dynamic data set that can be updated by insertions and deletions, is one of the most important application scenarios. An immediate question is how to properly approach the problem in the first place. An obvious attempt is to define the problem as: "an algorithm



(a) A dataset of 3 clusters (b) 2 clusters merge after insertions

Figure 1.2: Dynamic density-based clustering

should support fast updates, and in the meantime be prepared to return all the clusters any time upon requested". However, the cluster reporting *itself* already demands $\Omega(n)$ cost, where n is the number of points in P. This is at odds with the conventional database wisdom that "queries" should have response time significantly shorter than $O(n)$.

We eliminate the issue by introducing a novel query type called *cluster-group-by* (C-group-by), which makes the dynamic clustering problem much more interesting:

> Given an arbitrary *subset* Q of P, a C-group-by query groups the points of Q by the clusters they belong to.

Figure 1.2a shows a query with $Q = \{q_1, q_2, q_3, q_4, q_5\}$, which should return $\{q_1\}$, $\{q_2, q_3\}$, and $\{q_4, q_5\}$ indicating how they should be divided based on the clustering. The same query on Figure 1.2b returns $\{q_1, q_4, q_5\}$ and $\{q_2, q_3\}$.

By simply setting Q to P, the C-group-by query degenerates into returning all the clusters. In practice, however, a user is rarely interested in the entire dataset. Instead, s/he is much more likely to raise questions regarding selected objects, e.g., "*are stocks X, Y in the same cluster?*", or "*break the 10 stocks by the clusters that their profiles belong to in the entire stock database*." C-group-by queries aim to answer these questions with time *proportional only to* $|Q|$, as opposed to $|P|$.

Hardness of Dynamic DBSCAN. As discussed in Section 1.1, when $d \geq 3$, any DBSCAN algorithm must incur $\Omega(n^{4/3})$ worst-case time to cluster n static points. Unfortunately, this implies that no dynamic DBSCAN algorithm can be fast in both insertions and queries, as explained below.

Suppose, on the contrary, that an algorithm could process an insertion in $\tilde{O}(1)$ time (where notation $\tilde{O}(\cdot)$ hides a polylog factor), and a query in $\tilde{O}(|Q|)$ time. We would be able to solve the *static* DBSCAN problem using the dynamic algorithm by performing n insertions followed by a C-group-by query with

$Q = P$. The total cost would be only $\tilde{O}(n)$ which, however, is $o(n^{4/3})$—violating the impossibility result! To be practically useful, a dynamic algorithm must support an update in $\tilde{O}(1)$ time and a query in $\tilde{O}(|Q|)$ time. The above reduction suggests that no such algorithms can exist for DBSCAN, even if all the updates are insertions.

Our Findings. Lack of understanding on the computational efficiency of *dynamic* DBSCAN has become a serious issue, given the vast importance of this clustering technique, and the dynamic nature of numerous practical datasets in modern applications. Motivated by this, we present a comprehensive study on dynamic density-based clustering algorithms. Those results can be summarized as follows.

- *Fully Dynamic 2D Exact Algorithm:* When $d = 2$, we present an algorithm for (exact) DBSCAN that supports each insertion in $\tilde{O}(1)$ amortized time, and answers a C-group-by query in $\tilde{O}(|Q|)$ time.
- *Fast Insertion-Only ρ-Approximate Algorithms:* A dataset is *semi-dynamic*, if data points are only appended, but never deleted. In this case, we propose a ρ-approximate DBSCAN algorithm that supports each insertion in $\tilde{O}(1)$ amortized time, and answers a C-group-by query in $\tilde{O}(|Q|)$ time. The result holds for any fixed dimensionality d.
- *Fully Dynamic ρ-Approximate DBSCAN Is Hard!* A dataset is *fully-dynamic*, if data points can be inserted and deleted arbitrarily. We prove that, when $d \geq 3$, no ρ-approximate DBSCAN algorithm can be efficient in both updates and C-group-by queries at the same time! Specifically, such an algorithm must use $\tilde{\Omega}(n^{1/3})$ time either to process an update, or to answer a query—neither complexity is acceptable in practice (notation $\tilde{\Omega}(.)$ hides a polylog factor). This is true even if $|Q| = 2$ for all queries!
- *ρ-Double-Approx. DBSCAN and Fully Dynamic:* We show how to slightly relax ρ-approximate DBSCAN—into what we call *ρ-double-approximate DBSCAN*—to remove the above computational hardness. The relaxation leads to a fully-dynamic algorithm that processes an update in $\tilde{O}(1)$ amortized time, and answers a C-group-by query in time $\tilde{O}(|Q|)$. The new proposition preserves the clustering quality of (exact) DBSCAN in the same way (known as the *"sandwich guarantee"*) as ρ-approximate DBSCAN! In other words, the double approximation offers an *alternative* way to reach the *same* goal as ρ-approximate DBSCAN, without sharing the latter's deficiencies. The result holds for any fixed dimensionality d.

method	**update**	**C-group-by query**	**remark**
exact DBSCAN $d = 2$	$\tilde{O}(1)$	$\tilde{O}(\|Q\|)$	fully dynamic
exact DBSCAN $d \geq 3$	either $\Omega(n^{1/3})$ insertion or $\Omega(\|Q\|^{4/3})$ query		even if insertions only
ρ-approx. DBSCAN $d \geq 3$	$\tilde{O}(1)$ insertion	$\tilde{O}(\|Q\|)$	insertions only
ρ-approx. DBSCAN $d \geq 3$	either $\tilde{\Omega}(n^{1/3})$ update or $\tilde{\Omega}(n^{1/3})$ query even if $\|Q\| = 2$		fully dynamic
ρ-double-approx. DBSCAN $d \geq 3$	$\tilde{O}(1)$	$\tilde{O}(\|Q\|)$	fully dynamic

Table 1.1: Dynamic hardness of DBSCAN variants

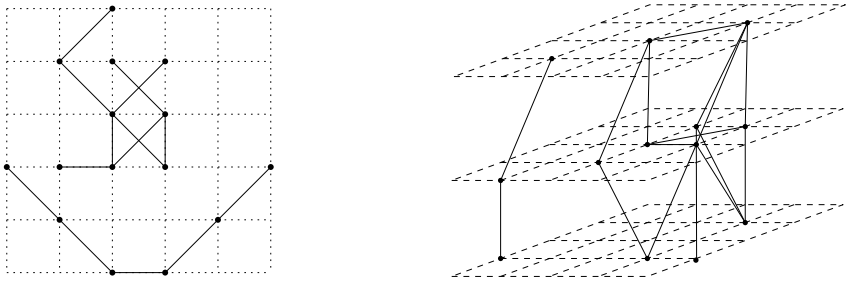
The dynamic hardness of different DBSCAN variants is summarized in Table 1.1. With these results, the dynamic tractability (i.e., polylog vs. polynomial) in all the fixed dimensionalities and update schemes has become well understood.

1.3 External Density-Based Clustering and Its Related Graph Problems

To support datasets that can not fit in the main memory, we study the density-based clustering problem (see popular textbooks [33, 64]) in the *external memory* (EM) computation model [4]. In this model, the machine is equipped with M words of (internal) memory, and a disk that has been formatted into *blocks*, each of which has B words. The values of M and B satisfy $M \geq 2B$. An *I/O* either reads a block of data from the disk into memory, or writes B words of memory into a disk block. The *cost* of an algorithm is measured in the number of I/Os performed. Define function $sort(n) = \Theta((n/B)\log_{M/B}(n/B))$, which is the I/O complexity of sorting n elements [4].

Interestingly, our techniques to achieve I/O-efficient algorithms for density-based clustering are of independent interests. Specifically, we consider a class of graphs called *d-dimensional grid graphs*. As defined by Nodine et al. [55], every member of the class is an undirected graph $G = (V, E)$ with two properties:

- Each vertex $v \in V$ is a distinct point in space $\mathbb{N}^d$, where $\mathbb{N}$ represents the set of integers.
- If E has an edge between $v_1, v_2 \in V$, then (i) the two vertices are distinct (i.e., no self-loops), and (ii) they differ by at most 1 in their coordinates on *every* dimension.



(a) A (traditional) 2-dimensional grid graph (b) A 3-dimensional grid graph

FIGURE 1.3: Multidimensional grid graphs

See Figure 1.3 for two illustrative examples. For $d = O(1)$, such a graph is *sparse*, namely, $|E| = O(|V|)$, because each vertex can have a degree at most $3^d = O(1)$.

Particularly, we are interested in certain vertex separators of a d-dimensional grid graph $G = (V, E)$. Specifically, given a positive integer $r \leq |V|$, a set $S \subseteq V$ is an *r-separator* if it satisfies:

- $|S| = O(|V|/r^{1/d})$
- Removing the vertices of S disconnects G into $h = O(|V|/r)$ subgraphs $G_i = (V_i, E_i)$ for $i \in [1, h]$ such that
  - $|V_i| = O(r)$;
  - No vertex of V_i is adjacent to any vertex of V_j, if $i \neq j$.
  - The vertices of V_i are adjacent to $O(r^{1-1/d})$ vertices of S (if a vertex $v \in V_i$ is adjacent to some vertex in S, v is said to be a *boundary vertex* of G_i).

Such separators are known [51, 63] to exist for any $r \in [1, |V|]$. Of special importance are M-separators, which are crucial for several fundamental graph problems, as described shortly. A primary contribution of Chapter 4 is an I/O-efficient algorithm for computing M-separator in $O(sort(|V|))$ I/Os, which uses ideas different from those of [51, 63].

Our next result stems from an unexpected source. It has been stated [67, 73] that the CCs of a 2D grid graph $G = (V, E)$ can be computed in $O(sort(|V|))$ I/Os. This is based on the belief that a 2D grid graph has the property of being *sparse under edge contractions*. Specifically, an *edge contraction* removes an edge between vertices v_1, v_2 from G, combines v_1, v_2 into a single vertex v, replaces every edge adjacent to v_1 or v_2 with an edge adjacent to v, and finally removes duplicate

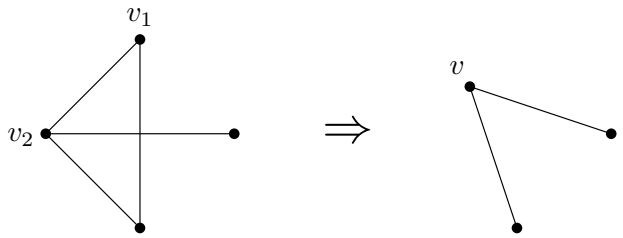


Figure 1.4: Contracting the edge between v_1, v_2 from the graph on the left produces the graph on the right

edges thus produced—all these steps then create a new graph; see Figure 1.4. The aforementioned property says that, if one performs any sequence of edge contractions to obtain a resulting graph $G' = (V', E')$, G' must still be sparse, namely, $|E'| = O(|V'|)$. Surprising enough, the belief—perhaps too intuitive—seemed to have been taken for granted, such that no proof has ever been documented.

We formally *disprove* this belief. Specifically, we show that there exists a 2D grid graph that is not sparse under edge contractions. With the belief invalidated, the best existing deterministic algorithm for computing the CCs of a 2D grid graph requires an I/O complexity that is the minimum of $O(sort(|V|) \cdot \log\log B)$ [54] and $O(sort(|V|) \cdot \log(|V|/|M|))$ [37]. By our M-separator construction algorithm, we improve this result with a $O(sort(|V|))$-I/O CC algorithm on d-dimensional grid graphs.

Equipped with our CC algorithm, the density-based clustering problem under L_∞ norm can be settled in $O(sort(n))$ I/Os for $d = 2$ and 3, and $O((n/B)\log^{d-2}_{M/B}(n/B))$ I/Os for $d \geq 4$. As we will see in Chapter 4, L_∞ density-based clustering essentially is an approximation of density-based clustering under L_p norm for any $p > 0$.

In addition, our M-separator construction algorithm also yeilds improved algorithms for the *single source shortest path* (SSSP) problem and *breadth first search* (BFS) on 2D grid graphs. All these algorithms are important for *flow analysis* [8, 9], *nearest-neighbor queries* [60, 72], and *navigation* [44] on *terrains* [2, 8, 9], *land surfaces* [22, 44, 60, 72] and road networks.

1.4 Organization of the Thesis

The rest of the thesis is organized as follows. In Chapter 2, we revisit the DBSCAN problem, show the hardness result of DBSCAN problem for $d \geq 3$, propose the notion of ρ-approximate DBSCAN, and give two new efficient algorithms in 2D space. In Chapter 3, we conduct a comprehensive study

on the DBSCAN problem under semi-dynamic and fully dynamic settings, where both the hardness results and the corresponding solutions are included. In Chapter 4, we study the density-based clustering problem in EM model, propose an I/O efficient algorithm for constructing an M-separator on d-dimensional grid graphs, and improve the state-of-the-art results on several related fundamental problems. Finally, we conclude the thesis by a summary in Chapter 5.

Chapter 2

On the Hardness of Euclidean DBSCAN and Its Approximation

In this chapter, Section 2.1 reviews the previous work related to ours. Section 2.2 provides theoretical evidence on the computational hardness of DBSCAN, and presents a sub-quadratic algorithm for solving the problem exactly. Section 2.3 proposes ρ-approximate DBSCAN, elaborates on our algorithm, and establishes its quality and efficiency guarantees. Section 2.4 presents new algorithms for solving the exact DBSCAN problem in 2D space. Section 2.5 discusses several issues related to the practical performance of different algorithms and implementations. Section 2.6 evaluates all the exact and approximation algorithms with extensive experimentation. Finally, Section 2.7 concludes the chapter with a summary of findings.

2.1 Related Work

Section 2.1.1 reviews the DBSCAN definitions as set out by [27]. Section 2.1.2 describes the 2D algorithm in [31] that solves the problem genuinely in $O(n \log n)$ time. Section 2.1.3 points out several results from computational geometry which will be needed to prove the intractability of DBSCAN later.

2.1.1 Definitions

As before, let P be a set of n points in d-dimensional space $\mathbb{R}^d$. Given two points $p, q \in \mathbb{R}^d$, we denote by $dist(p, q)$ the Euclidean distance between p and q. Denote by $B(p, r)$ the ball centered at a point $p \in \mathbb{R}^d$ with radius r. Remember that DBSCAN takes two parameters: ϵ and *MinPts*.

Definition 2.1. *A point $p \in P$ is a* **core point** *if $B(p, \epsilon)$ covers at least MinPts points of P (including p itself).*

If p is not a core point, it is said to be a *non-core point*. To illustrate, suppose that P is the set of points in Figure 2.1, where $MinPts = 4$ and the two circles have radius ϵ. Core points are shown in black, and non-core points in white.

Definition 2.2. *A point $q \in P$ is* **density-reachable** *from $p \in P$ if there is a sequence of points $p_1, p_2, ..., p_t \in P$ (for some integer $t \geq 2$) such that:*

- $p_1 = p$ *and* $p_t = q$
- $p_1, p_2, ..., p_{t-1}$ *are core points*
- $p_{i+1} \in B(p_i, \epsilon)$ *for each* $i \in [1, t-1]$.

Note that points p and q do *not* need to be different. In Figure 2.1, for example, o_1 is density-reachable from itself; o_{10} is density-reachable from o_1 and from o_3 (through the sequence o_3, o_2, o_1, o_{10}). On the other hand, o_{11} is *not* density-reachable from o_{10} (recall that o_{10} is not a core point).

Definition 2.3. *A* **cluster** *C is a non-empty subset of P such that:*

- *(Maximality) If a core point $p \in C$, then all the points density-reachable from p also belong to C.*
- *(Connectivity) For any points $p_1, p_2 \in C$, there is a point $p \in C$ such that both p_1 and p_2 are density-reachable from p.*

Definition 2.3 implies that each cluster contains at least a core point (i.e., p). In Figure 2.1, $\{o_1, o_{10}\}$ is *not* a cluster because it does not involve all the points density-reachable from o_1. On the other hand, $\{o_1, o_2, o_3, ..., o_{10}\}$ is a cluster.

Ester et al. [27] gave a nice proof that P has a unique set of clusters, which gives rise to:

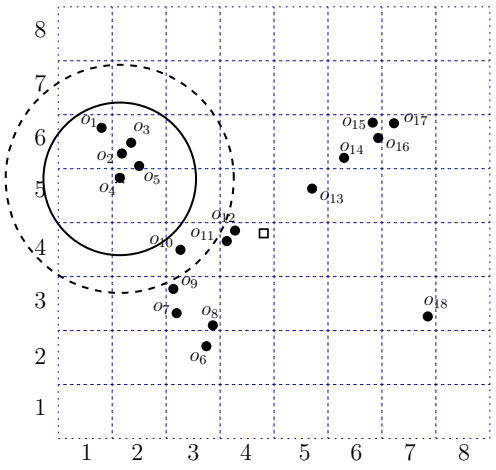


Figure 2.1: An example dataset (the two circles have radius ϵ; $MinPts = 4$)

Problem 2.1. *The* **DBSCAN problem** *is to find the unique set C of clusters of P.*

Given the input P in Figure 2.1, the problem should output two clusters: $C_1 = \{o_1, o_2, ..., o_{10}\}$ and $C_2 = \{o_{10}, o_{11}, ..., o_{17}\}$.

Remark. A cluster can contain both core and non-core points. Any non-core point p in a cluster is called a *border point*. Some points may not belong to any clusters at all; they are called *noise points*. In Figure 2.1, o_{10} is a border point, while o_{18} is noise.

The clusters in C are not necessarily disjoint (e.g., o_{10} belongs to both C_1 and C_2 in Figure 2.1). In general, if a point p appears in more than one cluster in C, then p must be a border point (see Lemma 2 of [27]). In other words, a core point always belongs to a unique cluster.

2.1.2 The 2D Algorithm with Genuine $O(n \log n)$ Time

Next, we explain in detail the algorithm of [31], which solves the DBSCAN problem in 2D space in $O(n \log n)$ time. The algorithm imposes an arbitrary grid T in the data space $\mathbb{R}^2$, where each cell of T is a $(\epsilon/\sqrt{2}) \times (\epsilon/\sqrt{2})$ square. Without loss of generality, we assume that no point of P falls on any boundary line of T (otherwise, move T infinitesimally to make this assumption hold). Figure 2.2a shows a grid on the data of Figure 2.1. Note that any two points in the same cell are at most distance ϵ apart. A cell c of T is *non-empty* if it contains at least one point of P; otherwise, c is *empty*. Clearly, there can be at most n non-empty cells.

The algorithm then launches a *labeling process* to decide for each point $p \in P$ whether p is core or non-core. Denote by $P(c)$ the set of points of P covered by c. A cell c is a *core cell* if $P(c)$ contains at least one core point. Denote by S_{core} the set of core cells in T. In Figure 2.2a where $MinPts = 4$, there are 6 core cells as shown in gray (core points are in black, and non-core points in white).

Let $G = (V, E)$ be a graph defined as follows:

- Each vertex in V corresponds to a distinct core cell in S_{core}.
- Given two different cells $c_1, c_2 \in S_{core}$, E contains an edge between c_1 and c_2 if and only if there exist *core* points $p_1 \in P(c_1)$ and $p_2 \in P(c_2)$ such that $dist(p_1, p_2) \leq \epsilon$.

Figure 2.2b shows the G for Figure 2.2a (note that there is no edge between cells c_4 and c_6).

The algorithm then proceeds by finding all the connected components of G. Let k be the number of connected components, V_i $(1 \leq i \leq k)$ be the set of vertices in the i-th connected component, and $P(V_i)$ be the set of core points covered by the cells of V_i. Then:

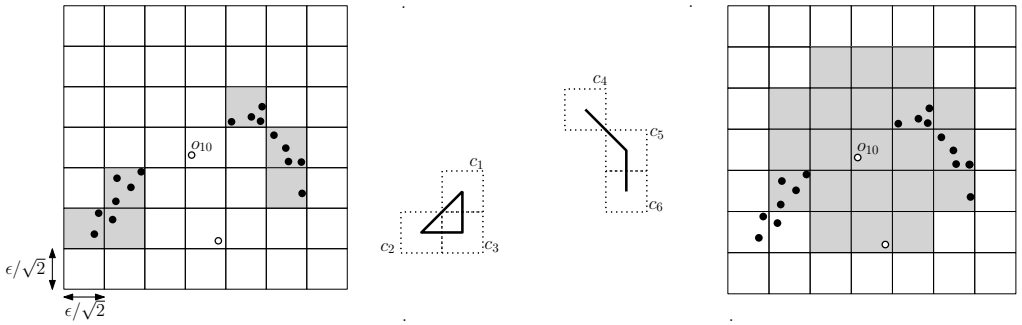
Lemma 2.1 ([31]). *The number k is also the number of clusters in P. Furthermore, $P(V_i)$ $(1 \leq i \leq k)$ is exactly the set of core points in the i-th cluster.*

Figure 2.2b, $k = 2$, and $V_1 = \{c_1, c_2, c_3\}, V_2 = \{c_4, c_5, c_6\}$. It is easy to verify the correctness of Lemma 2.1 on this example.

Labeling Process. Let c_1 and c_2 be two different cells in T. They are *ϵ-neighbors* of each other if the minimum distance between them is less than ϵ. Figure 2.2c shows in gray all the ϵ-neighbor cells of the cell covering o_{10}. It is easy to see that each cell has at most 21 ϵ-neighbors. If a non-empty cell c contains at least $MinPts$ points, then *all* those points must be core points.

Now consider a cell c with $|P(c)| < MinPts$. Each point $p \in P(c)$ may or may not be a core point. To find out, the algorithm simply calculates the distances between p and *all* the points covered by *each* of the ϵ-neighbor cells of c. This allows us to know exactly the size of $|B(p, \epsilon)|$, and hence, whether p is core or non-core. For example, in Figure 2.2c, for $p = o_{10}$, we calculate the distance between o_{10} and all the points in the gray cells to find out that o_{10} is a non-core point.

Computation of *G*. Fix a core cell c_1. We will explain how to obtain the edges incident on c_1 in E. Let c_2 be a core cell that is an ϵ-neighbor of c_1. For each core point $p \in P(c_1)$, we find the core point



(a) Core cells are shown in gray

(b) Graph G

(c) ϵ-neighbor cells (in gray) of the cell of o_{10}

Figure 2.2: DBSCAN with a grid ($MinPts = 4$)

$p' \in c_2$ that is the nearest to p. If $dist(p, p') \leq \epsilon$, an edge (c_1, c_2) is added to G. On the other hand, if all such $p \in P(c_1)$ have been tried but still no edge has been created, we conclude that E has no edge between c_1, c_2.

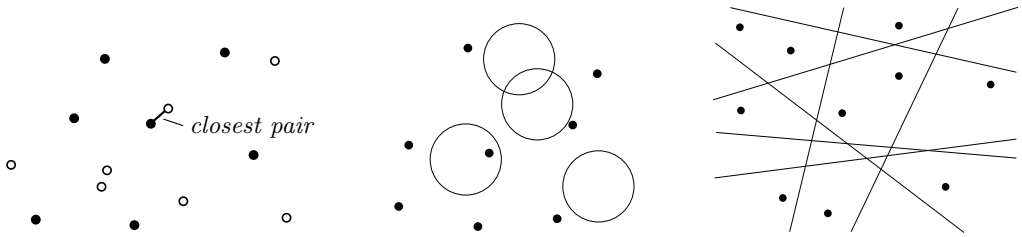
As a corollary of the above, each core cell c_1 has $O(1)$ incident edges in E (because it has $O(1)$ ϵ-neighbors). In other words, E has only a linear number $O(n)$ of edges.

Assigning Border Points. Recall that each $P(V_i)$ $(1 \leq i \leq k)$ includes only the core points in the i-th cluster of P. It is still necessary to assign each non-core point q (i.e., border point) to the appropriate clusters. The principle of doing so is simple: *if p is a core point and $dist(p, q) \leq \epsilon$, then q should be added to the (unique) cluster of p.* To find all such core points p, [31] adopted the following simple algorithm. Let c be the cell where q lies. For each ϵ-neighbor cell c' of c, simply calculate the distances from q to *all* the core points in c'.

Running Time. Gunawan [31] showed that, other than the computation of G, the rest of the algorithm runs in $O(MinPts \cdot n) = O(n)$ expected time or $O(n \log n)$ worst-case time. The computation of G requires $O(n)$ nearest neighbor queries, each of which can be answered in $O(\log n)$ time after building a Voronoi diagram for each core cell. Therefore, the overall execution time is bounded by $O(n \log n)$.

2.1.3 Some Geometric Results

Bichromatic Closest Pair (BCP). Let P_1, P_2 be two sets of points in $\mathbb{R}^d$ for some constant d. Set $m_1 = |P_1|$ and $m_2 = |P_2|$. The goal of the BCP problem is to find a pair of points $(p_1, p_2) \in P_1 \times P_2$



(a) BCP (b) USEC (c) Hopcroft

FIGURE 2.3: Three relevant geometric problems

with the smallest distance, namely, $dist(p_1, p_2) \leq dist(p'_1, p'_2)$ for any $(p'_1, p'_2) \in P_1 \times P_2$. Figure 2.3 shows the closest pair for a set of black points and a set of white points.

In 2D space, it is well-known that BCP can be solved in $O(m_1 \log m_1 + m_2 \log m_2)$ time. The problem is much more challenging for $d \geq 3$, for which currently the best result is due to [3]:

Lemma 2.2 ([3]). *For any fixed dimensionality $d \geq 4$, there is an algorithm solving the BCP problem in*

$$O\left((m_1 m_2)^{1-\frac{1}{\lceil d/2 \rceil + 1}+\delta'} + m_1 \log m_2 + m_2 \log m_1\right)$$

expected time, where $\delta' > 0$ can be an arbitrarily small constant. For $d = 3$, the expected running time can be improved to

$$O((m_1 m_2 \cdot \log m_1 \cdot \log m_2)^{2/3} + m_1 \log^2 m_2 + m_2 \log^2 m_1)).$$

Spherical Emptiness and Hopcroft. Let us now introduce the *unit-spherical emptiness checking* (USEC) *problem*:

> Let S_{pt} be a set of points, and S_{ball} be a set of balls *with the same radius*, all in data space $\mathbb{R}^d$, where the dimensionality d is a constant. The objective of USEC is to determine whether there is a point of S_{pt} that is covered by some ball in S_{ball}.

For example, in Figure 2.3b, the answer is *yes*.

Set $n = |S_{pt}| + |S_{ball}|$. In 3D space, the USEC problem can be solved in $O(n^{4/3} \cdot \log^{4/3} n)$ expected time [3]. Finding a 3D USEC algorithm with running time $o(n^{4/3})$ is a big open problem in computational geometry, and is widely believed to be impossible; see [23].

Strong hardness results are known about USEC when the dimensionality d is higher, owing to an established connection between the problem to the *Hopcroft's problem*:

Let S_{pt} be a set of points, and S_{line} be a set of lines, all in data space $\mathbb{R}^2$ (note that the dimensionality is always 2). The goal of the Hopcroft's problem is to determine whether there is a point in S_{pt} that lies on some line of S_{line}.

For example, in Figure 2.3c, the answer is *no*.

The Hopcroft's problem can be settled in time slightly higher than $O(n^{4/3})$ time (see [48] for the precise bound), where $n = |S_{pt}| + |S_{line}|$. It is widely believed [23] that $\Omega(n^{4/3})$ is a lower bound on how fast the problem can be solved. In fact, this lower bound has already been proved on a broad class of algorithms [24].

It turns out that the Hopcroft's problem is a key reason of difficulty for a large number of other problems [23]. We say that a problem X is *Hopcroft hard* if an algorithm solving X in $o(n^{4/3})$ time implies an algorithm solving the Hopcroft's problem in $o(n^{4/3})$ time. In other words, a lower bound $\Omega(n^{4/3})$ on the time of solving the Hopcroft's problem implies the same lower bound on X.

In [24], Erickson proved the following relationship between USEC and the Hopcroft's problem:

Lemma 2.3 ([24])**.** *The USEC problem in any dimensionality $d \geq 5$ is Hopcroft hard.*

2.2 DBSCAN in Dimensionality 3 and Above

This section paves the way towards approximate DBSCAN, which is the topic of the next section. In Section 2.2.1, we establish the computational hardness of DBSCAN in practice via a novel reduction from the USEC problem (see Section 2.1.3). For practitioners that *insist* on applying this clustering method with the utmost accuracy, in Section 2.2.2, we present a new exact DBSCAN algorithm that terminates in a sub-quadratic time complexity.

2.2.1 Hardness of DBSCAN

We will prove:

Theorem 2.1. *The following statements are true about the DBSCAN problem:*

- *It is Hopcroft hard in any dimensionality $d \geq 5$. Namely, the problem requires $\Omega(n^{4/3})$ time to solve, unless the Hopcroft problem can be settled in $o(n^{4/3})$ time.*

- *When $d = 3$ (and hence, $d = 4$), the problem requires $\Omega(n^{4/3})$ time to solve, unless the USEC problem can be settled in $o(n^{4/3})$ time.*

As mentioned in Section 2.1.3, it is widely believed that neither the Hopcroft problem nor the USEC problem can be solved in $o(n^{4/3})$ time—any such algorithm would be a celebrated breakthrough in theoretical computer science.

Proof of Theorem 2.1. We observe a subtle connection between USEC and DBSCAN:

Lemma 2.4. *For any dimensionality d, if we can solve the DBSCAN problem in $T(n)$ time, then we can solve the USEC problem in $T(n) + O(n)$ time.*

Proof. Recall that the USEC problem is defined by a set S_{pt} of points and a set S_{ball} of balls with equal radii, both in $\mathbb{R}^d$. Denote by $\mathcal{A}$ a DBSCAN algorithm in $\mathbb{R}^d$ that runs in $T(m)$ time on m points. Next, we describe an algorithm that deploys $\mathcal{A}$ as a *black box* to solve the USEC problem in $T(n) + O(n)$ time, where $n = |S_{pt}| + |S_{ball}|$.

Our algorithm is simple:

1. Obtain P, which is the union of S_{pt} and the set of centers of the balls in S_{ball}.
2. Set ϵ to the identical radius of the balls in S_{ball}.
3. Run $\mathcal{A}$ to solve the DBSCAN problem on P with this ϵ and $MinPts = 1$.
4. If any point in S_{pt} and any center of S_{ball} belong to the same cluster, then return *yes* for the USEC problem (namely, a point in S_{pt} is covered by some ball in S_{ball}). Otherwise, return *no*.

It is fundamental to implement the above algorithm in $T(n) + O(n)$ time. Next, we prove its correctness.

Case 1: We return yes. We will show that in this case there is indeed a point of S_{pt} that is covered by some ball in S_{ball}.

Recall that a *yes* return means a point $p \in S_{pt}$ and the center q of some ball in S_{ball} have been placed in the same cluster, which we denote by C. By connectivity of Definition 2.3, there exists a point $z \in C$ such that both p and q are density-reachable from z.

By setting $MinPts = 1$, we ensure that *all* the points in P are core points. In general, if a *core point* p_1 is density-reachable from p_2 (which by definition must be a core point), then p_2 is also density-reachable from p_1 (as can be verified by Definition 2.2). This means that z is density-reachable from p, which—together with the fact that q is density-reachable from z—shows that q is density-reachable from p.

It thus follows by Definition 2.2 that there is a sequence of points $p_1, p_2, ..., p_t \in P$ such that (i) $p_1 = p, p_t = q$, and (ii) $dist(p_i, p_{i+1}) \leq \epsilon$ for each $i \in [1, t-1]$. Let k be the smallest $i \in [2, t]$ such that p_i is the center of a ball in S_{ball}. Note that k definitely exists because p_t is such a center. It thus follows that p_{k-1} is a point from S_{pt}, and that p_{k-1} is covered by the ball in S_{ball} centered at p_k.

Case 2: We return no. We will show that in this case no point of S_{pt} is covered by any ball in S_{ball}.

This is in fact very easy. Suppose on the contrary that a point $p \in S_{pt}$ is covered by a ball of S_{ball} centered at q. Thus, $dist(p, q) \leq \epsilon$, namely, q is density-reachable from p. Then, by maximality of Definition 2.3, q must be in the cluster of p (recall that all the points of P are core points). This contradicts the fact that we returned *no*. □

Theorem 2.1 immediately follows from Lemmas 2.3 and 2.4.

2.2.2 A New Exact Algorithm for $d \geq 3$

It is well-known that DBSCAN can be solved in $O(n^2)$ time (e.g., see [64]) in any constant dimensionality d. Next, we show that it is possible to *always* terminate in $o(n^2)$ time regardless of the constant d. Our algorithm extends that of [31] with two ideas:

- Use a d-dimensional grid T with an appropriate side length for its cells.
- Compute the edges of the graph G with a BCP algorithm (as opposed to nearest neighbor search).

Next, we explain the details. T is now a grid on $\mathbb{R}^d$ where each cell of T is a d-dimensional hypersquare with side length $\epsilon/\sqrt{d}$. As before, this ensures that any two points in the same cell are within distance ϵ from each other.

The algorithm description in Section 2.1.2 carries over to any $d \geq 3$ almost *verbatim*. The only difference is the way we compute the edges of G. Given core cells c_1 and c_2 that are ϵ-neighbors of each other, we solve the BCP problem on the sets of core points in c_1 and c_2, respectively. Let (p_1, p_2) be the pair returned. We add an edge (c_1, c_2) to G *if and only if* $dist(p_1, p_2) \leq \epsilon$.

The adapted algorithm achieves the following efficiency guarantee:

Theorem 2.2. *For any fixed dimensionality $d \geq 4$, there is an algorithm solving the DBSCAN problem in $O(n^{2-\frac{2}{\lceil d/2 \rceil+1}+\delta})$ expected time, where $\delta > 0$ can be an arbitrarily small constant. For $d = 3$, the running time can be improved to $O((n \log n)^{4/3})$ expected.*

Proof. It suffices to analyze the time used by our algorithm to generate the edges of G. The other parts of the algorithm use $O(n)$ expected time, following the analysis of [31].

Let us consider first $d \geq 4$. First, fix the value of δ in Theorem 2.2. Define: $\lambda = \frac{1}{\lceil d/2 \rceil+1} - \delta/2$. Given a core cell c, we denote by m_c the number of core points in c. Then, by Lemma 2.2, the time we spend generating the edges of G is

$$\sum_{\substack{\epsilon\text{-neighbor} \\ \text{core cells } c,\, c'}} O\left((m_c m_{c'})^{1-\lambda} + m_c \log m_{c'} + m_{c'} \log m_c\right). \tag{2.1}$$

To bound the first term, we derive

$$\begin{aligned}
&\sum_{\epsilon\text{-neighbor core cells } c,\, c'} O\left((m_c m_{c'})^{1-\lambda}\right) \\
&= \sum_{\substack{\epsilon\text{-neighbor} \\ \text{core cells } c,\, c' \\ \text{s.t. } m_c \leq m_{c'}}} O\left((m_c m_{c'})^{1-\lambda}\right) + \sum_{\substack{\epsilon\text{-neighbor} \\ \text{core cells } c,\, c' \\ \text{s.t. } m_c > m_{c'}}} O\left((m_c m_{c'})^{1-\lambda}\right) \\
&= \sum_{\substack{\epsilon\text{-neighbor} \\ \text{core cells } c,\, c' \\ \text{s.t. } m_c \leq m_{c'}}} O\left(m_{c'} \cdot m_c^{1-2\lambda}\right) + \sum_{\substack{\epsilon\text{-neighbor} \\ \text{core cells } c,\, c' \\ \text{s.t. } m_c > m_{c'}}} O\left(m_c \cdot m_{c'}^{1-2\lambda}\right) \\
&= \sum_{\substack{\epsilon\text{-neighbor} \\ \text{core cells } c,\, c' \\ \text{s.t. } m_c \leq m_{c'}}} O\left(m_{c'} \cdot n^{1-2\lambda}\right) + \sum_{\substack{\epsilon\text{-neighbor} \\ \text{core cells } c,\, c' \\ \text{s.t. } m_c > m_{c'}}} O\left(m_c \cdot n^{1-2\lambda}\right) \\
&= O\Big(n^{1-2\lambda} \sum_{\epsilon\text{-neighbor core cells } c,\, c'} m_c\Big) = O(n^{2-2\lambda})
\end{aligned}$$

where the last equality used the fact that c has only $O(1)$ ϵ-neighbor cells as long as d is a constant (and hence, m_c can be added only $O(1)$ times). The other terms in (2.1) are easy to bound:

$$\sum_{\epsilon\text{-neighbor core cells } c,\, c'} O\left(m_c \log m_{c'} + m_{c'} \log m_c\right)$$
$$= \sum_{\epsilon\text{-neighbor core cells } c,\, c'} O\left(m_c \log n + m_{c'} \log n\right) = O(n \log n).$$

In summary, we spend $O(n^{2-2\lambda} + n \log n) = O(n^{2-\frac{2}{\lceil d/2 \rceil + 1} + \delta})$ time generating the edges of E. This proves the part of Theorem 2.2 for $d \geq 4$. An analogous analysis based on the $d = 3$ branch of Lemma 2.2 establishes the other part of Theorem 2.2. □

It is worth pointing out that the running time of our 3D algorithm nearly matches the lower bound in Theorem 2.1.

2.3 ρ-Approximate DBSCAN

The hardness result in Theorem 2.1 indicates the need of resorting to approximation if one wants to achieve near-linear running time for $d \geq 3$. In Section 2.3.1, we introduce the concept of ρ-approximate DBSCAN designed to replace DBSCAN on large datasets. In Section 2.3.2, we establish a strong quality guarantee of this new form of clustering. In Sections 2.3.3 and 2.3.4, we propose an algorithm for solving the ρ-approximate DBSCAN problem in time *linear* to the dataset size.

2.3.1 Definitions

As before, let P be the input set of n points in $\mathbb{R}^d$ to be clustered. We still take parameters ϵ and *MinPts*, but in addition, also a third parameter ρ, which can be any arbitrarily small positive constant, and controls the degree of approximation.

Next, we re-visit the basic definitions of DBSCAN in Section 2.1, and modify some of them to their "ρ-approximate versions". First, the notion of *core/non-core point* remains the same as Definition 2.1. The concept of *density-reachability* in Definition 2.2 is also inherited directly, but we will also need:

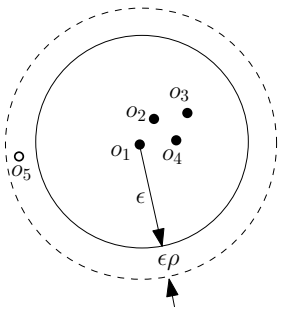


Figure 2.4: Density-reachability and ρ-approximate density-reachability ($MinPts = 4$)

Definition 2.4. *A point $q \in P$ is* **ρ-approximate density-reachable** *from $p \in P$ if there is a sequence of points $p_1, p_2, ..., p_t \in P$ (for some integer $t \geq 2$) such that:*

- $p_1 = p$ *and* $p_t = q$
- $p_1, p_2, ..., p_{t-1}$ *are core points*
- $p_{i+1} \in B(p_i, \epsilon(1 + \rho))$ *for each* $i \in [1, t - 1]$.

Note the difference between the above and Definition 2.2: in the third bullet, the radius of the ball is increased to $\epsilon(1 + \rho)$. To illustrate, consider a small input set P as shown in Figure 2.4. Set $MinPts = 4$. The inner and outer circles have radii ϵ and $\epsilon(1 + \rho)$, respectively. Core and non-core points are in black and white, respectively. Point o_5 is ρ-approximate density-reachable from o_3 (via sequence: o_3, o_2, o_1, o_5). However, o_5 is *not* density-reachable from o_3.

Definition 2.5. *A* **ρ-approximate cluster** *C is a non-empty subset of P such that:*

- *(Maximality) If a core point $p \in C$, then all the points density-reachable from p also belong to C.*
- *(ρ-Approximate Connectivity) For any points $p_1, p_2 \in C$, there exists a point $p \in C$ such that both p_1 and p_2 are ρ-approximate density-reachable from p.*

Note the difference between the above and the original cluster formulation (Definition 2.1): the connectivity requirement has been weakened into ρ-approximate connectivity. In Figure 2.4, both $\{o_1, o_2, o_3, o_4\}$ and $\{o_1, o_2, o_3, o_4, o_5\}$ are ρ-approximate clusters.

Problem 2.2. *The* **ρ-approximate DBSCAN problem** *is to find a set $\mathscr{C}$ of ρ-approximate clusters of P such that every core point of P appears in exactly one ρ-approximate cluster.*

Unlike the original DBSCAN problem, the ρ-approximate version may not have a unique result. In Figure 2.4, for example, it is legal to return either $\{o_1, o_2, o_3, o_4\}$ or $\{o_1, o_2, o_3, o_4, o_5\}$. Nevertheless, *any* result of the ρ-approximate problem comes with the quality guarantee to be proved next.

2.3.2 A Sandwich Theorem

Both DBSCAN and ρ-approximate DBSCAN are parameterized by ϵ and *MinPts*. It would be perfect if they can always return exactly the same clustering results. Of course, this is too good to be true. Nevertheless, in this subsection, we will show that this is *almost* true: the result of ρ-approximate DBSCAN is guaranteed to be somewhere between the (exact) DBSCAN results obtained by $(\epsilon, \mathit{MinPts})$ and by $(\epsilon(1+\rho), \mathit{MinPts})$! It is well-known that the clusters of DBSCAN rarely differ considerably when ϵ changes by just a small factor—in fact, if this really happens, it suggests that the choice of ϵ is very bad, such that the exact clusters are not stable anyway (we will come back to this issue later).

Let us define:

- $\mathscr{C}_1$ as the set of clusters of DBSCAN with parameters $(\epsilon, \mathit{MinPts})$
- $\mathscr{C}_2$ as the set of clusters of DBSCAN with parameters $(\epsilon(1+\rho), \mathit{MinPts})$.
- $\mathscr{C}$ as an arbitrary set of clusters that is a legal result of (ϵ, *MinPts*, ρ)-approx-DBSCAN.

The next theorem formalizes the quality assurance mentioned earlier:

Theorem 2.3 (Sandwich Quality Guarantee)**.** *The following statements are true:*

1. *For any cluster $C_1 \in \mathscr{C}_1$, there is a cluster $C \in \mathscr{C}$ such that $C_1 \subseteq C$.*
2. *For any cluster $C \in \mathscr{C}$, there is a cluster $C_2 \in \mathscr{C}_2$ such that $C \subseteq C_2$.*

Proof. To prove Statement 1, let p be an arbitrary core point in C_1. Then, C_1 is precisely the set of points in P density-reachable from p.[1] In general, if a point q is density-reachable from p in

[1]This should be folklore but here is a proof. By maximality of Definition 2.3, all the points density-reachable from p are in C_1. On the other hand, let q be any point in C_1. By connectivity, p and q are both density-reachable from a point z. As p is a core point, we know that z is also density-reachable from p. Hence, q is density-reachable from p.

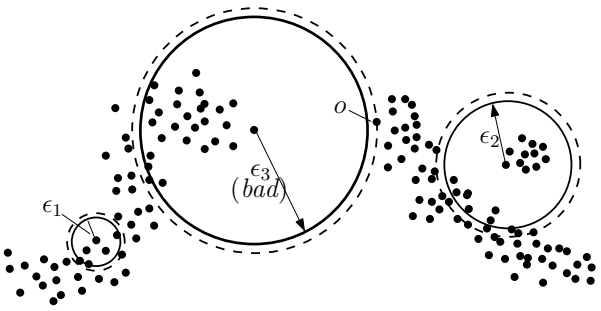


FIGURE 2.5: Good and bad choices of ϵ

$(\epsilon, MinPts)$-exact-DBSCAN, q is also density-reachable from p in $(\epsilon, MinPts, \rho)$-approx-DBSCAN. By maximality of Definition 2.5, if C is the cluster in $\mathscr{C}$ containing p, then all the points of C_1 must be in C.

To prove Statement 2, consider an arbitrary core point $p \in C$ (there must be one by Definition 2.5). In $(\epsilon(1+\rho), MinPts)$-exact-DBSCAN, p must also be a core point. We choose C_2 to be the cluster of $\mathscr{C}_2$ where p belongs. Now, fix an arbitrary point $q \in C$. In $(\epsilon, MinPts, \rho)$-approx-DBSCAN, by ρ-approximate connectivity of Definition 2.5, we know that p and q are both ρ-approximate reachable from a point z. This implies that z is also ρ-approximate reachable from p. Hence, q is ρ-approximate reachable from p. This means that q is density-reachable from p in $(\epsilon(1+\rho), MinPts)$-exact-DBSCAN, indicating that $q \in C_2$. □

Here is an alternative, more intuitive, interpretation of Theorem 2.3:

- Statement 1 says that if two points belong to the same cluster of DBSCAN with parameters $(\epsilon, MinPts)$, they are *definitely* in the same cluster of ρ-approximate DBSCAN with the same parameters.

- On the other hand, a cluster of ρ-approximate DBSCAN parameterized by $(\epsilon, MinPts)$ may also contain two points p_1, p_2 that are in different clusters of DBSCAN with the same parameters. However, this is not bad because Statement 2 says that as soon as the parameter ϵ increases to $\epsilon(1+\rho)$, p_1 and p_2 will fall into the same cluster of DBSCAN!

Figure 2.5 illustrates the effects of approximation. How many clusters are there? Interestingly, the answer is *it depends*. As pointed out in the classic OPTICS paper [6], different ϵ values allow us to

view the dataset from various granularities, leading to different clustering results. In Figure 2.5, given ϵ_1 (and some *MinPts* say 2), DBSCAN outputs 3 clusters. Given ϵ_2, on the other hand, DBSCAN outputs 2 clusters, which makes sense because at this distance, the two clusters on the right merge into one.

Now let us consider approximation. The dashed circles illustrate the radii obtained with ρ-approximation. For both ϵ_1 and ϵ_2, ρ-approximate DBSCAN will return exactly the same clusters, because these distances are *robustly chosen* by being insensitive to small perturbation. For ϵ_3, however, ρ-approximate DBSCAN may return only one cluster (i.e., all points in the same cluster), whereas exact DBSCAN will return only two (i.e., the same two clusters as ϵ_2). By looking at the figure closely, one can realize that this happens because the dashed circle of radius $(1+\rho)\epsilon_3$) "happens" to pass a point—namely point o—which falls outside the solid circle of radius ϵ_3. Intuitively, ϵ_3 is a poor parameter choice because it is *too* close to the distance between two clusters such that a small change to it will cause the clustering results to be altered.

Next we present a useful corollary of the sandwich theorem:

Corollary 2.1. *Let $\mathscr{C}_1, \mathscr{C}_2$, and $\mathscr{C}$ be as defined in Theorem 2.3. If a cluster C appears in both $\mathscr{C}_1$ and $\mathscr{C}_2$, then C must also be a cluster in $\mathscr{C}$.*

Proof. Suppose, on the contrary, that $\mathscr{C}$ does not contain C. By Theorem 2.3, (i) $\mathscr{C}$ must contain a cluster C' such that $C \subseteq C'$, and (ii) $\mathscr{C}_2$ must contain a cluster C'' such that $C' \subseteq C''$. This means $C \subseteq C''$. On the other hand, as $C \in \mathscr{C}_2$, it follows that, in $\mathscr{C}_2$, every core point in C belongs also to C''. This is impossible because a core point can belong to only one cluster. □

The corollary states that, even if some exact DBSCAN clusters have changed when ϵ increases by a factor of $1+\rho$ (i.e., ϵ is not robust), our ρ-approximation still captures all those clusters that do not change. For example, imagine that the points in Figure 2.5 are part of a larger dataset such that the clusters on the rest of the points are unaffected as ϵ_3 increases to $\epsilon_3(1+\rho)$. By Corollary 2.1, all those clusters are safely captured by ρ-approximate DBSCAN under ϵ_3.

2.3.3 Approximate Range Counting

Let us now take a break from DBSCAN, and turn our attention to a different problem, whose solution is vital to our ρ-approximate DBSCAN algorithm.

Let P still be a set of n points in $\mathbb{R}^d$ where d is a constant. Given any point $q \in \mathbb{R}^d$, a distance threshold $\epsilon > 0$ and an arbitrarily small constant $\rho > 0$, an *approximate range count query* returns an integer that is guaranteed to be between $|B(q, \epsilon) \cap P|$ and $|B(q, \epsilon(1 + \rho)) \cap P|$. For example, in Figure 2.4, given $q = o_1$, a query may return either 4 or 5.

Arya and Mount [10] developed a structure of $O(n)$ space that can be built in $O(n \log n)$ time, and answers any such query in $O(\log n)$ time. Next, we design an alternative structure with better performance in our context:

Lemma 2.5. *For any fixed ϵ and ρ, there is a structure of $O(n)$ space that can be built in $O(n)$ expected time, and answers any approximate range count query in $O(1)$ expected time.*

Structure. Our structure is a simple quadtree-like hierarchical grid partitioning of $\mathbb{R}^d$. First, impose a regular grid on $\mathbb{R}^d$ where each cell is a d-dimensional hyper-square with side length $\epsilon/\sqrt{d}$. For each *non-empty* cell c of the grid (i.e., c covers at least 1 point of P), divide it into 2^d cells of the same size. For each resulting *non-empty* cell c', divide it recursively in the same manner, until the side length of c' is at most $\epsilon\rho/\sqrt{d}$.

We use H to refer to the hierarchy thus obtained. We keep *only* the non-empty cells of H, and for each such cell c, record $cnt(c)$ which is the number of points in P covered by c. We will refer to a cell of H with side length $\epsilon/(2^i\sqrt{d})$ as a *level-i cell*. Clearly, H has only $h = \max\{1, 1+\lceil\log_2(1/\rho)\rceil\} = O(1)$ levels. If a level-$(i + 1)$ cell c' is inside a level-i cell c, we say that c' is a *child* of c, and c a *parent* of c'. A cell with no children is called a *leaf cell*.

Figure 2.6 illustrates the part of the first three levels of H for the dataset on the left. Note that empty cells are *not* stored.

Query. Given an approximate range count query with parameters q, ϵ, ρ, we compute its answer *ans* as follows. Initially, $ans = 0$. In general, given a non-empty level-i cell c, we distinguish three cases:

- If c is disjoint with $B(q, \epsilon)$, ignore it.

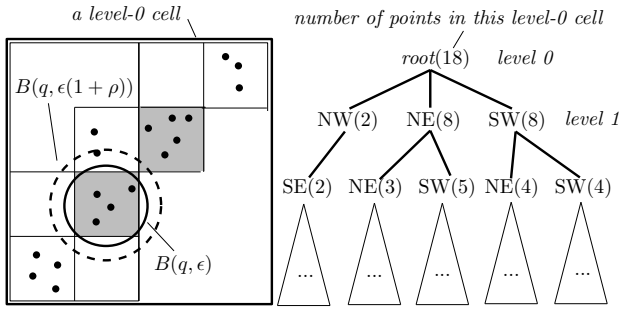


FIGURE 2.6: Approximate range counting

- If c is fully covered by $B(q, \epsilon(1+\rho))$, add $cnt(c)$ to ans.

- When neither of the above holds, check if c is a leaf cell in H. If not, process the child cells of c in the same manner. Otherwise (i.e., c is a leaf), add $cnt(c)$ to ans only if c intersects $B(q, \epsilon)$.

The algorithm starts from the level-0 non-empty cells that intersect with $B(q, \epsilon)$.

To illustrate, consider the query shown in Figure 2.6. The two gray cells correspond to nodes SW(5) and NE(4) at level 2. The subtree of *neither* of them is visited, but the reasons are different. For SW(5), its cell is disjoint with $B(q, \epsilon)$, so we ignore it (even though it intersects $B(q, \epsilon(1+\rho))$). For NE(4), its cell completely falls in $B(q, \epsilon(1+\rho))$, so we add its count 4 to the result (even though it is not covered by $B(q, \epsilon)$).

Correctness. The above algorithm has two guarantees. First, if a point $p \in P$ is inside $B(q, \epsilon)$, it is definitely counted in ans. Second, if p is outside $B(q, \epsilon(1+\rho))$, then it is definitely *not* counted in ans. These guarantees are easy to verify, utilizing the fact that if a leaf cell c intersects $B(p, \epsilon)$, then c must fall *completely* in $B(p, \epsilon(1+\rho))$ because any two points in a leaf cell are within distance $\epsilon\rho$. It thus follows that the ans returned is a legal answer.

Time Analysis. Remember that the hierarchy H has $O(1)$ levels. Since there are $O(n)$ non-empty cells at each level, the total space is $O(n)$. With hashing, it is easy to build the structure level by level in $O(n)$ expected time.

To analyze the running time of our query algorithm, observe that each cell c visited by our algorithm must satisfy one of the following conditions: (i) c is a level-0 cell, or (ii) the parent of c intersects the boundary of $B(q, \epsilon)$. For type-(i), the $O(1)$ level-0 cells intersecting $B(q, \epsilon)$ can be found

in $O(1)$ expected time using the coordinates of q. For type-(ii), it suffices to bound the number of cells intersecting the boundary of $B(q, \epsilon)$ because each such cell has $2^d = O(1)$ child nodes.

In general, a d-dimensional grid of cells with side length l has $O(1 + (\frac{\theta}{l})^{d-1})$ cells intersecting the boundary of a sphere with radius θ [10]. Combining this and the fact that a level-i cell has side length $\epsilon/(2^i \sqrt{d})$, we know that the total number of cells (of all levels) intersecting the boundary of $B(q, \epsilon)$ is bounded by:

$$\begin{aligned} \sum_{i=0}^{h-1} O\left(1 + \left(\frac{\epsilon}{\epsilon/(2^i \sqrt{d})}\right)^{d-1}\right) &= O\left((2^h)^{d-1}\right) \\ &= O\left(1 + (1/\rho)^{d-1}\right) \end{aligned}$$

which is a constant for any fixed ρ. This concludes the proof of Lemma 2.5.

2.3.4 Solving ρ-Approximate DBSCAN

We are now ready to solve the ρ-approximate DBSCAN problem by proving:

Theorem 2.4. *There is a ρ-approximate DBSCAN algorithm that terminates in $O(n)$ expected time, regardless of the value of ϵ, the constant approximation ratio ρ, and the fixed dimensionality d.*

Algorithm. Our ρ-approximate algorithm differs from the exact algorithm we proposed in Section 2.2.2 *only* in the definition and computation of the graph G. We re-define $G = (V, E)$ as follows:

- As before, each vertex in V is a core cell of the grid T (remember that the algorithm of Section 2.2.2 imposes a grid T on $\mathbb{R}^d$, where a cell is a core cell if it covers at least one core point).

- Given two different core cells c_1, c_2, whether E has an edge between c_1 and c_2 obeys the rules below:

  - yes, if there exist core points p_1, p_2 in c_1, c_2, respectively, such that $dist(p_1, p_2) \leq \epsilon$.
  - no, if no core point in c_1 is within distance $\epsilon(1 + \rho)$ from any core point in c_2.
  - *don't care*, in all the other cases.

To compute G, our algorithm starts by building, for each core cell c in T, a structure of Lemma 2.5 on the set of core points in c. To generate the edges of a core cell c_1, we examine each ϵ-neighbor cell c_2 of c_1 in turn. For every core point p in c_1, do an approximate range count query on the set of core points in c_2. If the query returns a non-zero answer, add an edge (c_1, c_2) to G. If all such p have been tried but still no edge has been added, we decide that there should be no edge between c_1 and c_2.

Correctness. Let C be an arbitrary cluster returned by our algorithm. We will show that C satisfies Definition 2.5.

Maximality. Let p be an arbitrary core point in C, and q be any point of P density-reachable from p. We will show that $q \in C$. Let us start by considering that q is a core point. By Definition 2.2, there is a sequence of core points $p_1, p_2, ..., p_t$ (for some integer $t \geq 2$) such that $p_1 = p$, $p_t = q$, and $dist(p_{i+1}, p_i) \leq \epsilon$ for each $i \in [1, t-1]$. Denote by c_i the cell of T covering p_i. By the way G is defined, there must be an edge between c_i and c_{i+1}, for each $i \in [1, t-1]$. It thus follows that c_1 and c_t must be in the same connected component of G; therefore, p and q must be in the same cluster. The correctness of the other scenario where q is a non-core point is trivially guaranteed by the way that non-core points are assigned to clusters.

ρ-Approximate Connectivity. Let p be an arbitrary core point in C. For *any* point $q \in C$, we will show that q is ρ-approximate density-reachable from p. Again, we consider first that q is a core point. Let c_p and c_q be the cells of T covering p and q, respectively. Since c_p and c_q are in the same connected component of G, there is a path $c_1, c_2, ..., c_t$ in G (for some integer $t \geq 2$) such that $c_1 = c_p$ and $c_t = c_q$. Recall that any two points in the same cell are within distance ϵ. Combining this fact with how the edges of G are defined, we know that there is a sequence of core points $p_1, p_2, ..., p_{t'}$ (for some integer $t' \geq 2$) such that $p_1 = p$, $p_{t'} = q$, and $dist(p_{i+1}, p_i) \leq \epsilon(1+\rho)$ for each $i \in [1, t'-1]$. Therefore, q is ρ-approximate density-reachable from p. The correctness of the other scenario where q is a non-core point is again trivial.

Time Analysis. It takes $O(n)$ expected time to construct the structure of Lemma 2.5 for all cells. The time of computing G is proportional to the number of approximate range count queries issued. For each core point of a cell c_1, we issue $O(1)$ queries in total (one for each ϵ-neighbor cell of c_2). Hence, the total number of queries is $O(n)$. The rest of the ρ-approximate algorithm runs in $O(n)$ expected time, following the same analysis in [31]. This completes the proof of Theorem 2.4. It is

worth mentioning that, intuitively, the efficiency improvement of our approximate algorithm (over the exact algorithm in Section 2.2.2) owes to the fact that we settle for an imprecise solution to the BCP problem by using Lemma 2.5.

Remark. It should be noted that the hidden constant in $O(n)$ is at the order of $(1/\rho)^{d-1}$; see the proof of Lemma 2.5. As this is exponential to the dimensionality d, our techniques are suitable only when d is low. Our experiments considered dimensionalities up to 7.

2.4 New 2D Exact Algorithms

This section gives two new algorithms for solving the (exact) DBSCAN problem in $\mathbb{R}^2$. These algorithms are based on different ideas, and are interesting in their own ways. The first one (Section 2.4.1) is conceptually simple, and establishes a close connection between DBSCAN and Delaunay graphs. The second one (Section 2.4.2) manages to identify coordinate sorting as the most expensive component in DBSCAN computation.

2.4.1 DBSCAN from a Delaunay Graph

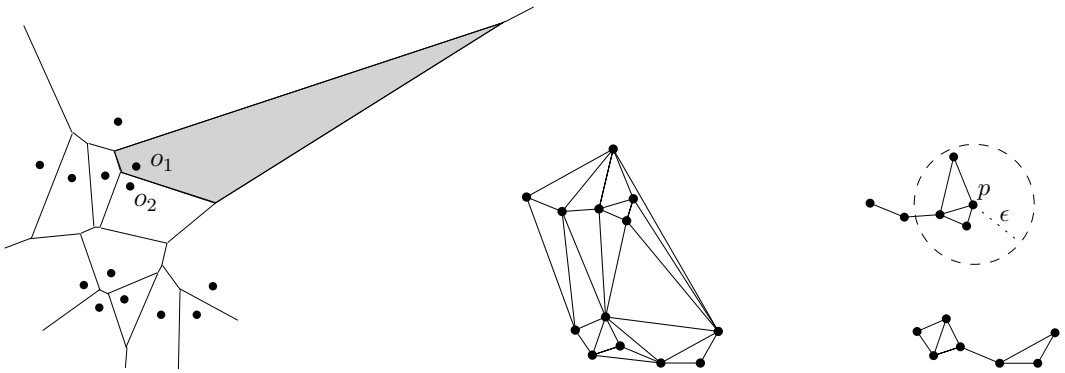
Recall from Section 2.1.2 that Gunawan's algorithm runs in three steps:

1. Label each point of the input set P as either core or non-core.
2. Partition the set P_{core} of core points into clusters.
3. Assign each non-core point to the appropriate cluster(s).

Step 2 is the performance bottleneck. Next, we describe a new method to accomplish this step.

Algorithm for Step 2. The Delaunay graph of P_{core} can be regarded as the dual of the Voronoi diagram of P_{core}. The latter is a subdivision of the data space $\mathbb{R}^2$ into $|P_{core}|$ convex polygons, each of which corresponds to a distinct $p \in P_{core}$, and is called the *Voronoi cell* of p, containing every location in $\mathbb{R}^2$ that finds p as its Euclidean nearest neighbor in P_{core}. The *Delaunay graph* of P_{core} is a graph $G_{dln} = (V_{dln}, E_{dln})$ defined as follows:

- $V_{dln} = P_{core}$, that is, every core point is a vertex of G_{dln}.



(a) Voronoi diagram (b) Delaunay graph (c) Remainder graph after edge removal

Figure 2.7: Illustration of our Step-2 Algorithm in Section 2.4.1

- E_{dln} contains an edge between two core points p_1, p_2 if and only if their Voronoi cells are adjacent (i.e., sharing a common boundary segment).

G_{dln}, in general, *always* has only a linear number of edges, i.e., $|E_{dln}| = O(|P_{core}|)$.

Figure 2.7a demonstrates the Voronoi diagram defined by the set of black points shown. The shaded polygon is the Voronoi cell of o_1; the Voronoi cells of o_1 and o_2 are adjacent. The corresponding Delaunay graph is given in Figure 2.7b.

Provided that G_{dln} is already available, we perform Step 2 using a simple strategy:

(2. 1) Remove all the edges (p_1, p_2) in E_{dln} such that $dist(p_1, p_2) > \epsilon$. Let us refer to the resulting graph as the *remainder graph*.

(2. 2) Compute the connected components of the remainder graph.

(2. 3) Put the core points in each connected component into a separate cluster.

Continuing the example in Figure 2.7b, Figure 2.7c illustrates the remainder graph after the edge removal in Step 2.1 (the radius of the circle centered at point p indicates the value of ϵ). There are two connected components in the remainder graph; the core points in each connected component constitute a cluster.

In general, the Delaunay graph of x 2D points can be computed in $O(x \log x)$ time [20]. Clearly, Steps 2.1-2.3 require only $O(|P_{core}|) = O(n)$ time. Therefore, our Step 2 algorithm finishes in $O(n \log n)$ time overall.

Correctness of the Algorithm. It remains to explain why the above simple strategy correctly clusters the core points. Remember that a core point p ought to be placed in the same cluster as another core point q if and only if there is a sequence of core points $p_1, p_2, ..., p_t$ (for some $t \geq 2$) such that:

- $p_1 = p$ and $p_t = q$
- $dist(p_i, p_{i+1}) \leq \epsilon$ for each $i \in [1, t-1]$.

We now prove:

Lemma 2.6. *Two core points p, q belong to the same cluster if and only if our Step-2 algorithm declares so.*

Proof. *The If Direction.* This direction is straightforward. Our algorithm declares p, q to be in the same cluster only if they appear in the same connected component of the remainder graph obtained at Step 2.1. This, in turn, suggests that the connected component has a path starting from p and ending at q satisfying the aforementioned requirement.

The Only-If Direction. Let p, q be a pair of core points that should be placed in the same cluster. Next, we will prove that our Step-2 algorithm definitely puts them in the same connected component of the remainder graph.

We will first establish this fact by assuming $dist(p, q) \leq \epsilon$. Consider the line segment pq. Since Voronoi cells are convex polygons, in moving on segment pq from p to q, we must be traveling through the Voronoi cells of a sequence of distinct core points—let them be $p_1, p_2, ..., p_t$ for some $t \geq 2$, where $p_1 = p$ and $p_t = q$. Our goal is to show that $dist(p_i, p_{i+1}) \leq \epsilon$ for *all* $i \in [1, t-1]$. This will indicate that the remainder graph must contain an edge between each pair of (p_i, p_{i+1}) for all $i \in [1, t-1]$, implying that all of $p_1 = p, p_2, ..., p_t = q$ must be in the same connected component at Step 2.3.

We now prove $dist(p_i, p_{i+1}) \leq \epsilon$ for an arbitrary $i \in [1, t-1]$. Let $\tilde{p}_i$ (for $i \in [1, t-1]$) be the intersection between pq and the common boundary of the Voronoi cells of p_i and p_{i+1}. Figure 2.8 illustrates the definition with an example where $t = 7$. We will apply triangle inequality a number of times to arrive at our target conclusion. Let us start with:

$$dist(p_i, p_{i+1}) \leq dist(p_i, \tilde{p}_i) + dist(p_{i+1}, \tilde{p}_i). \tag{2.2}$$

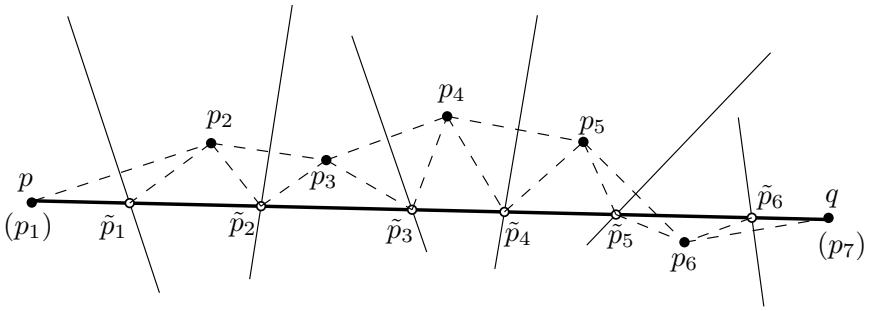


Figure 2.8: Correctness proof of our Step-2 algorithm

Regarding $dist(p_i, \tilde{p}_i)$, we have:

$$\begin{aligned}
dist(p_i, \tilde{p}_i) &\leq dist(p_i, \tilde{p}_{i-1}) + dist(\tilde{p}_{i-1}, \tilde{p}_i) \\
&= dist(p_{i-1}, \tilde{p}_{i-1}) + dist(\tilde{p}_{i-1}, \tilde{p}_i) \\
&\quad \text{(note: } dist(p_{i-1}, \tilde{p}_{i-1}) = dist(p_i, \tilde{p}_{i-1}) \text{ as } \tilde{p}_{i-1} \text{ is on the} \\
&\quad \text{perpendicular bisector of segment } p_i p_{i-1}) \\
&\leq dist(p_{i-1}, \tilde{p}_{i-2}) + dist(\tilde{p}_{i-2}, \tilde{p}_{i-1}) + dist(\tilde{p}_{i-1}, \tilde{p}_i) \\
&\quad \text{(triangle inequality)} \\
&= dist(p_{i-1}, \tilde{p}_{i-2}) + dist(\tilde{p}_{i-2}, \tilde{p}_i) \\
&\ldots \\
&\leq dist(p_2, \tilde{p}_1) + dist(\tilde{p}_1, \tilde{p}_i) \\
&= dist(p_1, \tilde{p}_1) + dist(\tilde{p}_1, \tilde{p}_i) \\
&= dist(p_1, \tilde{p}_i). \qquad (2.3)
\end{aligned}$$

Following a symmetric derivation, we have:

$$dist(p_{i+1}, \tilde{p}_i) \leq dist(\tilde{p}_i, p_t). \qquad (2.4)$$

The combination of (2.2)-(2.4) gives:

$$\begin{aligned}
dist(p_i, p_{i+1}) &\leq dist(p_1, \tilde{p}_i) + dist(\tilde{p}_i, p_t) \\
&= dist(p_1, p_t) \leq \epsilon
\end{aligned}$$

as claimed.

We now get rid of the assumption that $dist(p, q) \leq \epsilon$. This is fairly easy. By the given fact that p and q should be placed in the same cluster, we know that there is a path $p_1 = p, p_2, p_3, ..., p_t = q$ (where $t \geq 2$) such that $dist(p_i, p_{i+1}) \leq \epsilon$ for each $i \in [1, t-1]$. By our earlier argument, each pair of (p_i, p_{i+1}) must be in the same connected component of our remainder graph. Consequently, all of $p_1, p_2, ..., p_t$ are in the same connected component. This completes the proof. □

Remark. The concepts of Voronoi Diagram and Delaunay graph can both be extended to arbitrary dimensionality $d \geq 3$. Our Step-2 algorithm also works for any $d \geq 3$. While this may be interesting from a geometric point of view, it is not from an algorithmic perspective. Even at $d = 3$, a Delaunay graph on n points can have $\Omega(n^2)$ edges, necessitating $\Omega(n^2)$ time for its computation. In contrast, in Section 2.2.2, we already showed that the exact DBSCAN problem can be solved in $o(n^2)$ time for any constant dimensionality d.

2.4.2 Separation of Sorting from DBSCAN

We say that the 2D input set P is *bi-dimensionally sorted* if the points therein are given in two sorted lists:

- P_x, where the points are sorted by x-dimension;
- P_y, where the points are sorted by y-dimension.

This subsection will establish the last main result of this article:

Theorem 2.5. *If P has been bi-dimensionally sorted, the exact DBSCAN problem (in 2D space) can be solved in $O(n)$ worst-case time.*

The theorem reveals that coordinate sorting is actually the "hardest" part of the 2D DBSCAN problem! This means that we can even beat the $\Omega(n \log n)$ time bound for this problem in scenarios where sorting can be done fast. The corollaries below state two such scenarios:

Corollary 2.2. *If each dimension has an integer domain of size at most n^c for an arbitrary positive constant c, the 2D DBSCAN problem can be solved in $O(n)$ worst-case time (even if P is not bi-dimensionally sorted).*

Proof. Kirkpatrick and Reisch [40] showed that n integers drawn from a domain of size n^c (regardless of the constant $c \geq 1$) can be sorted in $O(n)$ time, by generalizing the idea of radix sort. Using their algorithm, P can be made bi-dimensionally sorted in $O(n)$ time. Then, the corollary follows from Theorem 2.5. □

The above corollary is important because, in real applications, (i) coordinates are always discrete (after digitalization), and (ii) when n is large (e.g., 10^6), the domain size of each dimension rarely exceeds n^2. The 2D DBSCAN problem can be settled in linear time in all such applications.

Corollary 2.3. *If each dimension has an integer domain, the 2D DBSCAN problem can be solved in $O(n \log\log n)$ worst-case time or $O(n\sqrt{\log\log n})$ expected time (even if P is not bi-dimensionally sorted).*

Proof. Andersson et al. [5] gave a deterministic algorithm to sort n integers in $O(n \log\log n)$ worst-case time. Han and Thorup [34] gave a randomized algorithm to do so in $O(n\sqrt{\log\log n})$ expected time. Plugging these results into Theorem 2.5 yields the corollary. □

Next, we provide the details of our algorithm for Theorem 2.5. The general framework is still the 3-step process as shown in Section 2.4.1, but we will develop new methods to implement Steps 1 and 2 in linear time, utilizing the property that P is bi-dimensionally sorted. Step 3 is carried out in the same manner as in the Gunawan's algorithm (Section 2.1.2), which demands only $O(n)$ time.

Step 1

Recall that, for this step, Gunawan's algorithm places an arbitrary grid T (where each cell is a square with side length $\epsilon/\sqrt{2}$) in $\mathbb{R}^2$, and then proceeds as follows:

(1. 1) For each non-empty cell c of T, compute the set $P(c)$ of points in P that are covered by c.

(1. 2) For each non-empty cell c of T, identify all of its non-empty ϵ-neighbor cells c' (i.e., the minimum distance between c and c' is less than ϵ).

(1. 3) Perform a labeling process to determine whether each point in P is a core or non-core point.

Our approach differs from Gunawan's in Steps 1.1 and 1.2 (his solution to Step 1.3 takes only $O(n)$ time, and is thus sufficient for our purposes). Before continuing, note that Steps 1.1 and 1.2 can be done easily with hashing using $O(n)$ *expected* time, but our goal is to attain the same time complexity in the worst case.

Step 1.1. We say that a column of T (a *column* contains all the cells of T sharing the same projection on the x-dimension) is *non-empty* if it has at least one non-empty cell. We label the leftmost non-empty column as 1, and the 2nd leftmost non-empty column as 2, and so on. By scanning P_x once in ascending order of x-coordinate, we determine, for each point $p \in P$, the label of the non-empty column that contains p; the time required is $O(n)$.

Suppose that there are n_{col} non-empty columns. Next, for each $i \in [1, n_{col}]$, we generate a sorted list $P_y[i]$ that arranges, in ascending of y-coordinate, the points of P covered by (non-empty) column i. In other words, we aim to "distribute" P_y into n_{col} sorted lists, one for each non-empty column. This can be done in $O(n)$ time as follows. First, initialize all the n_{col} lists to be empty. Then, scan P_y in ascending order of y-coordinate; for each point p seen, append it to $P_y[i]$ where i is the label of the column containing p. The point ordering in P_y ensures that each $P_y[i]$ thus created is sorted on y-dimension.

Finally, for each $i \in [1, n_{col}]$, we generate the target set $P(c)$ for every non-empty cell c in column i, by simply scanning $P_y[i]$ once in order to divide it into sub-sequences, each of which includes all the points in a distinct cell (sorted by y-coordinate). The overall cost of Step 1.1 is therefore $O(n)$. As a side product, for every $i \in [1, n_{col}]$, we have also obtained a list L_i of all the non-empty cells in column i, sorted in bottom-up order.

Step 1.2. We do so by processing each non-empty column in turn. First, observe that if a cell is in column $i \in [1, n_{col}]$, all of its ϵ-neighbor cells must appear in columns $i-2, i-1, i, i+1$, and $i+2$ (see Figure 2.2c). Motivated by this, for each $j \in \{i-2, i-1, i, i+1, i+2\} \cap [1, n_{col}]$, we scan synchronously the cells of L_i and L_j in bottom-up order (if two cells are at the same row, break the tie by scanning first the one from L_i). When a cell $c \in L_i$ is encountered, we pinpoint the last cell $c_0 \in L_j$ that was scanned. Define:

- c_{-1} as the cell in L_j immediately *before* c_0;

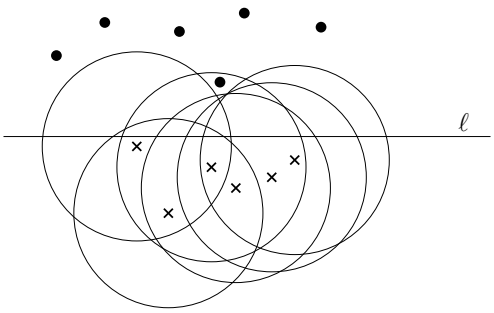


Figure 2.9: USEC with line separation

- c_1 as the cell in L_j immediately *after* c_0;
- c_2 as the cell in L_j immediately after c_1;
- c_3 as the cell in L_j immediately after c_2;

The 5 cells[2] $c_{-1}, c_0, ..., c_3$ are the only ones that can be ϵ-neighbors of c in L_j. Checking which of them are indeed ϵ-neighbors of c takes $O(1)$ time. Hence, the synchronous scan of L_i and L_j costs $O(|L_i| + |L_j|)$ time. The total cost of Step 1.2 is, therefore, $O(n)$, noticing that each L_i ($i \in [1, n_{col}]$) will be scanned at most 5 times.

Remark. By slightly extending the above algorithm, for each non-empty cell c, we can store the points of $P(c)$ in two sorted lists:

- $P_x(c)$, where the points of $P(c)$ are sorted on x-dimension;
- $P_y(c)$, where the points are sorted on y-dimension.

To achieve this purpose, first observe that, at the end of Step 1.1, the sub-sequence obtained for each non-empty cell c is precisely $P_y(c)$. This allows us to know, for each point $p \in P$, the id of the non-empty cell covering it. After this, the $P_x(c)$ of all non-empty cells c can be obtained with just another scan of P_x: for each point p seen in P_x, append it to $P_x(c)$, where c is the cell containing p. The point ordering in P_x ensures that each $P_x(c)$ is sorted by x-coordinate, as desired. The additional time required is still $O(n)$.

[2] If $c_0 = \emptyset$ (namely, no cell in L_j has been scanned), set c_1, c_2, c_3 to the lowest 3 cells in L_j.

Step 2

For this step, Gunawan's algorithm generates a graph $G = (V, E)$ where each core cell in T corresponds to a distinct vertex in V. Between core cells (a.k.a., vertices) c_1 and c_2, an edge exists in E if and only if there is a core point p_1 in c_1 and a core point p_2 in c_2 such that $dist(p_1, p_2) \leq \epsilon$. Once G is available, Step 2 is accomplished in $O(n)$ time by computing the connected components of G. The performance bottleneck lies in the creation of G, to which Gunawan's solution takes $O(n \log n)$ time. We develop a new algorithm below that fulfills the purpose in $O(n)$ time.

USEC with Line Separation. Let us introduce a special variant of the USEC problem defined in Section 2.1.3, which stands at the core of our $O(n)$-time algorithm. Recall that in the 2D USEC problem, we are given a set S_{ball} of discs with the same radius ϵ, and a set S_{pt} of points, all in the data space $\mathbb{R}^2$. The objective is to determine whether any point in S_{pt} is covered by any disc in S_{ball}. In our special variant, there are two extra constraints:

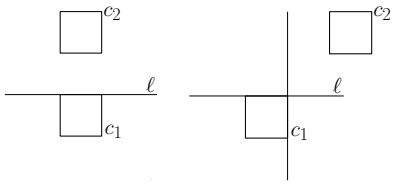
- There is a horizontal line ℓ such that (i) all the centers of the discs in S_{ball} are on or below ℓ, and (ii) all the points in S_{pt} are on or above ℓ.
- The centers of the discs in S_{ball} have been sorted by x-dimension, and so are the points in S_{pt}.

Figure 2.9 illustrates an instance of the above *USEC with line separation problem* (where crosses indicate disc centers). The answer to this instance is *yes* (i.e., a point falls in a disc).

Lemma 2.7. *The USEC with line separation problem can be settled in linear time, namely, with cost* $O(|S_{pt}| + |S_{ball}|)$.

An algorithm for achieving the above lemma is implied in [17]. However, the description in [17] is rather brief, and does not provide the full details. In the appendix, we reconstruct their algorithm, and prove its correctness (such a proof was missing in [17]). Nonetheless, we believe that credits on the lemma should be attributed to [17]. The reader may also see [21] for another account of the algorithm.

Generating G in $O(n)$ Time. We now return to our endeavor of finding an $O(n)$ time algorithm to generate G. The vertices of G, which are precisely the core cells, can obviously be collected in $O(n)$ time (there are at most n core cells). It remains to discuss the creation of the edges in G.



(a) Case 1 (b) Case 2

Figure 2.10: Deciding the existence of an edge by USEC with line separation

Now, focus on any two core cells c_1 and c_2 that are ϵ-neighbors of each other. Our mission is to determine whether there should be an edge between them. It turns out that this requires solving at most two instances of USEC with line separation. Following our earlier terminology, let $P(c_1)$ be the set of points of P that fall in c_1. Recall that we have already obtained two sorted lists of $P(c_1)$, that is, $P_x(c_1)$ and $P_y(c_1)$ that are sorted by x- and y-dimension, respectively. Define $P(c_2), P_x(c_2)$, and $P_y(c_2)$ similarly for c_2. Depending on the relative positions of c_1 and c_2, we proceed differently in the following two cases (which essentially have represented all possible cases by symmetry):

- Case 1: c_2 is in the same column as c_1, and is above c_1, as in Figure 2.10a. Imagine placing a disc centered at each point in $P(c_1)$. All these discs constitute S_{ball}. Set S_{pt} directly to $P(c_2)$. Together with the horizontal line ℓ shown, this defines an instance of USEC with line separation. There is an edge between c_1, c_2 if and only if the instance has a *yes* answer.

- Case 2: c_2 is to the northeast of c_1, as in Figure 2.10b. Define S_{ball} and S_{pt} in the same manner as before. They define an instance of USEC with line separation based on ℓ. There is an edge between c_1, c_2 if and only if the instance has a *yes* answer.

It is immediately clear from Lemma 2.7 that we can make the correct decision about the edge existence between c_1, c_2 using $O(|P(c_1)| + |P(c_2)|)$ time. Therefore, the total cost of generating all the edges in G is bounded by:

$$\sum_{\text{core cell } c_1} \left(\sum_{\epsilon\text{-neighbor } c_2 \text{ of } c_1} O(|P(c_1)| + |P(c_2)|) \right) = \sum_{\text{core cell } c_1} O(|P(c_1)|) = O(n)$$

where the first equality used the fact that each core cell has $O(1)$ ϵ-neighbors, and hence, can participate in only $O(1)$ instances of USEC with line separation.

2.5 Discussion on Practical Efficiency

Besides our theoretical findings, we have developed a software prototype based on the proposed algorithms. Our implementation has evolved beyond that of [29] by incorporating new heuristics. Next, we will explain the most crucial heuristics adopted which apply to all of our algorithms (since they are based on the same grid-based framework). Then, we will discuss when the original DBSCAN algorithm of [27] is or is not expected to work well in practice. Finally, a qualitative comparison of the precise and ρ-approximate DBSCAN algorithms will be presented.

Heuristics. The three most effective heuristics in our implementation can be summarized as follows:

- Recall that our ρ-approximate algorithm imposes a grid T on $\mathbb{R}^d$. We manage all the non-empty cells in a (main memory) R-tree which is constructed by bulkloading. This R-tree allows us to efficiently find, for any cell c, all its ϵ-neighbor non-empty cells c'. Recall that such an operation is useful in a number of scenarios: (i) in the labeling process when a point p falls in a cell covering less than *MinPts* points, (ii) in deciding the edges of c in G, and (iii) assigning a non-core point in c to appropriate clusters.

- For every non-empty cell c, we store all its ϵ-neighbor non-empty cells in a list, after they have been computed for the first time. As each list has length $O(1)$, the total space of all the lists is $O(n)$ (recall that at most n non-empty cells exist). The lists allow us to avoid re-computing ϵ-neighbor non-empty cells of c.

- Theoretically speaking, we achieve $O(n)$ expected time by first generating the edges of G and then computing its connected components (CC). In reality, it is faster not to produce the edges, but instead, maintain the CCs using a union-find structure [66].

  Specifically, whenever an edge between non-empty cells c and c' is found, we perform a "union" operation using c and c' on the structure. After all the edges have been processed like this, the final CCs can be easily determined by issuing a "find" operation on every non-empty cell. In theory, this approach entails $O(n \cdot \alpha(n))$ time, where $\alpha(n)$ is the inverse of the Ackermann which is extremely slow growing such that $\alpha(n)$ is very small for all practical n.

  An advantage of this approach is that, it avoids a large amount of edge detection that was needed

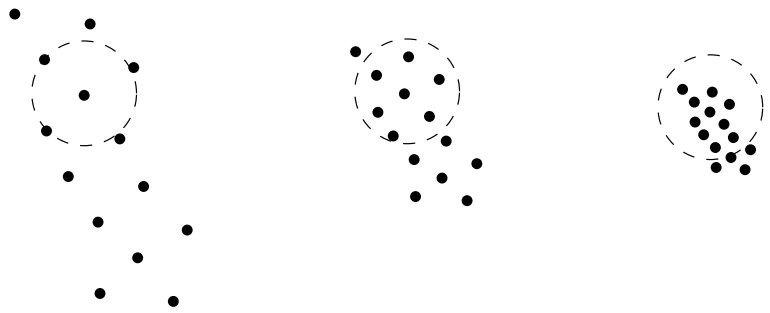


Figure 2.11: A small ϵ for the left cluster is large for the other two clusters

in [29]. Before, such detection was performed for each pair of non-empty cells c and c' that were ϵ-neighbors of each other. Now, we can safely skip the detection if these cells are already found to be in the same CC.

Characteristics of the KDD'96 Algorithm. As mentioned in Section 1.1, the running time of the algorithm in [27] is determined by the total cost of n region queries, each of which retrieves $B(p, \epsilon)$ for each $p \in P$. Our hardness result in Theorem 2.1 implies that, even if *each* $B(p, \epsilon)$ returns just p itself, the cost of all n queries must still sum up to $\Omega(n^{4/3})$ for a hard dataset.

As reasonably argued by [27], on practical data, the cost of a region query $B(p, \epsilon)$ depends on how many points are in $B(p, \epsilon)$. The KDD'96 algorithm may have acceptable efficiency when ϵ is small such that the total number of points returned by all the region queries is near linear.

Such a value of ϵ, however, may not exist when the clusters have varying densities. Consider the example in Figure 2.11 where there are three clusters. Suppose that $MinPts = 4$. To discover the sparsest cluster on the left, ϵ needs to be at least the radius of the circles illustrated. For each point p from the right (i.e., the densest) cluster, however, the $B(p, \epsilon)$ under such an ϵ covers a big fraction of the cluster. On this dataset, therefore, the algorithm of [27] either does not discover all three clusters, or must do so with expensive cost.

A Comparison. The preceding discussion suggests that the relative superiority between the KDD'96 algorithm and our proposed ρ-approximate algorithm depends primarily on two factors: (i) whether the cluster densities are similar or varying, and (ii) whether the value of ϵ is small or large. For a dataset with varying-density clusters, our algorithm is expected to perform better because, as explained, a good ϵ that finds all clusters must be relatively large for the dense clusters, forcing the

KDD'96 algorithm to entail high cost on those clusters.

For a dataset with similar-density clusters, the proposed algorithm is also expected to be the winner unless ϵ is sufficiently small. In fact, our empirical experience indicates a pattern: when the ρ-approximate algorithm is slower, the grid T it imposes on $\mathbb{R}^d$ has $\Omega(n)$ non-empty cells—more specifically, we observe that the cutoff threshold is roughly $n/\sqrt{2}$ cells, regardless of d. This makes sense because, in such a case, most non-empty cells have very few points (e.g., one or two), thus the extra overhead of creating and processing the grid no longer pays off.

The above observations will be verified in the next section.

2.6 Experiments

The philosophy of the following experiments differs from that in the short version [29]. Specifically, in [29], we treated DBSCAN clustering as a computer science problem, and aimed to demonstrate the quadratic nature of the previous DBSCAN algorithms for $d \geq 3$. In this work, we regard DBSCAN as an *application*, and will focus on parameter values that are more important in practice.

All the experiments were run on a machine equipped with 3.4GHz CPU and 16 GB memory. The operating system was Linux (Ubuntu 14.04). All the programs were coded in C++, and compiled using g++ with -o3 turned on.

Section 2.6.1 describes the datasets in our experimentation, after which Section 2.6.2 explores their clusters to understand what are the suitable parameter values. The evaluation of the proposed techniques will then proceed in three parts. First, Section 2.6.3 assesses the clustering precision of ρ-approximate DBSCAN. Section 2.6.4 demonstrates the efficiency gain achieved by our approximation algorithm compared to exact DBSCAN in dimensionality $d \geq 3$. Finally, Section 2.6.5 examines the performance of exact DBSCAN algorithms for $d = 2$.

2.6.1 Datasets

In all datasets, the underlying data space had a normalized integer domain of $[0, 10^5]$ for every dimension. We deployed both synthetic and real datasets whose details are explained next.

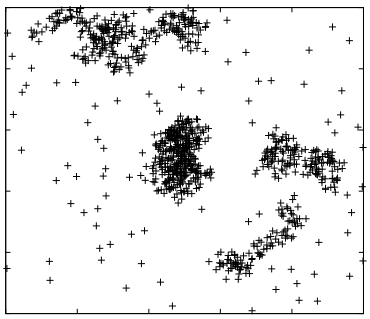


FIGURE 2.12: A 2D seed spreader dataset

Synthetic: Seed Spreader (SS). A synthetic dataset was generated in a "random walk with restart" fashion. First, fix the dimensionality d, take the target cardinality n, a *restart probability* $\rho_{restart}$, and a *noise percentage* ρ_{noise}. Then, we simulate a *seed spreader* that moves about in the space, and spits out data points around its current location. The spreader carries a *local counter* such that whenever the counter reaches 0, the spreader moves a distance of r_{shift} towards a random direction, after which the counter is reset to c_{reset}. The spreader works in *steps*. In each step, (i) with probability $\rho_{restart}$, the spreader *restarts*, by jumping to a random location in the data space, and resetting its counter to c_{reset}; (ii) no matter if a restart has happened, the spreader produces a point uniformly at random in the ball centered at its current location with radius $r_{vincinity}$, after which the local counter decreases by 1. Intuitively, every time a restart happens, the spreader begins to generate a new cluster. In the first step, a restart is forced so as to put the spreader at a random location. We repeat in total $n(1 - \rho_{noise})$ steps, which generate the same number of points. Finally, we add $n \cdot \rho_{noise}$ noise points, each of which is uniformly distributed in the whole space.

Figure 2.12 shows a small 2D dataset which was generated with $n = 1000$ and 4 restarts; the dataset will be used for visualization. The other experiments used larger datasets created with $c_{reset} = 100$, $\rho_{noise} = 1/10^4$, $\rho_{restart} = 10/(n(1 - \rho_{noise}))$. In expectation, around 10 restarts occur in the generation of a dataset. The values of $r_{vincinity}$ and r_{shift} were set in two different ways to produce clusters with either similar or varying densities:

- *Similar-density* dataset: Namely, the clusters have roughly the same density. Such a dataset was obtained by fixing $r_{vincinity} = 100$ and $r_{shift} = 50d$.

TABLE 2.1: Parameter values (defaults are in bold)

parameter	**values**
n (synthetic)	100k, 0.5m, 1m, **2m**, 5m, 10m
d (synthetic)	2, 3, **5**, 7
ϵ	from 100 (or 40 for $d = 2$) to 5000 (each dataset has its own default)
MinPts	10, 20, 40, 60, 100 (each dataset has its own default)
ρ	**0.001**, 0.01, 0.02, ..., 0.1

- *Varying-density* dataset: Namely, the clusters have different densities. Such a dataset was obtained by setting $r_{vincinity} = 100 \cdot ((i \mod 10) + 1)$ and $r_{shift} = r_{vincinity} \cdot d/2$, where i equals the number of restarts that have taken place (at the beginning $i = 0$). Note that the "modulo 10" ensures that there are at most 10 different cluster densities.

The value of n ranged from 100k to 10 million, while d from 2 to 7; see Table 2.1. Hereafter, by *SS-simden-dD*, we refer to a d-dimensional similar-density dataset (the default cardinality is 2m), while by *SS-varden-dD*, we refer to a d-dimensional varying-density dataset (same default on cardinality).

Real. Three real datasets were employed in our experimentation:

- The first one, *PAMAP2*, is a 4-dimensional dataset with cardinality 3,850,505, obtained by taking the first 4 principle components of a PCA on the PAMAP2 database [58] from the UCI machine learning archive [13].

- The second one, *Farm*, is a 5-dimensional dataset with cardinality 3,627,086, which contains the VZ-features [69] of a satellite image of a farm in Saudi Arabia[3]. It is worth noting that VZ-feature clustering is a common approach to perform color segmentation of an image [69].

- The third one, *Household*, is a 7-dimensional dataset with cardinality 2,049,280, which includes all the attributes of the Household database again from the UCI archive [13] except the temporal columns *date* and *time*. Points in the original database with missing coordinates were removed.

[3] *http://www.satimagingcorp.com/gallery/ikonos/ikonos-tadco-farms-saudi-arabia*

2.6.2 Characteristics of the Datasets

This subsection aims to study the clusters in each dataset under different parameters, and thereby, decide the values of *MinPts* and ϵ suitable for the subsequent efficiency experiments.

Clustering Validation Index. We resorted to a method called *clustering validation* (CV) [52] whose objective is to quantify the quality of clustering using a real value. In general, a set of good density-based clusters should have two properties: first, the points in a cluster should be "tightly" connected; second, any two points belonging to different clusters should have a large distance. To quantify the first property for a cluster C, we compute a *Euclidean minimum spanning tree* (EMST) on the set of core points in C, and then, define $DSC(C)$ as the maximum weight of the edges in the EMST. "DSC" stands for *density sparseness of a cluster*, a term used by [52]. Intuitively, the EMST is a "backbone" of C such that if C is tightly connected, $DSC(C)$ ought to be small. Note that the border points of C are excluded because they are not required to have a dense vicinity. To quantify the second property, define $DSPC(C_i, C_j)$ between two clusters C_i and C_j as

$$\min\{dist(p_1, p_2) \mid p_1 \in C_1 \text{ and } p_2 \in C_2 \text{ are core points}\}$$

where "DSPC" stands for *density separation for a pair of clusters* [52]. Let $\mathcal{C} = \{C_1, C_2, ..., C_t\}$ (where $t \geq 2$) be a set of clusters returned by an algorithm. For each C_i, we define (following [52]):

$$V_{\mathcal{C}}(C_i) \quad = \quad \frac{\left(\min_{1 \leq j \leq t, j \neq i} DSPC(C_i, C_j)\right) - DSC(C_i)}{\max\left\{DSC(C_i), \min_{1 \leq j \leq t, j \neq i} DSPC(C_i, C_j)\right\}}$$

Then, the *CV index* of $\mathcal{C}$ is calculated as in [52]:

$$\sum_{i=1}^{t} \frac{|C_i|}{n} V_{\mathcal{C}}(C_i)$$

where n is the size of the dataset. A higher validity index indicates better quality of $\mathcal{C}$.

Moulavi et al. [52] computed $DSC(C_i)$ and $DSPC(C_i, C_j)$ differently, but their approach requires $O(n^2)$ time which is intolerably long for the values of n considered here. Our proposition follows the same rationale, admits faster implementation (EMST is well studied [3, 11]), and worked well in our experiments as shown below.

Influence of *MinPts* and ϵ on DBSCAN Clusters. For each dataset, we examined the quality of its clusters under different combinations of *MinPts* and ϵ. For *MinPts*, we inspected values 10 and 100,

TABLE 2.2: Cluster quality under different ($MinPts, \epsilon$): SS similar density

	MinPts = 10			*MinPts* = 100		
ϵ	CV Index	# clusters	# noise pts	CV Index	# clusters	# noise pts
40	0.978	10	325	0.555	230	309224
60	0.994	9	197	0.577	72	33489
80	0.994	9	197	0.994	9	506
100	0.994	9	197	0.994	9	197
200	0.994	9	197	0.994	9	197

(a) *SS-simden-2D*

	MinPts = 10			*MinPts* = 100		
ϵ	CV Index	# clusters	# noise pts	CV Index	# clusters	# noise pts
100	0.996	14	200	0.205	240	467
200	0.996	14	200	0.996	14	200
400	0.996	14	200	0.996	14	200
800	0.996	14	200	0.996	14	200
1000	0.996	14	200	0.996	14	200

(b) *SS-simden-3D*

	MinPts = 10			*MinPts* = 100		
ϵ	CV Index	# clusters	# noise pts	CV Index	# clusters	# noise pts
100	0.102	4721	219	0.583	19057	632
200	0.996	13	200	0.996	13	241
400	0.996	13	200	0.996	13	200
800	0.996	13	200	0.996	13	200
1000	0.996	13	200	0.996	13	200

(c) *SS-simden-5D*

	MinPts = 10			*MinPts* = 100		
ϵ	CV Index	# clusters	# noise pts	CV Index	# clusters	# noise pts
100	0.588	19824	215	0.705	19822	1000
200	0.403	14988	215	0.403	14976	998
400	0.992	17	200	0.992	17	200
800	0.984	17	200	0.984	17	200
1000	0.980	17	200	0.980	17	200

(d) *SS-simden-7D*

while for ϵ, we inspected a wide range starting from $\epsilon = 40$ and 100 for $d = 2$ and $d \geq 3$, respectively. Only two values of *MinPts* were considered because (i) either 10 or 100 worked well on the synthetic and real data deployed, and (ii) the number of combinations was already huge.

Table 2.2 presents some key statistics for *SS-simden-dD* datasets with $d = 2, 3, 5$ and 7, while Table 2.3 shows the same statistics for *SS-varden-dD*. Remember that the cardinality here is $n = 2$m, implying that there should be around 200 noise points. The number of intended clusters should not exceed the number of restarts whose expectation is 10. But the former number can be smaller, because the seed spreader may not necessarily create a new cluster after a restart, if it happens to jump into the region of a cluster already generated.

TABLE 2.3: Cluster quality under different (*MinPts*, ϵ): SS varying density

	MinPts = 10			*MinPts* = 100		
ϵ	CV Index	# clusters	# noise pts	CV Index	# clusters	# noise pts
100	0.480	1294	50904	0.457	164	774095
200	0.574	70	2830	0.584	153	250018
400	0.946	6	161	0.836	21	18383
800	0.904	6	154	0.939	6	154
1000	0.887	6	153	0.905	6	153

(a) *SS-varden-2D*

	MinPts = 10			*MinPts* = 100		
ϵ	CV Index	# clusters	# noise pts	CV Index	# clusters	# noise pts
100	0.321	1031	577830	0.055	114	1358330
200	0.698	1989	317759	0.403	100	600273
400	0.864	573	23860	0.751	91	383122
800	0.917	11	195	0.908	91	50711
1000	0.904	11	194	0.884	27	236

(b) *SS-varden-3D*

	MinPts = 10			*MinPts* = 100		
ϵ	CV Index	# clusters	# noise pts	CV Index	# clusters	# noise pts
400	0.244	5880	267914	0.523	10160	568393
800	0.755	286	200	0.858	4540	432
1000	0.952	12	200	0.903	1667	357
2000	0.980	8	200	0.980	8	200
3000	0.980	8	200	0.980	8	200

(c) *SS-varden-5D*

	MinPts = 10			*MinPts* = 100		
ϵ	CV Index	# clusters	# noise pts	CV Index	# clusters	# noise pts
400	0.423	7646	801947	0.450	6550	837575
800	0.780	9224	10167	0.686	5050	425229
1000	0.804	7897	200	0.860	8054	506
2000	0.781	1045	200	0.781	1044	400
3000	0.949	13	200	0.949	13	200
4000	0.949	13	200	0.949	13	200

(d) *SS-varden-7D*

Both *MinPts* = 10 and 100, when coupled with an appropriate ϵ, were able to discover all the intended clusters—observe that the CV index stabilizes soon as ϵ increases. We set 10 as the default for *MinPts* on the synthetic datasets, as it produced better clusters than 100 under most values of ϵ. Notice that, for varying-density datasets, ϵ needed to be larger to ensure good clustering quality (compared to similar-density datasets). This is due to the reason explained in Section 2.5 (c.f. Figure 2.11). The bold ϵ values in Tables 2.2 and 2.3 were chosen as the default for the corresponding datasets (they were essentially the smallest that gave good clusters).

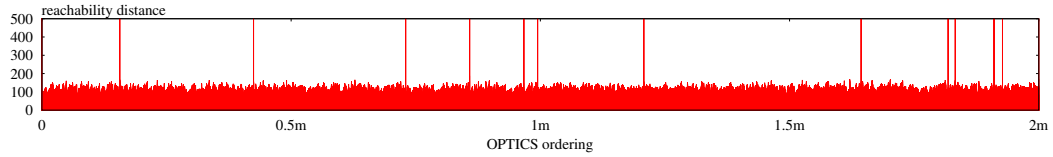
Figure 2.13 plots the OPTICS diagrams[4] for *SS-simden-5D* and *SS-varden-5D*, obtained with

[4]The OPTICS algorithm [6] requires a parameter called *maxEps*, which was set to 10000 in our expeirments.

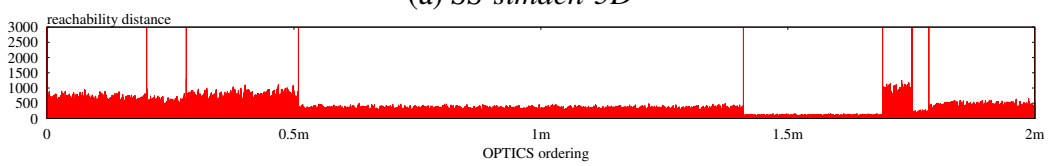
MinPts = 10. In an OPTICS diagram [6], the data points are arranged into a sequence as given along the x-axis. The diagram shows the area beneath a function $f(x) : [1, n] \to \mathbb{R}$, where $f(x)$ can be understood roughly as follows: if p is the x-th point in the sequence, then $f(x)$ is the smallest ϵ value which (together with the chosen *MinPts*) puts p into some cluster—in other words, p remains as a noise point for $\epsilon < f(x)$. A higher/lower $f(x)$ indicates that p is in a denser/sparser area. The ordering of the sequence conveys important information: each "valley"—a subsequence of points between two "walls"–corresponds to a cluster. Furthermore, the points of this valley will remain in a cluster under any ϵ greater than the maximum $f(x)$ value of those points.

Figure 2.13a has 13 valleys, matching the 13 clusters found by $\epsilon = 200$. Notice that the points in these valleys have roughly the same $f(x)$ values (i.e., similar density). Figure 2.13b, on the other hand, has 8 valleys, namely, the 8 clusters found by $\epsilon = 2000$. Points in various valleys can have very different $f(x)$ values (i.e., varying density). The OPTICS diagrams for the other synthetic datasets are omitted because they illustrate analogous observations about the composition of clusters.

Next, we turned to the real datasets. Table 2.4 gives the statistics for *PAMAP2*, *Farm*, and *Household*. The CV indexes are much lower (than those of synthetic data), indicating that the clusters in these datasets are less obvious. For further analysis, we chose *MinPts* = 100 as the default (because it worked much better than *MinPts* = 10), using which Figure 2.14 presents the OPTICS diagrams for the real datasets, while Table 2.5 details the sizes (unit: 1000) of the 10 largest clusters under each ϵ value in Table 2.4. By combining all these data, we make the following observations:



(a) *SS-simden-5D*



(b) *SS-varden-5D*

FIGURE 2.13: Optics diagrams for 5D synthetic data

TABLE 2.4: Cluster quality under different ($MinPts, \epsilon$): real data

	MinPts = 10			*MinPts* = 100		
ϵ	CV Index	# clusters	# noise pts	CV Index	# clusters	# noise pts
100	0.174	6585	2578125	0.103	478	3369657
200	0.222	17622	1890108	0.210	818	2800524
400	0.092	11408	620932	0.226	1129	2396808
800	0.037	3121	215925	0.099	756	949167
1000	0.032	2530	159570	0.078	483	594075
2000	0.033	549	28901	0.126	237	209236
3000	0.237	110	5840	0.302	100	75723
4000	0.106	30	1673	0.492	31	24595
5000	0.490	9	673	0.506	12	9060

(a) *PAMAP2*

	MinPts = 10			*MinPts* = 100		
ϵ	CV Index	# clusters	# noise pts	CV Index	# clusters	# noise pts
100	0.002	925	3542419	0.001	3	3621494
200	0.005	3296	2473933	0.008	21	3404402
400	0.006	1420	1153340	0.191	13	1840989
700	0.004	962	514949	0.364	28	1039114
800	0.004	994	410432	0.198	18	859002
1000	0.005	689	273723	0.295	15	594462
2000	0.002	217	46616	0.120	13	181628
3000	0.001	55	15096	0.131	6	62746
4000	0.058	35	8100	0.764	3	24791
5000	0.024	27	5298	0.157	6	12890

(b) *Farm*

	MinPts = 10			*MinPts* = 100		
ϵ	CV Index	# clusters	# noise pts	CV Index	# clusters	# noise pts
100	0.057	3342	1702377	0.026	54	1944226
200	0.114	5036	1314498	0.074	87	1829873
400	0.085	4802	911088	0.088	165	1598323
800	0.048	2148	490634	0.257	47	974566
1000	0.045	1800	404306	0.227	55	829398
2000	0.129	601	139483	0.416	28	327508
3000	0.074	447	73757	0.241	48	193502
4000	0.007	195	34585	0.565	10	112231
5000	0.015	131	18059	0.649	8	68943

(c) *Household*

- *PAMAP2:* From Figure 2.14a, we can see that this dataset contains numerous "tiny valleys", which explains the large number of clusters as shown in Table 2.4(a). The most interesting ϵ value is 4000, which discovers the two "high-level" clusters: the first one ranges from 0 to 3.5m on the x-axis of Figure 2.14a, while the other one from 3.5m to 3.8m (see the largest two clusters in Table 2.5(a)). Notice from Table 2.4(a) that the CV index is relatively high at $\epsilon = 4000$.

- *Farm:* There are two clusters in the dataset. The first one is the valley between 2.6m and 2.8m on the x-axis of Figure 2.14b, and the second one is the small dagger-shape valley at 3.5m. The

TABLE 2.5: Sizes of the 10 largest clusters: real data (unit: 10^3)

ϵ	1st	2nd	3rd	4th	5th	6th	7th	8th	9th	10th
100	18.8	16.9	11.9	10.2	9.72	8.10	7.00	5.57	5.54	5.54
200	25.9	21.6	20.0	18.9	18.8	18.2	18.2	17.4	16.9	15.5
400	66.9	54.9	39.1	35.2	29.1	28.1	23.8	21.7	20.0	19.4
800	2219	41.5	37.3	26.7	20.5	19.2	19.0	17.4	15.3	13.9
1000	2794	116	20.5	16.4	13.1	9.12	9.08	8.69	7.65	7.10
2000	3409	78.0	18.2	13.3	9.65	9.57	6.60	6.60	5.11	5.04
3000	3470	239	18.5	11.8	2.03	1.90	1.83	1.21	1.20	1.09
4000	*3495*	*315*	2.03	1.86	1.84	0.965	0.786	0.735	0.698	0.687
5000	3497	339	1.85	0.977	0.553	0.551	0.328	0.328	0.217	0.216

(a) *PAMAP2*

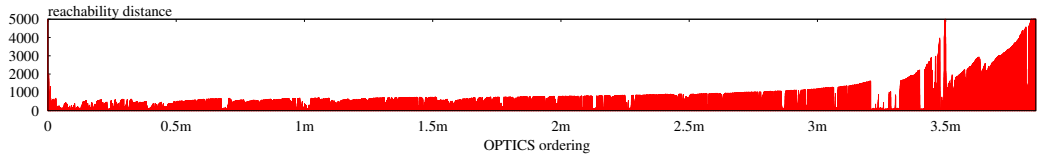
ϵ	1st	2nd	3rd	4th	5th	6th	7th	8th	9th	10th
100	4.10	1.34	0.150							
200	195	18.0	3.93	0.868	0.717	0.647	0.529	0.393	0.391	0.303
400	1604	129	37.3	11.1	1.75	0.713	0.408	0.327	0.265	0.226
700	2282	*218*	44.1	*17.0*	10.4	4.27	3.59	1.12	1.09	0.863
800	2358	381	17.4	6.28	1.34	0.921	0.859	0.545	0.528	0.414
1000	3009	18.0	1.47	0.740	0.718	0.446	0.422	0.332	0.287	0.214
2000	3418	18.8	2.79	1.88	1.45	0.386	0.374	0.230	0.186	0.165
3000	3562	0.951	0.681	0.350	0.190	0.177				
4000	3600	1.08	0.470							
5000	3611	1.18	0.537	0.273	0.130	0.114				

(b) *Farm*

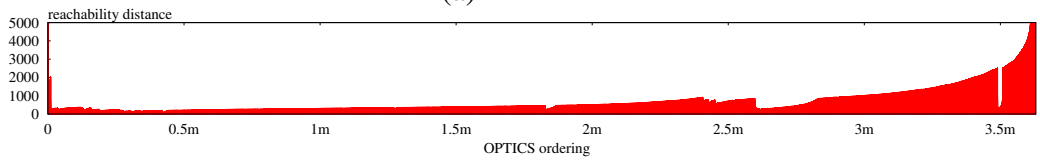
ϵ	1st	2nd	3rd	4th	5th	6th	7th	8th	9th	10th
100	37.8	19.8	16.2	12.3	3.84	3.31	2.33	0.529	0.525	0.505
200	47.8	30.9	24.8	24.5	20.3	15.2	7.79	6.91	5.82	4.39
400	52.6	39.9	34.2	31.9	27.3	25.9	21.1	18.8	15.9	15.8
800	274	193	117	97.3	70.2	48.0	33.9	33.6	27.0	24.8
1000	294	198	158	99.7	94.5	81.7	51.6	30.8	27.2	25.0
2000	*560*	*320*	*222*	*220*	*110*	75.2	71.4	68.2	25.1	25.0
3000	586	337	243	221	111	91.8	85.0	69.7	26.6	26.1
4000	1312	575	17.0	10.9	9.98	7.07	3.71	0.381	0.197	0.100
5000	1918	22.2	14.9	13.3	11.2	0.299	0.101	0.100		

(c) *Household*

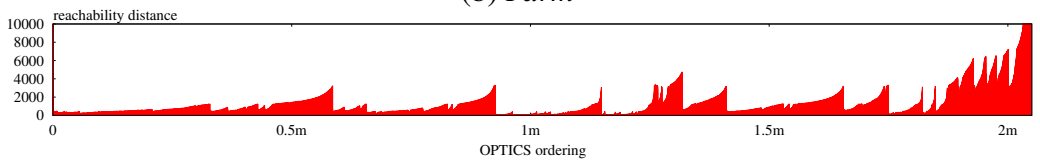
Note: interesting clusters are underlined.



(a) *PAMAP2*



(b) *Farm*



(c) *Household*

FIGURE 2.14: Optics diagrams for real datasets

best value of ϵ that discovers both clusters lies around 700—they are the 2nd and 4th largest clusters at the row of $\epsilon = 700$ in Table 2.5(b).

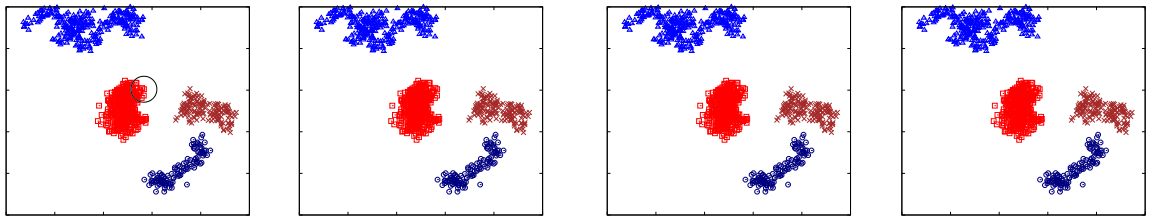
- *Household:* This is the "most clustered" real dataset of the three. It is evident that $\epsilon = 2000$ is an interesting value: it has a relatively high CV index (see Table 2.4(c)), and discovers most of the important valleys in Figure 2.14, whose clusters are quite sizable as shown in Table 2.5(c).

Based on the above discussion, we set the default ϵ of each real dataset to the bold values in Table 2.5.

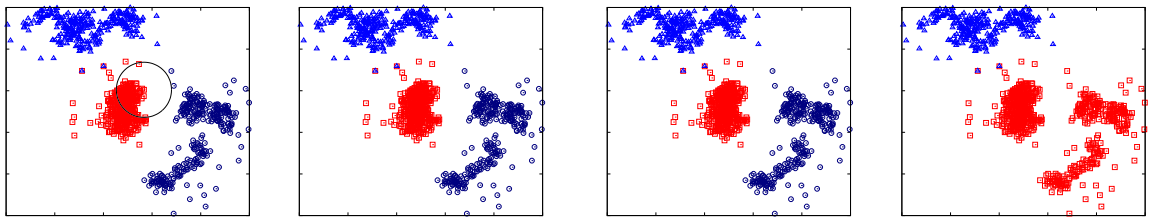
2.6.3 Approximation Quality

In this subsection, we evaluate the quality of the clusters returned by the proposed ρ-approximate DBSCAN algorithm.

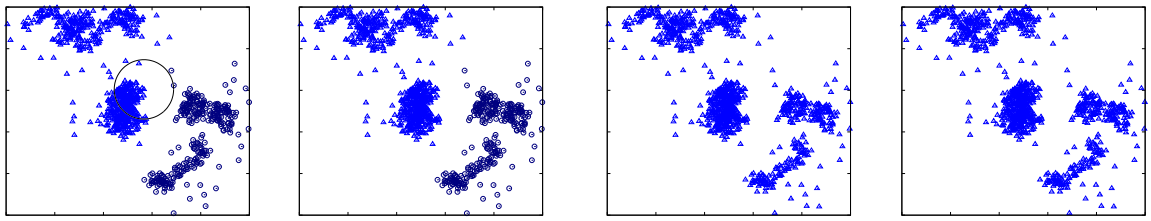
2D Visualization. To show directly the effects of approximation, we take the 2D dataset in Figure 2.12 as the input (note that the cardinality was deliberately chosen to be small to facilitate visualization), and fixed $MinPts = 20$. Figure 2.15a demonstrates the 4 clusters found by exact DBSCAN with $\epsilon = 5000$ (which is the radius of the circle shown). The points of each cluster are depicted with



(a) Exact ($\epsilon = 5000$) (b) $\rho = 0.001, \epsilon = 5000$ (c) $\rho = 0.01, \epsilon = 5000$ (d) $\rho = 0.1, \epsilon = 5000$



(e) Exact ($\epsilon = 11300$) (f) $\rho = 0.001, \epsilon = 11300$ (g) $\rho = 0.01, \epsilon = 11300$ (h) $\rho = 0.1, \epsilon = 11300$

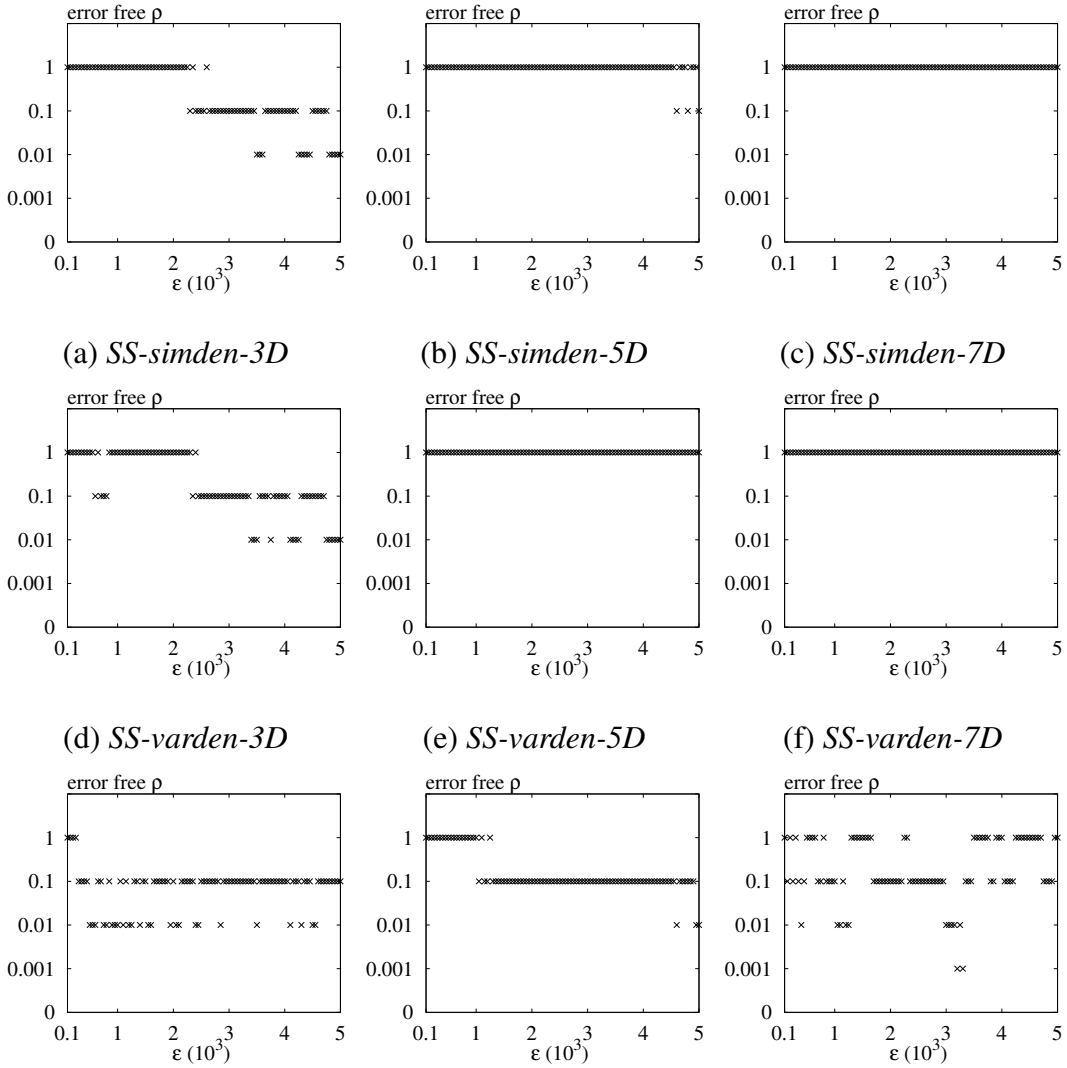


(i) Exact ($\epsilon = 12200$) (j) $\rho = 0.001, \epsilon = 12200$ (k) $\rho = 0.01, \epsilon = 12200$ (l) $\rho = 0.1, \epsilon = 12200$

Figure 2.15: Comparison of the clusters found by exact DBSCAN and ρ-approximate DBSCAN

the same color and marker. Figures 2.15b, 2.15c, and 2.15d present the clusters found by our ρ-approximate DBSCAN when ρ equals 0.001, 0.01, and 0.1, respectively. In all cases, ρ-approximate DBSCAN returned exactly the same clusters as DBSCAN.

Making things more interesting, in Figure 2.15e, we increased ϵ to 11300 (again, ϵ is the radius of the circle shown). This time, DBSCAN found 3 clusters (note that 2 clusters in Figure 2.15a have merged). Figures 2.15f, 2.15g, and 2.15h give the clusters of ρ-approximate DBSCAN for $\rho = 0.001$, 0.01, and 0.1, respectively. Once again, the clusters of $\rho = 0.001$ and 0.01 are exactly the same as DBSCAN. However, 0.1-approximate DBSCAN returned only 2 clusters. This can be understood by observing that the circle in Figure 2.15e almost touched a point from a different cluster. In fact, it will, once ϵ increases by 10%, which explains why 0.1-approximate DBSCAN produced different results.



(a) *SS-simden-3D* (b) *SS-simden-5D* (c) *SS-simden-7D*

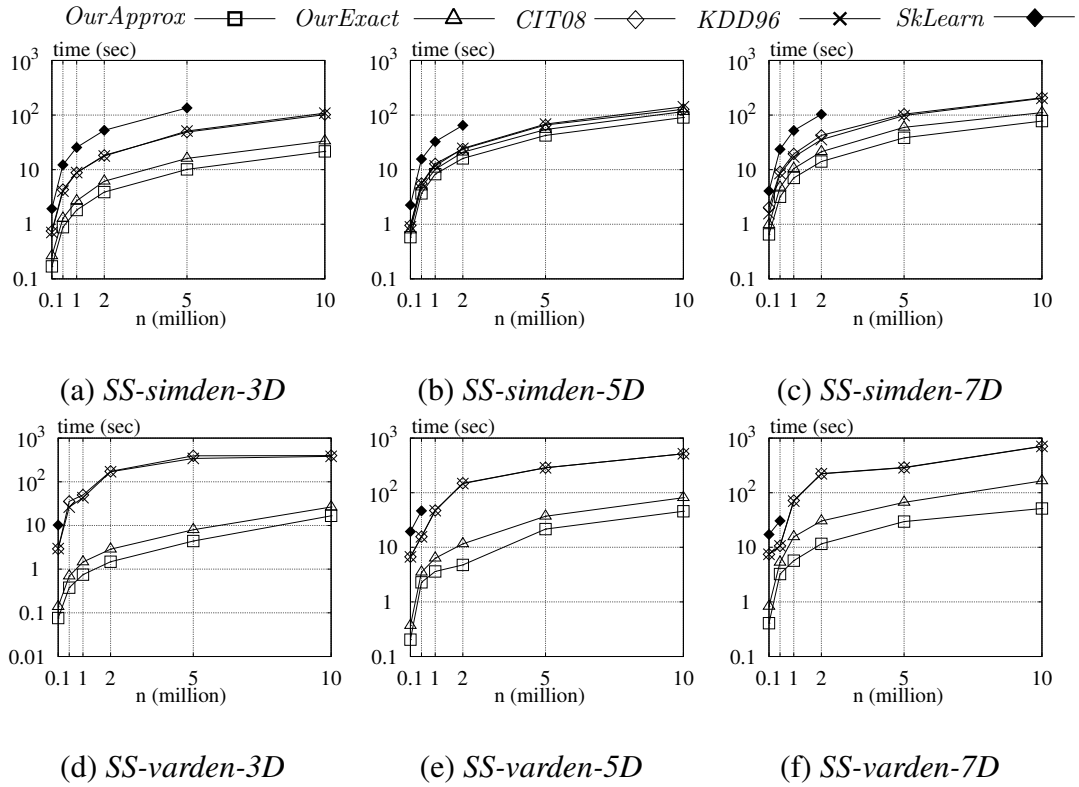
(d) *SS-varden-3D* (e) *SS-varden-5D* (f) *SS-varden-7D*

(g) *PAMAP2* (h) *Farm* (i) *Household*

FIGURE 2.16: Largest ρ in {0.001, 0.01, 0.1, 1} for our ρ-approximate DBSCAN algorithm to return the same results as precise DBSCAN

Then we pushed ϵ even further to 12200 so that DBSCAN yielded 2 clusters as shown in Figure 2.15i. Figures 2.15j, 2.15k, and 2.15l illustrate the clusters of ρ-approximate DBSCAN for ρ = 0.001, 0.01, and 0.1, respectively. Here, both ρ = 0.01 and 0.1 had given up, but ρ = 0.001 still churned out exactly the same clusters as DBSCAN.

Surprised by ρ = 0.01 not working, we examined the reason behind its failure. It turned out that 12200 was extremely close to the "boundary ϵ" for DBSCAN to output 2 clusters. Specifically, as soon as ϵ grew up to 12203, the exact DBSCAN would return only a single cluster. Actually, this can



(a) *SS-simden-3D* (b) *SS-simden-5D* (c) *SS-simden-7D*

(d) *SS-varden-3D* (e) *SS-varden-5D* (f) *SS-varden-7D*

FIGURE 2.17: Running time vs. n ($d \geq 3$)

be seen from Figure 2.15i—note how close the circle is to the point from the right cluster! In other words, 12200 is in fact an "unstable" value for ϵ.

Dimensionalities $d \geq 3$. We deployed the same methodology to study the approximation quality in higher dimensional space. Specifically, for a dataset and a value of ϵ, we varied ρ among $0.001, 0.01, 0.1$ and 1 to identify the highest *error-free* ρ under which our ρ-approximate algorithm returned exactly the same result as precise DBSCAN. Figure 2.16 plots the highest error-free ρ for various datasets when ϵ grew from 100 to 5000. For example, by the fact that in Figure 2.16a the (highest) error-free ρ is 1 at $\epsilon = 100$, one should understand that our approximate algorithm also returned the exact clusters at $\rho = 0.001, 0.01$, and 0.1 at this ϵ. Notice that in nearly all the cases, 0.01-approximation already guaranteed the precise results.

As shown in the next subsection, our current implementation was fast enough on all the tested datasets even when ρ was set to 0.001. We therefore recommend this value for practical use, which was also the default ρ in the following experiments. Recall that, by the sandwich theorem (Theorem 2.3), the result of 0.001-approximate DBSCAN must fall between the results of DBSCAN with

ϵ and 1.001ϵ, respectively. Hence, if 0.001-approximate DBSCAN differs from DBSCAN in the outcome, it means that the (exact) DBSCAN clusters must have changed within the parameter range $[\epsilon, 1.001\epsilon]$.

2.6.4 Computational Efficiency for $d \geq 3$

We now proceed to inspect the running time of DBSCAN clustering in dimensionality $d \geq 3$ using four algorithms:

- *KDD96* [27]: the original DBSCAN algorithm in [27], which deployed a memory R-tree whose leaf capacity was 12 and internal fanout was 4 (these values achieved the best performance);
- *CIT08* [47]: the state of the art of exact DBSCAN, namely, the fastest existing algorithm able to produce the same DBSCAN result as *KDD96*;
- *SkLearn (http://scikit-learn.org/stable)*: the DBSCAN implementation in the popular machine learning tool-kit *scikit-learn*;
- *OurExact*: the exact DBSCAN algorithm we developed in Theorem 2.2, except that we did not use the BCP algorithm in Lemma 2.2; instead, we indexed the core points of each cell with an R-tree, and solved the BCP problem between two cells by repetitive nearest neighbor search [38] using the R-tree;
- *OurApprox*: the ρ-approximate DBSCAN algorithm we proposed in Theorem 2.4.

It is worth noting that, utilizing the heuristics in Section 2.5, our programs have improved efficiency compared to those used in the short version [29].

Each parameter was set to its default value unless otherwise stated. Remember that the default values of *MinPts* and ϵ may be different for various datasets; see Section 2.6.2.

Scalability with *n*. The first experiment examined how each method scales with the number n objects. For this purpose, we used synthetic SS datasets by varying n from 100k to 10m, using the default ϵ and *MinPts* values in Tables 2.2 and 2.3. The results are presented in Figure 2.17—note that the y-axis is in log scale. If *SkLearn* does not have a result at a value of n, it ran out of memory on our

machine (same convention adopted in the rest of the evaluation). *KDD96* and *CIT08* had competitive performance on similar-density datasets, but they were considerably slower (by a factor over an ordge of magnitude) than *OurApprox* and *OurExact* on varying-density data, confirming the analysis in Section 2.5.

Influence of ϵ. The next experiment aimed to understand the behavior of each method under the influence of ϵ. Figure 2.18 plots the running time as a function of ϵ, when this parameter varied from 100 to 5000 (we refer the reader to [29] for running time comparison under $\epsilon > 5000$).

KDD96 and *CIT08* retrieve, for each data point p, all the points in $B(p, \epsilon)$. As discussed in Section 2.5, these methods may be efficient when ϵ is small, but their performance deteriorates rapidly as ϵ increases. This can be observed from the results in Figure 2.22. *OurExact* and *OurApprox* (particularly the latter) offered either competitive or significantly better efficiency at a vast majority of ϵ values. Such a property is useful in tuning this crucial parameter in reality. Specifically, it enables a user to try out a large number of values in a wide spectrum, without having to worry about the possibly prohibitive cost—note that *KDD96* and *CIT08* demanded over 1000 seconds at many values of ϵ that have been found to be interesting in Section 2.6.2.

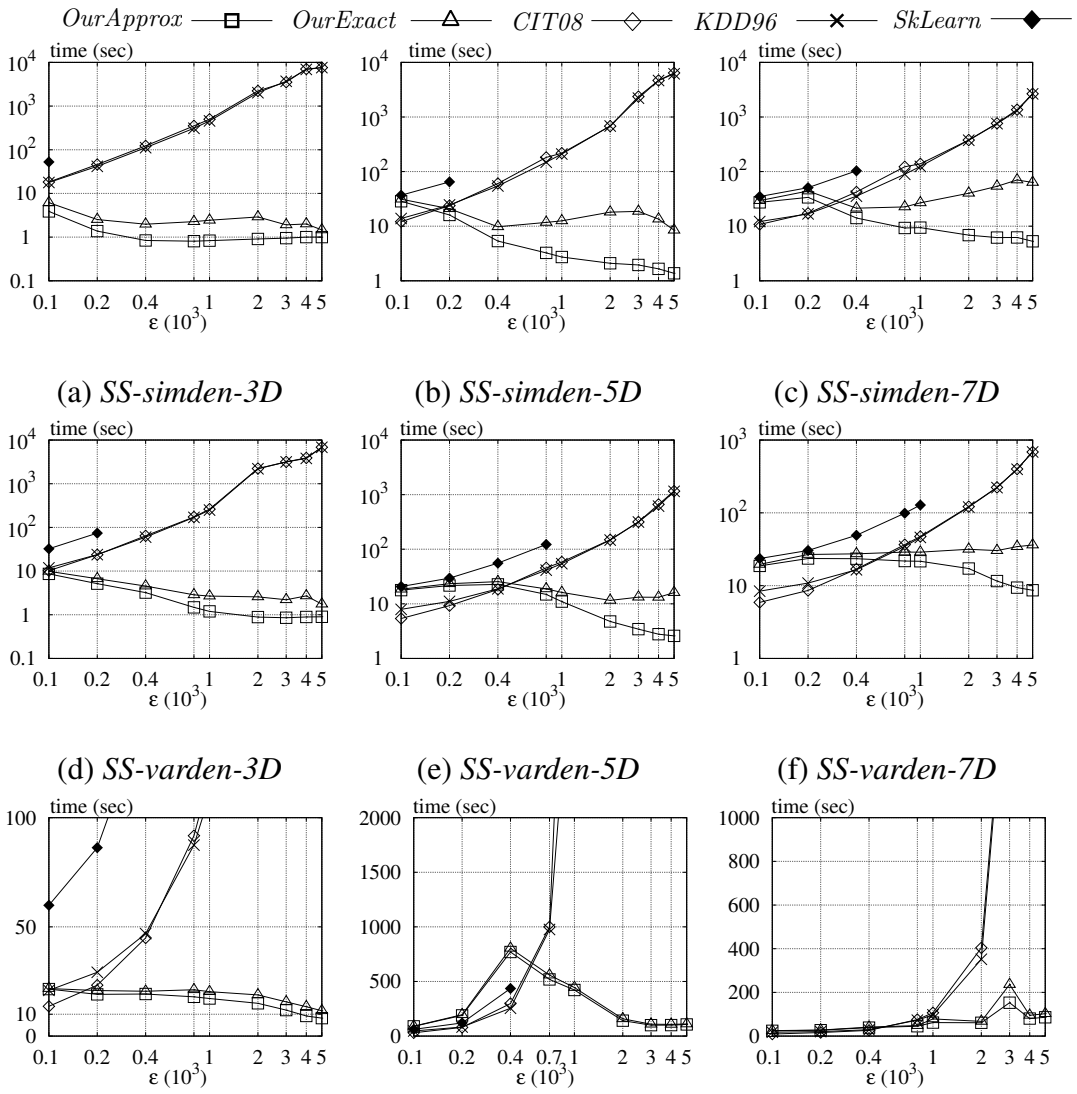
Influence of ρ. Figure 2.19 shows the running time of *OurApprox* as ρ changed from 0.001 to 0.1. The algorithm—thanks to the union-find heuristic explained in Section 2.5—did not appear sensitive to this parameter.

Influence of *MinPts*. The last set of experiments in this section measured the running time of each method when *MinPts* increased from 10 to 100. The results are given in Figure 2.20. The impact of this parameter was limited, and did not change any of the observations made earlier.

2.6.5 Computational Efficiency for $d = 2$

In this subsection, we will focus on exact DBSCAN in 2D space, and compare the following algorithms:

- *KDD96* and *SkLearn*: As introduced at the beginning of Section 2.6.4.
- *G13* [31]: The $O(n \log n)$ time algorithm by Gunawan, as reviewed in Section 2.1.2.

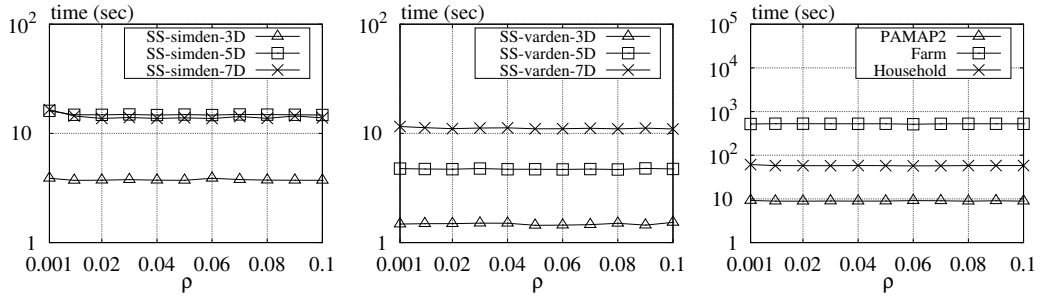


(a) *SS-simden-3D* (b) *SS-simden-5D* (c) *SS-simden-7D*

(d) *SS-varden-3D* (e) *SS-varden-5D* (f) *SS-varden-7D*

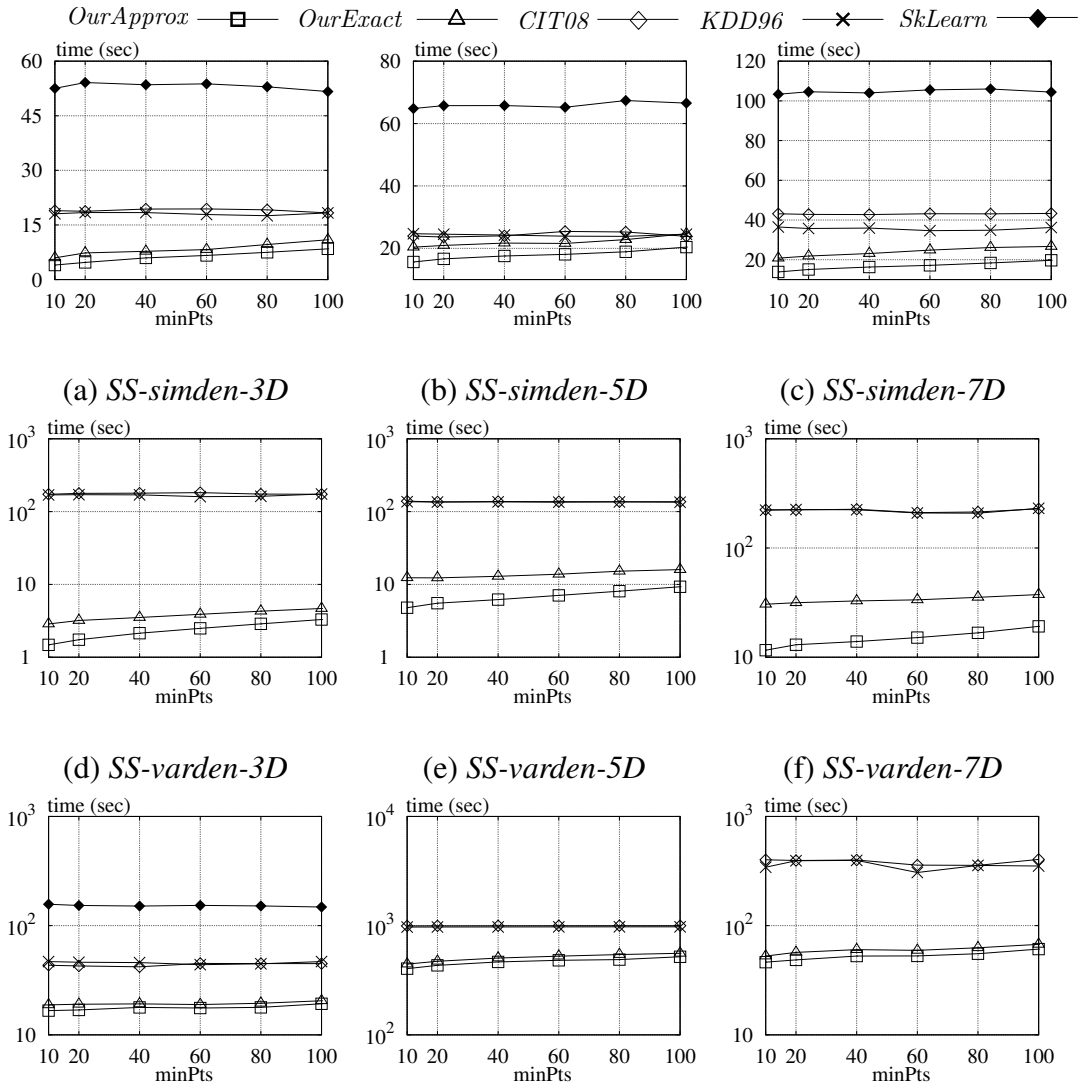
(g) *PAMAP2* (h) *Farm* (i) *Household*

FIGURE 2.18: Running time vs. ϵ ($d \geq 3$)



(a) SS similar density data (b) SS varying density data (c) Real datasets

FIGURE 2.19: Running time vs. ρ ($d \geq 3$)

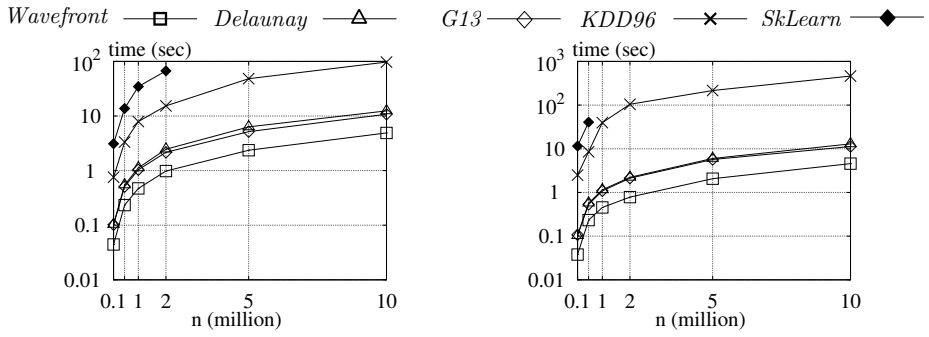


(a) *SS-simden-3D* (b) *SS-simden-5D* (c) *SS-simden-7D*

(d) *SS-varden-3D* (e) *SS-varden-5D* (f) *SS-varden-7D*

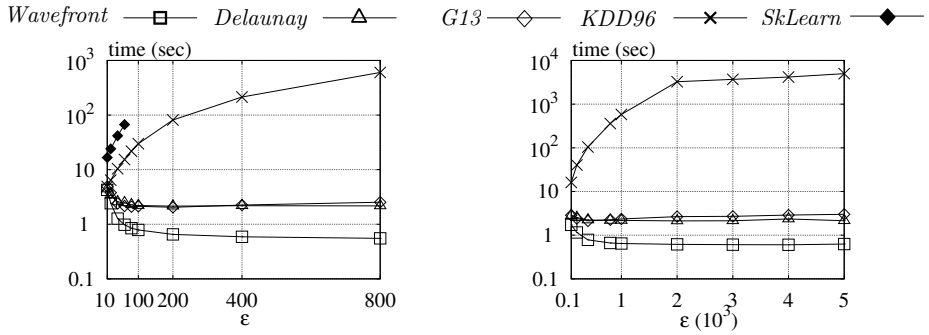
(g) *PAMAP2* (h) *Farm* (i) *Household*

FIGURE 2.20: Running time vs. *MinPts* ($d \geq 3$)



(a) *SS-simden-2D* (b) *SS-varden-2D*

FIGURE 2.21: Running time vs. n ($d = 2$)



(a) *SS-simden-2D* (b) *SS-varden-2D*

FIGURE 2.22: Running time vs. ϵ ($d = 2$)

- *Delaunay*: Our algorithm as explained in Section 2.4.1, which runs in $O(n \log n)$ time.

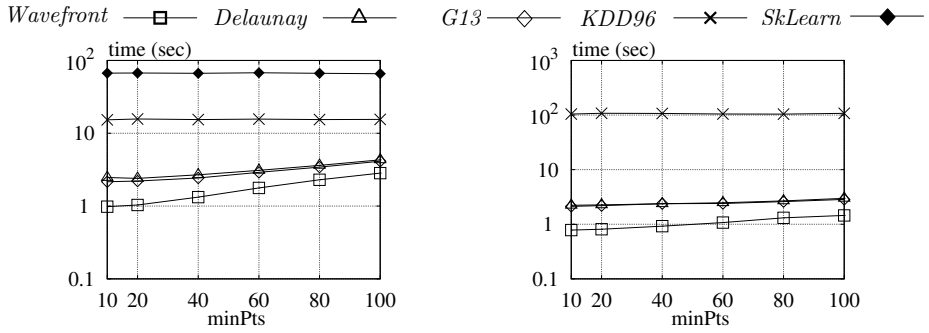
- *Wavefront*: Our algorithm as in Theorem 2.5, assuming that the dataset has been bi-dimensionally sorted—recall that this is required to ensure the linear-time complexity of the algorithm.

Once again, each parameter was set to its default value (see Table 2.1 and Section 2.6.2) unless otherwise stated. All the experiments in this subsection were based on SS similar- and varying-density datasets.

Results. In the experiment of Figure 2.21, we measured the running time of each algorithm as the cardinality n escalated from 100k to 10m. *Wavefront* consistently outperformed all the other methods, while *Delaunay* was observed to be comparable to *G13*. It is worth pointing out the vast difference between the running time here and that shown in Figure 2.17 for $d \geq 3$ (one can feel the difficulty gap of the DBSCAN problem between $d = 2$ and $d \geq 3$).

Next, we compared the running time of the five algorithms by varying ϵ. As shown in Figure 2.22, the cost of *Wavefront*, *Delaunay*, and *G13* actually improved as ϵ grew, whereas *KDD96* and *SkLearn* worsened. *Wavefront* was the overall winner by a wide margin.

Finally, we inspected the influence of *MinPts* on the running time. The results are presented in Figure 2.23. In general, for a larger *MinPts*, *Wavefront*, *Delaunay*, and *G13* require a higher cost in labeling the data points as core or non-core points. The influence, however, is contained by the fact that this parameter is set as a small constant compared to the dataset size. The relative superiority of all the methods remained the same.



(a) *SS-simden-2D* (b) *SS-varden-2D*

FIGURE 2.23: Running time vs. *MinPts* ($d = 2$)

2.7 Summary

DBSCAN is an effective technique for density-based clustering, which is very extensively applied in data mining, machine learning, and databases. However, currently there has not been clear understanding on its theoretical computational hardness. All the existing algorithms suffer from a time complexity that is quadratic to the dataset size n when the dimensionality d is at least 3.

In this chapter, we show that, unless very significant breakthroughs (ones widely believed to be impossible) can be made in theoretical computer science, the DBSCAN problem requires $\Omega(n^{4/3})$ time to solve for $d \geq 3$ under the Euclidean distance. This excludes the possibility of finding an algorithm of near-linear running time, thus motivating the idea of computing approximate clusters. Towards that direction, we propose ρ-approximate DBSCAN, and prove both theoretical and experimentally that the new method has excellent guarantees both in the quality of cluster approximation and computational efficiency.

The exact DBSCAN problem in dimensionality $d = 2$ is known to be solvable in $O(n \log n)$ time. This chapter further enhances that understanding by showing how to settle the problem in $O(n)$ time, provided that the data points have already been pre-sorted on each dimension. In other words, coordinating sorting is in fact the hardest component of the 2D DBSCAN problem. The result immediately implies that, when all the coordinates are integers, the problem can be solved in $O(n \log \log n)$ time deterministically, or $O(n \sqrt{\log \log n})$ expected time randomly.

2.8 Appendix: Solving USEC with Line Separation

We consider that all the discs in S_{ball} intersect ℓ (any disc completely below ℓ can be safely discarded), and that all discs are distinct (otherwise, simply remove the redundant ones).

For each disc $s \in S_{ball}$, we define its portion on or above ℓ as its *active region* (because only this region may contain points of S_{pt}). Also, we use the term *upper arc* to refer to the portion of the boundary of s that is strictly above ℓ. See Figure 2.24 for an illustration of these notions (the upper arc is in bold). Note that, as the center of s is on or below ℓ, the active region and upper arc of s are at most a semi-disc and a semi-circle, respectively. The following is a basic geometric fact:

Proposition 2.1. *The upper arcs of any two discs in S_{ball} can have at most one intersection point.*

Define the *coverage region*—denoted by U—of S_{ball} as the union of the active regions of all the discs in S_{ball}. Figure 2.25a demonstrates U for the example of Figure 2.9. Evidently, the answer of the USEC instance is *yes* if and only if S_{pt} has at least a point falling in U.

We use the term *wavefront* to refer to the part of the boundary of U that is strictly above ℓ; see the solid curve in Figure 2.25b. A disc in S_{ball} is said to be *contributing*, if it defines an arc on the wavefront. In Figure 2.25b, for instance, the wavefront is defined by 3 contributing discs, which are shown in bold and labeled as s_1, s_3, s_6 in Figure 2.25a.

It is rudimentary to verify the next three facts:

Proposition 2.2. *U equals the union of the active regions of the contributing discs in S_{ball}.*

Proposition 2.3. *Every contributing disc defines exactly one arc on the wavefront.*

Proposition 2.4. *The wavefront is x-monotone, namely, no vertical line can intersect it at two points.*

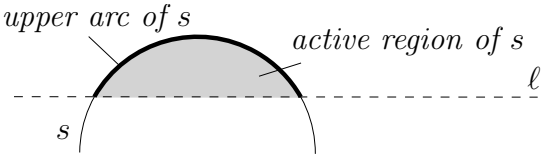
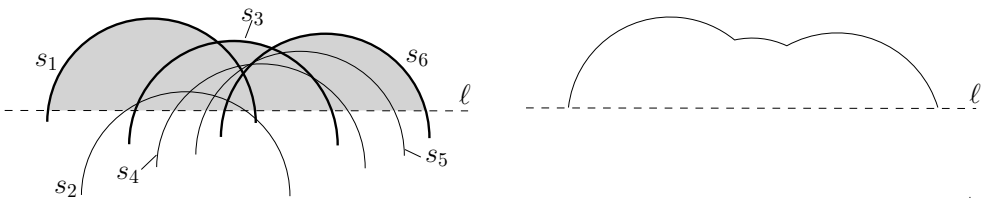


FIGURE 2.24: Illustration of active region and upper arc



(a) Coverage region U (the shaded area) (b) Wavefront (solid curve)

FIGURE 2.25: Deciding the existence of an edge by USEC with line separation

2.8.1 Computing the Wavefront in Linear Time

Utilizing the property that the centers of the discs in S_{ball} have been sorted by x-dimension, next we explain how to compute the wavefront in $O(|S_{ball}|)$ time.

Label the discs in S_{ball} as $s_1, s_2, s_3, ...$, in ascending order of their centers' x-coordinates. Let U_i ($1 \leq i \leq |S_{ball}|$) be the coverage region that unions the active regions of the first i discs. Apparently, $U_1 \subseteq U_2 \subseteq U_3 \subseteq ...$, and $U_{|S_{ball}|}$ is exactly U. Define W_i to be the wavefront of U_i, namely, the portion of the boundary of U_i strictly above ℓ. Our algorithm captures W_i in a linked list $\mathcal{L}(W_i)$, which arranges the defining discs (of W_i) in left-to-right order of the arcs (on W_i) they define (e.g., in Figure 2.10, $\mathcal{L}(W_6)$ lists s_1, s_3, s_6 in this order). By Proposition 2.3, every disc appears in $\mathcal{L}(W_i)$ at most once. Our goal is to compute $W_{|S_{ball}|}$, which is sufficient for deriving U according to Proposition 2.2.

It is straightforward to obtain W_1 from s_1 in constant time. In general, provided that W_{i-1} ($i \geq 2$) is ready, we obtain W_i in three steps:

1. Check if s_i defines any arc on W_i.

2. If the answer is *no*, set $W_i = W_{i-1}$.

3. Otherwise, derive W_i from W_{i-1} using s_i.

Next, we describe how to implement Steps 1 and 3.

Step 1. We perform this step in constant time as follows. Compute the intersection of s_i and ℓ. The intersection is an interval on ℓ, denoted as I_i. Let s_{last} be the rightmost defining disc of W_{i-1} (i.e., the last disc in $\mathcal{L}(W_{i-1})$). If the right endpoint of I_i lies in s_{last}, return *no* (that is, s_i does not define any arc on W_i); otherwise, return *yes*.

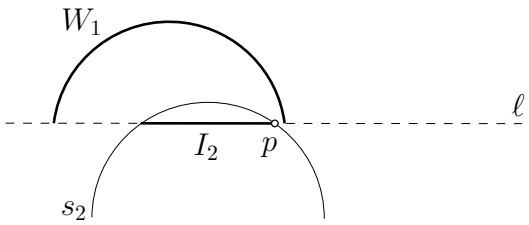


Figure 2.26: Illustration of the Step-1 algorithm in Section 2.8.1

As an example, consider the processing of s_2 in Figure 2.25a. At this moment, W_1 is as shown in Figure 2.26, and includes a single arc contributed by s_1. Point p is the right endpoint of I_2. As p falls in s_1 (= s_{last}), we declare that s_2 does not define any arc on W_2 (which therefore equals W_1).

The lemma below proves the correctness of our strategy in general:

Lemma 2.8. *Our Step-1 algorithm always makes the correct decision.*

Proof. Consider first the case where the right endpoint p of I_i is covered by s_{last}. Let I_{last} be the intersection between s_{last} and ℓ. By the facts that (i) the x-coordinate of the center of s_i is larger than or equal to that of the center of s_{last}, and (ii) s_i and s_{last} have the same radius, it must hold that I_i is contained in I_{last}. This implies that the active region of s_i must be contained in that of s_{last} (otherwise, the upper arc of s_i needs to go out of s_{last} and then back in, and hence, must intersect the upper arc of s_{last} at 2 points, violating Proposition 2.1). This means that s_i cannot define any arc on W_i; hence, our *no* decision in this case is correct.

Now consider the case where p is not covered by s_{last}. This implies that p is not covered by U_{i-1}, meaning that I_i must define an arc on W_i (because at least p needs to appear in U_i). Our *yes* decision is therefore correct. □

Step 3. We derive $\mathcal{L}(W_i)$ by possibly removing several discs at the end of $\mathcal{L}(W_{i-1})$, and then eventually appending s_i. Specifically:

(3. 1) Set $\mathcal{L}(W_i)$ to $\mathcal{L}(W_{i-1})$.

(3. 2) Set s_{last} to the last disc in $\mathcal{L}(W_i)$.

(3. 3) If the arc on W_{i-1} defined by s_{last} is contained in s_i, remove s_{last} from $\mathcal{L}(W_i)$ and repeat from Step 3.2.

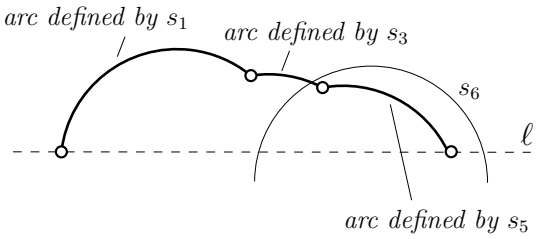


FIGURE 2.27: Illustration of the Step-3 algorithm in Section 2.8.1

(3. 4) Otherwise, append s_i to the end of $\mathcal{L}(W_i)$ and finish.

To illustrate, consider the processing of s_6 in Figure 2.25a. At this moment, the wavefront W_5 is as shown in Figure 2.27, where the arcs are defined by s_1, s_3, and s_5, respectively. Our Step-3 algorithm starts by setting $\mathcal{L}(W_6)$ to $\mathcal{L}(W_5)$, which lists s_1, s_3, s_5 in this order. Currently, $s_{last} = s_5$. As the arc on W_5 defined by s_5 is covered by s_6 (see Figure 2.27), we remove s_5 from $\mathcal{L}(W_6)$, after which s_{last} becomes s_3. As the arc on W_5 defined by s_3 is not contained in s_6, the algorithm terminates by adding s_6 to the end of $\mathcal{L}(W_6)$, which now lists s_1, s_3, s_6 in this order.

Now we prove the correctness of our algorithm:

Lemma 2.9. *Our Step-3 algorithm always obtains the correct $\mathcal{L}(W_i)$.*

Proof. If the arc on W_{i-1} defined by s_{last} is covered by s_i, the upper arc of s_{last} must be covered by the union of the discs in $\{s_1, s_2, ..., s_i\} \setminus \{s_{last}\}$. Therefore, s_{last} is not a defining disc of W_i and should be removed.

Otherwise, s_{last} must be retained. Furthermore, in this case, s_i cannot touch the arc on W_{i-1} defined by any of the discs that are before s_{last} in $\mathcal{L}(W_i)$. All those discs, therefore, should also be retained.

Finally, by Lemma 2.8 and the fact that the execution is at Step 3, we know that s_i defines an arc on W_i, and thus, should be added to $\mathcal{L}(W_i)$. □

Running Time. It remains to bound the cost of our wavefront computation algorithm. Step 1 obviously takes $O(|S_{ball}|)$ time in total. Step 2 demands $\sum_{i=1}^{n} O(1 + x_i)$ time, where x_i is the number of discs deleted at Step 3.3 when processing disc s_i. The summation evaluates to $O(|S_{ball}|)$, noticing that $\sum_{i=1}^{n} x_i \leq n$ because every disc can be deleted at most once.

2.8.2 Solving the USEC Problem

Recall that the USEC instance has a *yes* answer if and only if a point of S_{pt} is on or below the wavefront. Proposition 2.4 suggests a simple planesweep strategy to determine the answer. Specifically, imagine sweeping a vertical line from left to right, and at any moment, remember the (only) arc of the wavefront intersecting the sweeping line. Whenever a point $p \in S_{pt}$ is swept by the line, check whether it falls below the arc mentioned earlier. Because (i) the arcs of the wavefront have been listed from left to right, and (ii) the points of S_{pt} have been sorted on x-dimension, the planesweep can be easily implemented in $O(|S_{ball}| + |S_{pt}|)$ time, by scanning the arcs in the wavefront and the points of S_{pt} synchronously in ascending order of x-coordinate. This concludes the proof of Lemma 2.7, and also that of Theorem 2.5.

Chapter 3

Dynamic Euclidean DBSCAN

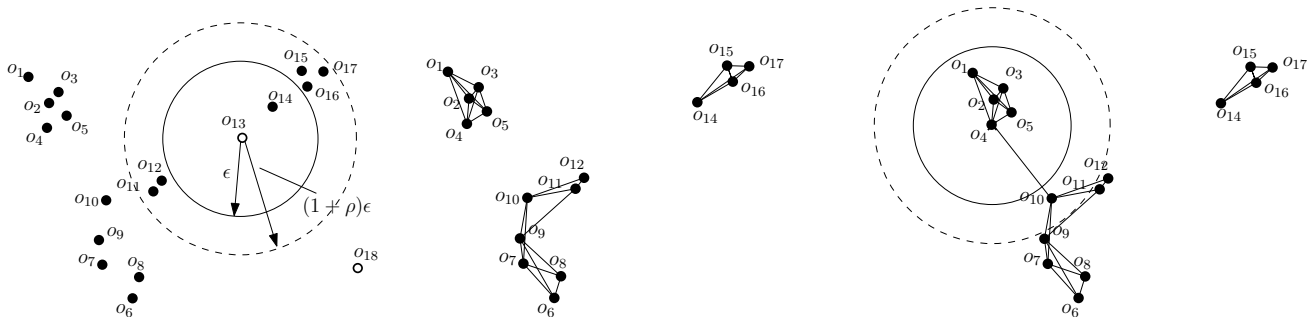
The chapter is organized as follows. The next section reviews the basic concepts and properties of DBSCAN and its ρ-approximate version. Then, Section 3.2 formally defines the dynamic clustering problem studied in this work. Section 3.3 presents a generic framework that captures all the algorithms proposed in this chapter. Section 3.4 elaborates on our semi-dynamic solutions to ρ-approximate DBSCAN. Section 3.5 proves our impossibility result for fully dynamic ρ-approximate DBSCAN, and introduces our "double-approximate" version of DBSCAN, for which Section 3.6 describes fast fully-dynamic algorithms. Section 3.7 reports the results of our experimental evaluation. Finally, Section 3.8 concludes this chapter with a summary of our findings.

3.1 Preliminaries

This section paves the foundation for our technical discussion by clarifying the basic concepts and properties of DBSCAN and its ρ-approximate version.

Let P be a set of points in d-dimensional space $\mathbb{R}^d$. Let us consider the dataset of 18 points in Figure 3.1a, where ϵ is the radius of the inner solid circle, and $MinPts = 3$. The core points have been colored black, while the non-core points colored white. The dashed circle can be ignored for the time being.

A Graph-based Definition. Besides the defintion in Section 2.1.1, DBSCAN clusters can also be equivalently defined in two steps. The first one focuses exclusively on the core points, and groups



(a) Dataset (b) Core graph (c) One possible ρ-approximate core graph

FIGURE 3.1: Illustration of DBSCAN and ρ-approximate DBSCAN ($\rho = 0.5, MinPts = 3$)

them into preliminary clusters. The second step determines how the non-core points should be assigned to the clusters. Next, we explain the two steps in detail.

Step 1: Clustering Core Points. It will be convenient to imagine an undirected *core graph* G on P—this graph is conceptual and need not be materialized. Specifically, each vertex of G corresponds to a distinct *core* point in P. There is an edge between two core points (a.k.a. vertices) p_1, p_2 if and only if $dist(p_1, p_2) \leq \epsilon$, where $dist(\cdot, \cdot)$ represents the Euclidean distance between two points. Figure 3.1b shows the core graph for the dataset of Figure 3.1a.

Each connected component (CC) of G constitutes a preliminary cluster. In Figure 3.1b, there are 3 CCs (a.k.a. preliminary clusters). Note that every core point belongs to *exactly one* preliminary cluster.

Step 2: Non-Core Assignment. This step augments the preliminary clusters with non-core points. For each non-core point p, DBSCAN looks at every core point $p_{core} \in B(p, \epsilon)$, and assigns p to the (only) preliminary cluster containing p_{core}. Note that, in this manner, p may be assigned to zero, one, or more than one preliminary cluster. After all the non-core points have been assigned, the preliminary clusters become final clusters.

It should be clear from the above that the DBSCAN clusters are uniquely defined by the parameters ϵ and *MinPts*, but they are not necessarily disjoint. A non-core point may belong to multiple clusters, while a core point must exist only in a single cluster. It is possible that a non-core point is not in any cluster; such a point is called *noise*.

In Figure 3.1a, there are two non-core points o_{13} and o_{18}. Since $B(o_{13}, \epsilon)$ covers o_{14}, o_{13} is assigned to the preliminary cluster of o_{14}. $B(o_{18}, \epsilon)$, however, covers no core points, indicating that o_{18} is noise. The final DBSCAN clusters are $\{o_1, o_2, ..., o_5\}, \{o_6, o_7, ..., o_{12}\}, \{o_{13}, o_{14}, ..., o_{17}\}$.

The ρ-Approximate Version. The ρ-approximate clusters can also be defined in the same two steps as in exact DBSCAN, as explained below.

Step 1: Clustering Core Points. It will also be convenient to follow a graph-based approach. Let us define an undirected *ρ-approximate core graph* G_ρ on the dataset P—again, this graph is conceptual and need not be materialized. Each vertex of G_ρ corresponds to a distinct core point in P. Given two core points p_1, p_2, whether or not G_ρ has an edge between their vertices is determined as:

- The edge definitely exists if $dist(p_1, p_2) \leq \epsilon$.
- The edge definitely does not exist if $dist(p_1, p_2) > (1 + \rho)\epsilon$.
- Don't care, otherwise.

Each preliminary cluster is still a CC, but of G_ρ. Unlike the core graph G, G_ρ may not be unique. This flexibility is the key to the vast improvement in time complexity.

To illustrate, consider the dataset of Figure 3.1a again with the ϵ shown and $MinPts = 3$, but also with $\rho = 0.5$ (the radius of the dashed circle indicates the length of $(1 + \rho)\epsilon$). Figure 3.1c illustrates a possible ρ-approximate core graph. Attention should be paid to the edge (o_4, o_{10}). Note (from the circles in Figure 3.1c) that $\epsilon < dist(o_4, o_{10}) < (1 + \rho)\epsilon$—this belongs to the "don't-care" case meaning that there may or may not be an edge (o_4, o_{10}). If the edge exists (as in Figure 3.1c), there are 2 CCs (i.e., preliminary clusters); otherwise, the ρ-approximate core graph is the same as in Figure 3.1b, giving 3 preliminary clusters.

Step 2: Non-Core Assignment. Each non-core point p may be assigned to zero, one, or multiple preliminary clusters. Specifically, let S be a CC of G_ρ. Whether p should be added to the preliminary cluster of S is determined as:

- Yes, if S has a core point in $B(p, \epsilon)$.
- No, if S has no core point in $B(p, (1 + \rho)\epsilon)$.
- Don't care, otherwise.

The preliminary clusters after all the assignment constitute the final clusters.

As mentioned, o_{13} and o_{18} are the only two non-core points in Figure 3.1a. While o_{18} is still a noise point, the case of o_{13} is more interesting. First, it *must* be assigned to the preliminary cluster of o_{14}, just like exact DBSCAN. Second, it *may or may not* be assigned to the preliminary cluster of o_{12} (also the cluster of o_{14}). Either case is regarded as a correct result.

3.2 Problem Definition and State of the Art

The Problem of Dynamic Clustering. We now provide a formal formulation of dynamic clustering, using the C-group-by query as the key stepping stone. Our approach is to define the problem in a way that is orthogonal to the semantics of clusters, so that the problem remains valid regardless of whether we have DBSCAN or any of its approximate versions in mind.

Let P be a set of points in $\mathbb{R}^d$ that is subject to *updates*, each of which inserts a new point to P, or deletes an existing point from P. We are given a *clustering description* which specifies correct ways to cluster P. The description is what distinguishes DBSCAN from, e.g., ρ-approximate DBSCAN.

Suppose that, by the clustering description, $C(P)$ is a legal set of clusters on the current contents of P. Without loss of generality, assume that $C(P) = \{C_1, C_2, ..., C_x\}$, where x is the number of clusters, and C_i $(1 \leq i \leq x)$ is a subset of P. Note that the clusters do not need to be disjoint.

Given an arbitrary subset Q of P, a *cluster-group-by* (C-group-by) *query* must return for every $C_i \in C(P)$ $(i \in [1, x])$:

- Nothing at all, if $C_i \cap Q = \emptyset$
- $C_i \cap Q$, otherwise.

This definition has several useful properties:

- It breaks *only* the points of Q by how they should appear together in the clusters of P. Points in $P \setminus Q$ are not reported at all, thus avoiding "cheating algorithms" that use "expensive report time" as an excuse for high processing cost.
- When $Q = P$, the query result $Q(P)$ is simply $C(P)$.

- All the query results must be based on the *same* $C(P)$. This prevents another form of "cheating" when the clustering description permits multiple legal possibilities of $C(P)$. Specifically, the algorithm can no longer argue that the results $Q_1(P)$ and $Q_2(P)$ of two queries Q_1 and Q_2 should both be "correct" because $Q_1(P)$ is defined on one possible $C(P)$, while $Q_2(P)$ is defined on another. Instead, they must be consistently defined on the same $C(P)$—the one output by the query with $Q = P$.

Our objective is to design an algorithm that is fast in processing both updates and queries. We distinguish two scenarios: (i) *semi-dynamic*: where all the updates are insertions, and (ii) *fully-dynamic*, where the updates can be arbitrary insertions and deletions.

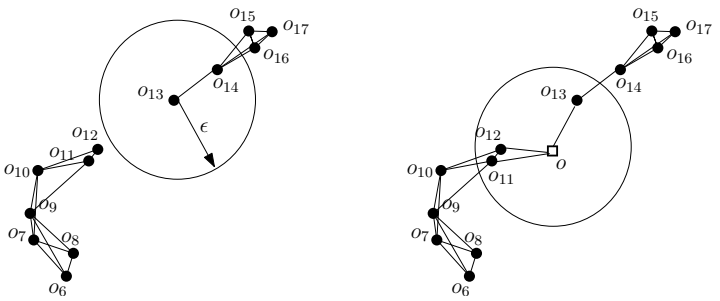
We consider that the dimensionality d is small such that $(\sqrt{d})^d$ is an acceptable hidden constant. All our theoretical results will carry this constant, and hence, are suitable only for low dimensionality. Our experiments run up to $d = 7$.

Dynamic Exact DBSCAN [26]. Dynamic maintenance of density-based clusters has been studied by Ester et al. [26] for exact DBSCAN. Next, we review their method—named *incremental DBSCAN* (*IncDBSCAN*)—assuming $MinPts = 1$ so that all the points of P are core points. This allows us to concentrate on the main ideas without the relatively minor details of handling non-core points.

Insertion. Recall that, for exact DBSCAN, the clusters $C(P)$ are uniquely determined by the input parameters ϵ and $MinPts$. Given a new point p_{new}, the insertion algorithm retrieves all the points in $B(p_{new}, \epsilon)$, and then merges the clusters of those points into one.

The correctness can be seen from the core graph G, where effectively an edge is added between p_{new} and every other point in $B(p_{new}, \epsilon)$ (remember: this view is conceptual, and G does not need to be materialized). Figure 3.2a shows the G before the insertion, which has two CCs. To insert point o as in Figure 3.2b, the algorithm finds the points o_{11}, o_{12}, and o_{13} in $B(p_{new}, \epsilon)$. The two clusters of those points are merged—the newly added edges (o, o_{11}), (o, o_{12}), (o, o_{13}) in Figure 3.2b connect the two CCs into one.

In merging the clusters, IncDBSCAN does not modify the cluster ids of the points in the affected clusters, which can be prohibitively expensive because the number of such points can be exceedingly high. Instead, it remembers the "merging history" of the cluster ids.



(a) Core graph without o (b) Core graph with o

FIGURE 3.2: Illustration of the IncDBSCAN method

Deletion. The deletion algorithm, in general, reverses the steps of insertion, except for a breadth first search (BFS) that is needed to judge whether (and how) a cluster has split into several.

Let p_{old} be the point being deleted. IncDBSCAN retrieves all the points in $B(p_{old}, \epsilon)$. In (the conceptual) G, all the edges between such points and p_{old} are removed, after which the CC of p_{old} may or may not be broken into separate CCs (e.g., in Figure 3.2b, the CC is torn into two only if o_{13}, o_{14}, or o is deleted). To find out, the deletion algorithm performs as many threads of BFS on G as the number of points in $B(p_{old}, \epsilon)$. If two threads "meet up", they are combined into one because they must still be in the same CC. As soon as only one thread is left, the algorithm terminates, being sure that no cluster split has taken place. Otherwise, the remaining threads continue until the end, in which case each thread has enumerated all the points in a new cluster that is spawned by the deletion. All those points can now be labeled with the same cluster id.

For example, suppose that we delete o in Figure 3.2b, which starts three threads of BFS from o_{11}, o_{12} and o_{13}, respectively. The resulting core graph reverts back to Figure 3.2a. The threads of o_{11} and o_{12} meet up into one, which eventually traverses the entire CC containing o_{11} and o_{12}. Similarly, the other thread traverses the entire CC containing o_{13}.

When a thread of BFS needs the adjacent neighbors of a point p_1 in G, the algorithm finds all the other points $p_2 \in B(p_1, \epsilon)$ through a *range query* [14, 32]. Every such p_2 is an adjacent neighbor of p_1. This essentially "fetches" the edge between p_1 and p_2.

Query. The algorithm of [26] can easily answer a C-group-by query Q by grouping the points of Q by their cluster ids (some ids need to be obtained from the merging history).

Drawbacks of IncDBSCAN. Both insertion and deletion start with a range query to extract the points in $B(p, \epsilon)$, which are called the *seed points* [26]. The query is expensive when p falls in a dense region of P where there are many seed points. The issue is more serious in a deletion, because multiple range queries are needed to perform BFS. The worst situation happens in a cluster split, where the number of range queries is simply huge.

3.3 The Overall Framework

All the DBSCAN variants (including the new one to be proposed in Section 3.5.2) accept parameters ϵ, *MinPts*, and ρ (for exact DBSCAN, $\rho = 0$). This permits us to extract a common structural framework behind all our solutions, as we describe in this section. The components of the framework will be instantiated differently in later sections for individual variants.

3.3.1 A Grid Graph Approach

The key idea behind our framework is to turn dynamic clustering into the problem of maintaining CCs (connected components) of a special graph.

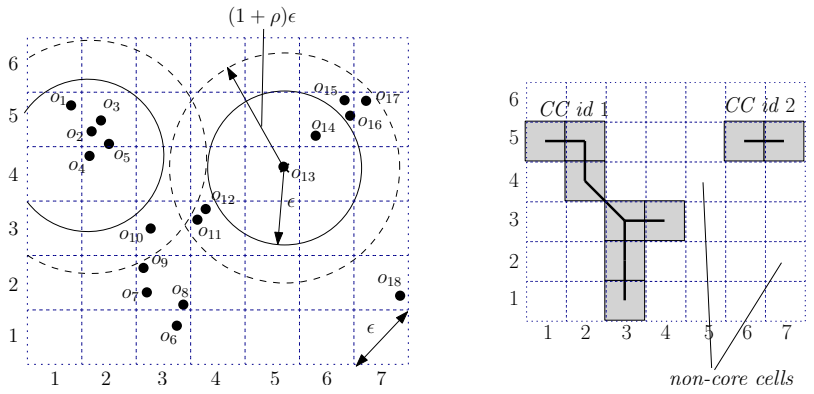
Grid and Cells. We impose an arbitrary grid $\mathbb{D}$ in the data space $\mathbb{R}^d$, where each cell is a d-dimensional square with side length $\epsilon/\sqrt{d}$ on every dimension. This ensures that any two points in the same cell are within distance at most ϵ from each other.

Given a cell c of $\mathbb{D}$, we denote by $P(c)$ the set of points in P that are covered by c. We call c

- A *non-empty cell* if $P(c)$ contains at least one point.
- A *core cell* if $P(c)$ contains at least one core point.
- A *dense cell* if $|P(c)| \geq MinPts$, and a *sparse cell* if $1 \leq |P(c)| < MinPts$.

Given two cells c_1, c_2, we say that they are *ϵ-close* if the smallest distance between the boundary of c_1 and that of c_2 is at most ϵ.

Consider for instance the grid in Figure 3.3a, imposed on a set P of 18 points. Again, the radii of the solid and dashed circles indicate ϵ and $(1 + \rho)\epsilon$, respectively; and $MinPts = 3$. The only non-core



(a) Grid and cells (b) A grid graph

FIGURE 3.3: Our grid-graph framework (*MinPts* = 3)

points are o_{13} and o_{18}. The core cells are shaded in Figure 3.3b. The non-core cells are $(5, 4)$ and $(7, 2)$; note that the minimum distance between the two cells is ϵ—hence, they are ϵ-close.

Grid Graph. In a *grid graph* $\mathbb{G} = (\mathbb{V}, \mathbb{E})$, $\mathbb{V}$ is the set of core cells of $\mathbb{D}$, while $\mathbb{E}$ is a set of edges satisfying the following *CC requirement*:

> Let p_1, p_2 be two core points of P, and c_1 (or c_2, resp.) be the core cell that contains p_1 (or p_2, resp.). Then, p_1 and p_2 are in the same cluster *if and only if* c_1 and c_2 are in the same CC of $\mathbb{G}$.

The above requirement is fulfilled by using the following rules to decide if $\mathbb{E}$ should have an edge between two core cells $c_1, c_2 \in \mathbb{V}$:

- Yes, if there is a pair of core points $(p_1, p_2) \in P(c_1) \times P(c_2)$ satisfying $dist(p_1, p_2) \leq \epsilon$.
- No, if there is *no* pair of core points $(p_1, p_2) \in P(c_1) \times P(c_2)$ satisfying $dist(p_1, p_2) \leq (1+\rho)\epsilon$.
- Don't care, otherwise.

$\mathbb{G}$ differs significantly from the core graph G and the ρ-approximate core graph of G_ρ as reviewed in Section 3.1. $\mathbb{G}$ has at most n vertices because there are at most n non-empty cells. If $\mathbb{G}$ has an edge between core cells c_1, c_2, they must be ϵ-close. A core cell can have $O\Big(\Big(\frac{\epsilon}{\epsilon/\sqrt{d}}\Big)^d\Big) = O((\sqrt{d})^d) = O(1)$ ϵ-close core cells (recall that our target is low dimensionality). Hence, $\mathbb{G}$ can have only $O(n)$ edges, which makes it suitable for materialization when the dimensionality d is low.

Figure 3.3b demonstrates the grid graph for the dataset in Figure 3.3a. Note that the edge between cells $(2, 4)$ and $(3, 3)$ fall into the don't-care case, because $\epsilon < dist(o_4, o_{10}) \leq (1+\rho)\epsilon$. That $\mathbb{G}$ satisfies

the CC requirement can be seen together with the ρ-approximate core graph in Figure 3.1c. For example, o_3 and o_6 are in the same cluster, consistent with the fact that cells $(2, 5)$ and $(3, 1)$ are in the same CC of $\mathbb{G}$. Conversely, as cells $(2, 5)$ and $(6, 5)$ are in different CCs of $\mathbb{G}$, we know o_3 and o_{14} must be in different clusters.

3.3.2 Query Algorithm

All our solutions actually use the same algorithm to answer C-group-by queries. We explain the algorithm in this subsection, as well as the necessary data structures.

Core-Status Structure. We explicitly record whether each point in P is a core or non-core point. This structure maintains these core-status labels under insertions and deletions of the points in P. A semi-dynamic structure only needs to support insertions.

ρ-Approximate ϵ-Emptiness. Given a point q and a core cell c, an *emptiness query* $empty(q, c)$ returns:

- 1, if $P(c)$ has a core point p satisfying $dist(p, q) \leq \epsilon$.
- 0, if *no* core point $p \in P(c)$ satisfies $dist(p, q) \leq (1 + \rho)\epsilon$.
- 1 or 0 (don't care), otherwise.

As a furthermore requirement, if the output is 1, the query must also return a *proof point* $p \in P(c)$, which is a core point satisfying $dist(q, p) \leq (1 + \rho)\epsilon$.

For example, for $q = o_{13}$ and $c =$ cell $(6, 5)$, the emptiness query must return 1 due to the presence of o_{14}. If c changes to cell $(3, 2)$, then the query must return 0. Setting $q = o_4$ and $c =$ cell $(3, 3)$ gives a don't-care case.

We maintain an *emptiness structure* on every core cell c to support (i) such queries efficiently, and (ii) insertions/deletions of core points in $P(c)$. Deletions are not needed if the structure is semi-dynamic.

CC Structure. We maintain a structure on $\mathbb{G}$ to support:

- $EdgeInsert(c_1, c_2)$: Add an edge between core cells c_1, c_2 to $\mathbb{G}$.
- $EdgeRemove(c_1, c_2)$: Remove the aforementioned edge.

- *CC-Id*(c): Given a core cell c, return a unique id of the CC of $\mathbb{G}$ where c belongs.

If a CC structure is semi-dynamic, it does not need to support *EdgeRemove*.

C-Group-By Query. Next, we clarify how to answer a C-group-by query Q. Divide Q into a set Q_1 of core points, and a set Q_2 of non-core points. This takes $O(|Q|)$ time using the core-status structure. For every core point $q \in Q_1$, we retrieve the core cell c covering q, perform *CC-Id*(c), and set the CC id as the cluster id of q.

A non-core point $q \in Q_2$, on the other hand, is "snapped" to the nearby core cells. Again, obtain the cell c covering q. If c is a core cell, assign the cluster id *CC-Id*(c) to q. In any case, we enumerate all the ϵ-close core cells c' of c. For every c', issue an emptiness query *empty*(q, c'). If the emptiness query returns 1, assign the output of *CC-Id*(c') as a cluster id of q. Note that q may get assigned multiple cluster ids.

We can now group the points in Q by cluster id. A non-core point belongs to as many groups as the number of its distinct cluster ids; if a non-core point has no cluster ids, it is a noise point.

Consider $Q = \{o_{13}, o_{14}, o_8\}$ in the dataset of Figure 3.3a. Q_1 includes core points o_{14} and o_8, which are in cells $(6, 5)$ and $(3, 2)$, respectively. Invoking *CC-Id* on $(6, 5)$ returns 2 (see Figure 3.3b), while doing so to $(3, 2)$ returns 1. Q_2 has only a single non-core point, in cell $(5, 4)$, whose ϵ-close core cells are $(4, 3)$, $(3, 3)$, $(3, 2)$, $(6, 5)$, and $(7, 5)$. We perform an emptiness query using o_{13} on each of those 5 cells. Suppose that the emptiness queries on $(4, 3)$ and $(6, 5)$ return 1, while the others 0. We thus assign two distinct CC ids to o_{13}: 1 and 2. The final result of the C-group-by query is therefore $\{o_{14}, o_{13}\}$, and $\{o_8, o_{13}\}$.

3.3.3 Graph Maintenance

To guarantee correctness, we must keep the grid graph $\mathbb{G}$ up-to-date along with the insertions and deletions on the underlying dataset P. This is accomplished through the collaboration of the core-status structure, GUM (see below), and the CC structure.

Graph Update Module (GUM). This module is responsible for maintaining the vertices and edges in $\mathbb{G}$.

Remark. Figure 3.4 illustrates the data flow in the internal workings of our update mechanism. The

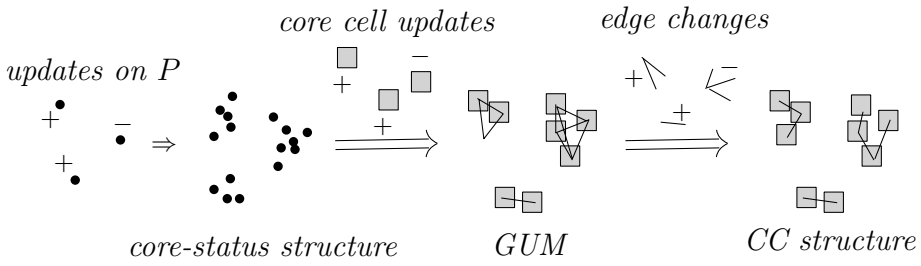


Figure 3.4: Flow of updates in our structures

point insertions and deletions in P are fed into the core-status structure, which informs GUM about which cells have turned into core/non-core cells. Utilizing such information, GUM updates $\mathbb{G}$ by generating the necessary edge changes, which are passed to the CC structure for properly maintaining the CCs of $\mathbb{G}$.

Overall, our design will focus on GUM and the core-status structure. CC and emptiness structures have been well-studied in graph theory and computational geometry, respectively; it suffices to plug in the best existing structures suiting our purposes.

3.4 Semi-Dynamic Algorithms

This section presents maintenance algorithms for exact/approximate DBSCAN clustering when all the updates are insertions. We will do so by specializing the framework of the last section.

The Core-Status Structure. For each non-core point $p \in P$, we remember a *vicinity count* $vincnt(p)$ which equals the number of points of P covered by $B(p, \epsilon)$. By non-core definition, $vincnt(p) < MinPts$. Once $vincnt(p)$ reaches $MinPts$, p becomes a core point, after which we no longer keep track of such a count.

Let us see how to maintain the above information when a new point p_{new} is inserted. Let c_{new} be the cell of $\mathbb{D}$ that contains p_{new}. We start by checking if p_{new} is a core point as follows:

1 If c_{new} is dense, p_{new} must a core point (all the points in c_{new} are within distance ϵ from p_{new}).

2 Otherwise, we simply enumerate all the $O(1)$ ϵ-close cells c of c_{new}, and calculate the distances from p_{new} to all the points in $P(c)$. This way, we obtain the precise number of points in $B(p_{new}, \epsilon)$, noticing that any point within distance ϵ from p_{new} must be in an ϵ-close cell. The

core-status of p_{new} can now be decided.

The appearance of p_{new} may increase the vicinity count $vincnt(p)$ of some non-core points p. Such p must be covered in cells c that are (i) sparse, and (ii) ϵ-close to c_{new}. We find all these points by simply visiting the $P(c)$ of all such c.

GUM. In general, whenever we have a new core point p_{core} (it may be p_{new}, or a point p that just has its $vincnt(p)$ increased), $\mathbb{G}$ may need to be updated. Let c_{core} be the cell covering p_{core}. If c_{core} just became a core cell, we add it into $\mathbb{V}$. In any case, new edges are potentially added to $\mathbb{E}$ as follows:

1 For every ϵ-close cell c of c_{core} that currently has no edge with c_{core} in $\mathbb{G}$

1.1 Perform an emptiness query $empty(p_{core}, c)$.

1.2 If the query returns 1, add (c, c_{core}) to $\mathbb{E}$ and call *EdgeInsert* (c, c_{core}).

Performance Guarantees. Using the best CC and emptiness structures under the semi-dynamic scheme, we prove in the appendix:

Theorem 3.1. *For any fixed dimensionality d and fixed constant $\rho > 0$, there is a semi-dynamic ρ-approximate DBSCAN algorithm that processes each insertion in $\tilde{O}(1)$ amortized time, and answers a C-group-by query Q in $\tilde{O}(|Q|)$ time.*

The same insertion and query efficiency can also be achieved in 2D space for exact DBSCAN.

3.5 Dynamic Hardness and Double Approximation

We now come to perhaps the most surprising section of the paper. Recall that ρ-approximate DBSCAN was proposed to address the computational hardness of DBSCAN on *static* datasets. In Section 3.5.1, we will show that the ρ-approximate version suffers from the same hardness on *fully dynamic* datasets. Interestingly, the culprit this time is the definition of core point. This motivates the proposition of *ρ-double-approximate* DBSCAN in Section 3.5.2, where we also prove that the new proposition has a sandwich guarantee *as strong as* the ρ-approximate version.

3.5.1 Hardness of Dynamic ρ-Approximation

USEC with Line Separation. Next, let us review the *USEC with line separation* (USEC-LS) *problem*, which has a subtle connection with ρ-approximate DBSCAN, as shown later.

In USEC-LS, we are given a set S_{red} of red points and a set S_{blue} of blue points in $\mathbb{R}^d$, which are separated by a d-dimensional plane ℓ perpendicular to the first dimension, such that all the red points are on one side of ℓ, and all the blue points on the other. All the points have distinct coordinates on every dimension. The objective, as with USEC (see Section 3.1), is to determine whether there exist $p_{red} \in S_{red}$ and $p_{blue} \in S_{blue}$ such that $dist(p_{red}, p_{blue}) \leq 1$. We define $n = |S_{red}| + |S_{blue}|$. Figure 3.5a shows an example where the answer is "yes".

Recall from Section 3.1 that USEC is computationally hard. In the appendix, we prove that this is also true for USEC-LS:

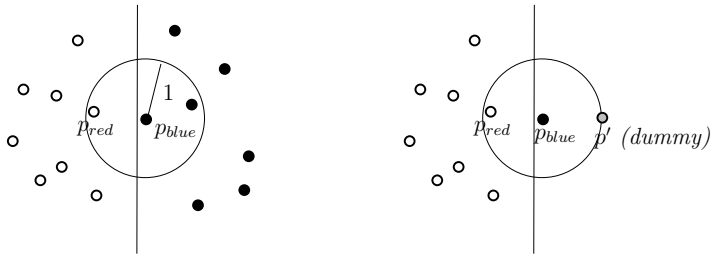
Lemma 3.1. *If we can solve USEC-LS in $o(n^{4/3})$ time, then we can solve USEC in $o(n^{4/3})$ time.*

Dynamic Hardness. Suppose that we have a ρ-approximate DBSCAN algorithm that handles an update (insertion/deletion) in $T_{upd}(n)$ amortized time, and answers a C-group-by query with $|Q| = 2$ in $T_{qry}(n)$ amortized time. Then:

Lemma 3.2. *We can solve the USEC-LS problem in $O(n \cdot (T_{upd}(n) + T_{qry}(n))$ time.*

Proof. Let x be the coordinate where the separation plane ℓ in USEC-LS intersects dimension 1. Without loss of generality, let us assume that the red points are on the *left* of ℓ, i.e., having coordinates less than x on dimension 1. Conversely, the blue points are on the *right* of ℓ. We solve USEC-LS using the given dynamic ρ-approximate DBSCAN algorithm A as follows.

1. Initialize a ρ-approximate DBSCAN instance with $\epsilon = 1$, $MinPts = 3$, and an arbitrary $\rho \geq 0$. Let P be the input set, which is empty at this moment.
2. Use A to insert all the red points into P.
3. For every blue point $p = (x_1, x_2, ..., x_d)$ (hence, $x_1 > x$), carry out the following steps:
   3.1 Use A to insert p into P.



(a) USEC-LS (b) Reduction to dynamic clustering

FIGURE 3.5: Illustration of our hardness proof

3.2 Use A to insert a dummy point $p' = (x_1 + 1, x_2, ..., x_d)$ into P. That is, p' has the same coordinates as p on all dimensions $i \in [2, d]$, except for the first dimension where p' has coordinate $x_1 + 1$. See Figure 3.5b for an illustration (where $p = p_{blue}$).

3.3 Use A to answer a C-group-by query with $Q = \{p, p'\}$. If the query returns the same cluster id for p and p', terminate the algorithm, and return "yes" to the USEC-LS problem.

3.4 Use A to delete p' and p from P.

4. Return "no" to the USEC-LS problem.

The running time of the algorithm is $O(n \cdot (T_{upd}(n) + T_{qry}(n))$ because we issue at most $2n$ insertions and $2n$ deletions, as well as n queries, in total. Next, we prove that the algorithm is correct.

Consider Lines 3.1-3.4. A crucial observation is that the dummy point p' must be a non-core point, because $B(p', \epsilon)$ contains only two points p, p'. Therefore, p' and p are placed into the same cluster by ρ-approximate DBSCAN *if and only if* p is a core point. However, p is a core point if and only if $B(p, \epsilon)$ covers at least 3 points, which must include p, p', and at least one point p'' on the other side of ℓ—red point p'' and blue point p are therefore within distance 1.

It is now straightforward to verify that our algorithm always returns the correct answer for USEC-LS. □

Theorem 3.2. *For any $\rho \geq 0$ and any dimensionality $d \geq 3$, any ρ-approximate DBSCAN algorithm must incur $\Omega(n^{1/3})$ amortized time either to process an update, or to answer a C-group-by query (even if $|Q| = 2$), unless the USEC problem in $\mathbb{R}^d$ could be solved in $o(n^{4/3})$ time.*

Proof. Suppose that the algorithm were able to process an update and a query both in $o(n^{1/3})$ amortized time. By Lemma 3.2, we would solve USEC-LS in $o(n^{4/3})$ time which, by Lemma 3.1, means that we would solve USEC in $o(n^{4/3})$ time. □

As explained in Section 3.1, for USEC, a lower bound of $\Omega(n^{4/3})$ is known [24] in $d \geq 5$, whereas beating the $O(n^{4/3})$ bound in $d = 3, 4$ is a major open problem in theoretical computational geometry, and believed to be impossible [23].

This is disappointing because DBSCAN succumbing to the hardness of USEC was what motivated ρ-approximate DBSCAN. Theorem 3.2 shows that the latter suffers from the same hardness when both insertions and deletions are allowed! Note that the theorem does not apply to the semi-dynamic update scheme because the deletions at Line 3.4 are essential. In fact, Theorem 3.1 already proved that efficient semi-dynamic algorithms exist for ρ-approximate DBSCAN.

Finally, it is worth pointing out that Theorem 3.2 holds even for $\rho = 0$, i.e., it is applicable to exact DBSCAN as well.

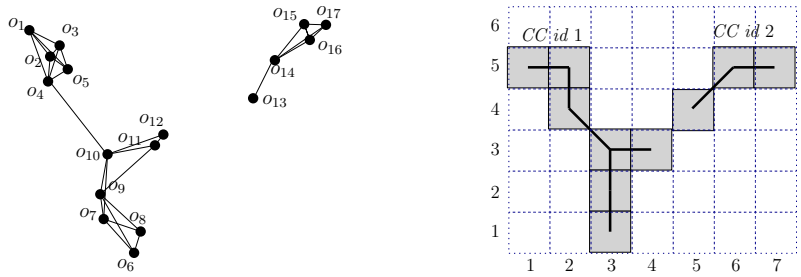
3.5.2 ρ-Double-Approximate DBSCAN and Sandwich Guarantee

The New Proposition. To enable both (fully-dynamic) update and query efficiency, we propose ρ-double-approximate DBSCAN, which takes the same parameters $\epsilon, \mathit{MinPts}$, and ρ as ρ-approximate DBSCAN. Whether a point $p \in P$ is a core point is now decided in a relaxed manner:

- Definitely a core point if $B(p, \epsilon)$ covers at least *MinPts* points of P.
- Definitely not a core point if $B(p, (1 + \rho)\epsilon)$ covers less than *MinPts* points of P.
- Don't care, otherwise.

A good example to illustrate this is point o_{13} in Figure 3.3a. Since $B(o_{13}, \epsilon)$ covers $2 < \mathit{MinPts} = 3$ points, o_{13} is not a core point under exact or ρ-approximate DBSCAN. Under double approximation, however, it falls into the don't-care case for $\rho = 0.5$, because $B(p, (1 + \rho)\epsilon)$ covers 7 points.

The clusters of ρ-double-approximate DBSCAN are defined by the same two-step approach of ρ-approximate DBSCAN (see Section 3.1), but with respect to the above core-point semantics. Namely, the only difference between the ρ-double-approximate and the ρ-approximate versions lies in the



(a) One possible ρ-double-approx core graph

(b) Grid graph

FIGURE 3.6: Illustration of ρ-double approximation

core point definition. For example, swaying o_{13} into a core point, Figure 3.6a shows the ρ-double-approximate core graph (defined precisely as the ρ-approximate version), while Figure 3.6b gives the corresponding grid graph.

Sandwich Guarantee. Recall that this is an attractive feature of ρ-approximate DBSCAN. Next, we prove that ρ-double-approximate DBSCAN provides just the same guarantee. Following the style of [29], we define:

- $\mathscr{C}_1$ as the set of clusters of exact DBSCAN with parameters $(\epsilon, \mathit{MinPts})$.
- $\mathscr{C}_2$ as the set of clusters of exact DBSCAN with parameters $(\epsilon(1+\rho), \mathit{MinPts})$.
- $\mathscr{C}$ as a set of clusters that is a legal result of ρ-double-approximate DBSCAN with parameters ϵ, *MinPts*, and ρ.

Then, the sandwich guarantee is:

Theorem 3.3. *The following statements are true: (i) For any cluster $C_1 \in \mathscr{C}_1$, there is a cluster $C \in \mathscr{C}$ such that $C_1 \subseteq C$, and (ii) for any cluster $C \in \mathscr{C}$, there is a cluster $C_2 \in \mathscr{C}_2$ such that $C \subseteq C_2$.*

The proof can be found in the appendix. Note that the theorem is purposely worded exactly the same as Theorem 2.3.

3.6 Fully Dynamic Algorithms

This section presents our algorithms for maintaining ρ-double-approximate DBSCAN clusters under both insertions and deletions (again, exact DBSCAN is captured with $\rho = 0$). We will achieve the purpose by instantiating the general framework in Section 3.3. The reader is reminded that the core-point definition has changed to the one in Section 3.5.2.

3.6.1 Approximate Bichromatic Close Pair

We now take a short break from clustering to discuss a computational geometry problem which we name the *approximate bichromatic close pair* (aBCP) *problem*. In this problem, we have two disjoint axis-parallel squares c_1, c_2 in $\mathbb{R}^d$. There is a set $S(c_1)$ of points in c_1, and a set $S(c_2)$ of points in c_2. The two sets are subject to insertions and deletions. Let ϵ and ρ be positive real values. We are asked to maintain a *witness pair* of (p_1^*, p_2^*) such that

- It may be an empty pair (i.e., p_1^* and p_2^* are null).
- If it is not empty, we must have $dist(p_1^*, p_2^*) \leq (1 + \rho)\epsilon$.
- The pair must not be empty, if there exist a point $p_1 \in S(c_1)$ and a point $p_2 \in S(c_2)$ such that $dist(p_1, p_2) \leq \epsilon$. Note that the pair does *not* have to be (p_1, p_2) though.

We have at our disposal an emptiness structure (as defined in Section 3.3.2) on each cell, so that an emptiness query (q, c) with $c = c_1$ or c_2 can be answered with cost $\tilde{O}(\tau)$ for some time function τ. The objective is to minimize the cost of (i) finding an initial witness pair, and (ii) maintaining the pair along with updates in $S(c_1)$ and $S(c_2)$.

Lemma 3.3. *For the aBCP problem, an initial witness pair can be found in $\tilde{O}(\tau \cdot \min\{|S(c_1)|, |S(c_2)|\})$ time. After that, the pair can be maintained by $\tilde{O}(\tau)$ amortized time when a point is inserted or deleted in $S(c_1)$ or $S(c_2)$.*

The proof can be found in the appendix.

3.6.2 Edges in the Grid Graph and aBCP

Returning to ρ-double-approximate DBSCAN clustering, let us recall that in the grid graph $\mathbb{G}$, if there is an edge between core cells c_1 and c_2, then the two cells must be ϵ-close. Such an edge may disappear/re-appear as the core points of $P(c_1)$ and $P(c_2)$ are deleted/inserted. Maintaining this edge can be regarded as an instance of the aBCP problem, where $S(c_1)$ is the set of core points in $P(c_1)$, and $S(c_2)$ is the set of core points in $P(c_2)$—the edge exists if and only if the witness pair is not empty!

We run a thread of the aBCP algorithm of Lemma 3.3 on every pair of ϵ-close core cells c_1 and c_2. Those threads will be referred to as the *aBCP instances* of c_1 (or c_2). Whenever the edge between c_1 and c_2 (re-)appears, we call $\mathit{EdgeInsert}(c_1, c_2)$ of the CC structure; whenever it disappears, we call $\mathit{EdgeRemove}(c_1, c_2)$.

3.6.3 The Core-Status Structure

Given a point q, an *approximate range count query* returns an integer k that falls between $|B(q, \epsilon)|$ and $|B(q, (1+\rho)\epsilon)|$. The query can be answered in $\tilde{O}(1)$ time by a structure that can be updated in $\tilde{O}(1)$ time per insertion and deletion [53]. Under the relaxed core-point definition of ρ-double-approximation, whether a point $p \in P$ is a core point can be decided directly by issuing such a query with p. If the query returns k, we declare p a core point if and only if $k \geq \mathit{MinPts}$.

Leveraging this fact, next we describe how to explicitly update the core-status of all points in P along with insertions and deletions:

- To insert a point p_{new} in cell c_{new}, we first check whether p_{new} itself is a core point. Remember that the insertion may turn some existing non-core points into core points. To identify such points, we look at each of the $O(1)$ ϵ-close sparse cells c of c_{new}. Simply check all the points $p \in P(c)$ to see if p is currently a core point.

- The deletion of a point p_{old} from cell c_{old} may turn some existing core points into non-core points. Following the same idea in insertion, we look at every ϵ-close sparse cell c of c_{old}, and check all the points $p \in P(c)$ for their current core status.

3.6.4 GUM

When a point p_{core} (say, in cell c_{core}) has turned into a core point, we check whether c_{core} is already in $\mathbb{V}$:

- If so, simply insert p_{core} into every aBCP instance (Lemma 3.3) of c_{core}—as explained in Section 3.6.2, this properly maintains the edges of c_{core}.
- Otherwise, it must hold that $|P(c_{core})| \leq MinPts = O(1)$. We add c_{core} to $\mathbb{V}$. Then, for every ϵ-close core cell c of c_{core}, decide whether to create an edge between c and c_{core} by using the algorithm of Lemma 3.3 to find an initial witness pair (thereby starting the aBCP instance on c_{core} and c).

Consider now the scenario where a core point p in cell c_{core} has turned into a non-core point. If c_{core} is still a core cell, we remove p from all the aBCP instances of c_{core}. Otherwise, we simply remove c_{core} from $\mathbb{V}$, and destroy all its aBCP instances.

3.6.5 Performance Guarantees

Utilizing the best CC and emptiness structures under the fully-dynamic scheme, we prove in the appendix our last main result:

Theorem 3.4. *For any fixed dimensionality d and fixed constant $\rho > 0$, there is a fully-dynamic ρ-double-approximate DBSCAN algorithm that processes each insertion and deletion in $\tilde{O}(1)$ amortized time, and answers a C-group-by query Q in $\tilde{O}(|Q|)$ time.*

The same update and query efficiency can also be achieved in 2D space for exact DBSCAN.

3.7 Experiments

Section 3.7.1 describes the setup of our empirical evaluation. Then, Sections 3.7.2 and 3.7.3 report the results on semi-dynamic and fully-dynamic algorithms, respectively.

3.7.1 Setup

Workload. We evaluate a clustering algorithm by its efficiency in processing a *workload*, which is a mixed sequence of updates and queries. Each update or query is collectively referred to as an *operation*. A workload is characterized by several parameters:

- N: the total number of updates.
- $\%_{ins}$: the percentage of insertions. In other words, the workload has $N \cdot \%_{ins}$ insertions, and $N(1 - \%_{ins})$ deletions. This parameter is fixed to 1 in semi-dynamic scenarios.
- f_{qry}: the query frequency, which is an integer controlling how often a C-group-by query is issued.

The production of a workload involves 3 steps, as explained below.

Step 1: Insertions. The sequence of insertions is obtained by first generating a "static dataset" of $I = N \cdot \%_{ins}$ points, and then, randomly permuting these points (i.e., if a point stands at the i-th position, it is the i-th inserted). We generate static datasets whose clusters are the outcome of a "random walk with restart", as was the approach suggested in [29], and will be reviewed shortly. Note that the random permutation mentioned earlier allows the clusters to form up even at an early stage of the workload.

The *data space* is a d-dimensional square that has range $[0, 10^5]$ on each dimension. A static dataset is created using the *seed spreader* technique in Section 2.6, which generates around 10 clusters and $0.0001 \cdot I$ noise points as follows. First, place a spreader at a random location p of the data space. At each *time tick*, the spreader adds to the dataset a point that is uniformly distributed in $B(p, 25)$. Whenever the spreader has generated 100 points while stationed at the same p, it is forced to move towards a random direction by a distance of 50. Finally, with probability $10/(0.9999I)$, the spreader "restarts" by jumping to another random location of the data space. Regardless of whether a restart happens, the current time tick finishes, and the next one starts. The spreader works for $0.9999I$ time ticks (thus producing $0.9999I$ points). After that, $0.0001 \cdot I$ random points are added to the dataset as noise.

Step 2: Deletions. First, append to the insertion sequence $D = N - I$ deletion *tokens*, where each token is simply a "place-holder" into which we will later fill a concrete point to delete. Then, randomly

permute the resulting sequence (which has length N). Check whether the permutation is *bad*, namely, if any of its prefixes has more tokens than insertions. If so, we attempt another random permutation until a *good* one is obtained.

Now we have a good sequence of insertions and deletion tokens. To fill in the tokens, scan down the sequence, and add each inserted point into S, until coming across the first token. Select a random point in S as the one deleted by the token, and then remove the point from S. The scan continues in this fashion until all the tokens have been filled.

Step 3: Queries. We simply insert a C-group-by query after every f_{qry} updates in the sequence. Recall that the query specifies a parameter Q, which is generated as follows. Let S be the set of "alive" points that have been inserted, but not yet deleted before the query. We first decide the value of $|Q|$ by choosing an integer uniformly at random from $[2, 100]$. Then, Q is populated by random sampling $|Q|$ points from S without replacement.

DBSCAN Algorithms. Our experimentation examined:

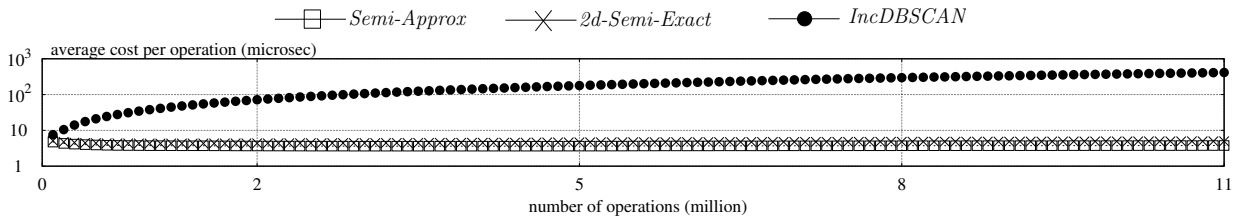
- *IncDBSCAN [26]*: the state-of-the-art dynamic algorithm for exact DBSCAN, as reviewed in Section 3.2.
- *2d-Semi-Exact*: our semi-dynamic algorithm in Theorem 3.1 for exact DBSCAN in 2D space.
- *Semi-Approx*: our semi-dynamic algorithm in Theorem 3.1 for ρ-approximate DBSCAN in d-dimensional space with $d \geq 2$.
- *2d-Full-Exact*: our fully-dynamic algorithm in Theorem 3.4 for exact DBSCAN in 2D space.
- *Double-Approx*: our fully-dynamic algorithm in Theorem 3.4 for ρ-double-approximate DBSCAN in d-dimensional space with $d \geq 2$.

All the algorithms were implemented in C++, and compiled with gcc version 4.8.4.

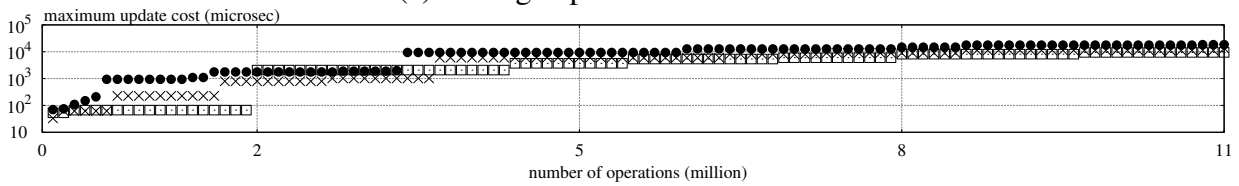
Parameters and Machine. We fixed N to 10 million, namely, each workload contains this number of updates. The value of *MinPts* in all the DBSCAN variants was 10. The value of ρ in the approximate variants was set to 0.001, under which ρ-double-approximate DBSCAN is required to return precisely the same clusters as ρ-approximate DBSCAN.

parameter	value
d	$2, 3, 5, 7$
ϵ	$\mathbf{50d}$, $100d, 200d, 400d, 800d$
$\%_{ins}$	$\frac{2}{3}, \frac{4}{5}, \mathbf{5/6}, \frac{8}{9}, \frac{10}{11}$
f_{qry}	$0.01N, 0.02N, 0.03N, ..., \mathbf{0.1N}$

TABLE 3.1: Variable parameter values (defaults in bolds)



(a) Average operation cost vs. time



(b) Max update cost vs. time

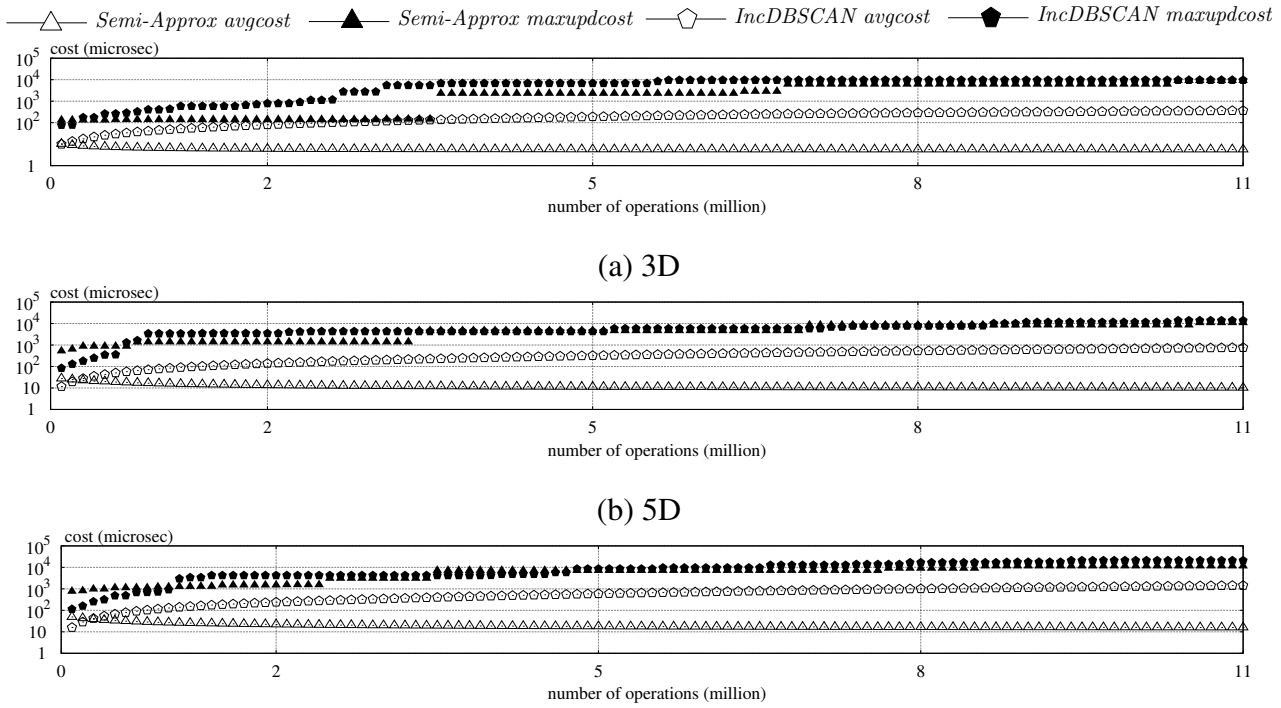
FIGURE 3.7: Performance of semi-dynamic algorithms in 2D

The other parameters were varied in different experiments. Their values are as shown in Table 3.1; unless otherwise stated, a parameter was set to its default as shown in bolds. Note that $\%_{ins} = 5/6$ indicates on average 1 deletion every 5 insertions.

Finally, all the experiments were run on a machine equipped with an Intel Core i7-6700 CPU @ 3.40GHz × 8 and 16GB memory. The operating system was Linux (Ubuntu 14.04.1).

3.7.2 Semi-Dynamic Results

This subsection will focus on insertion-only workloads. Consider executing an algorithm on such a workload. We define the algorithm's *average cost* as a function of time: $avgcost(t) = \frac{1}{t}\sum_{i=1}^{t} cost[i]$, where $cost[i]$ is the overhead of the algorithm in processing the i-th operation of the workload. Similarly, define the algorithm's *max update cost* as: $maxupdcost(t) = \max_{i=1}^{x} updcost[i]$, where (i) x is the



(a) 3D

(b) 5D

(c) 7D

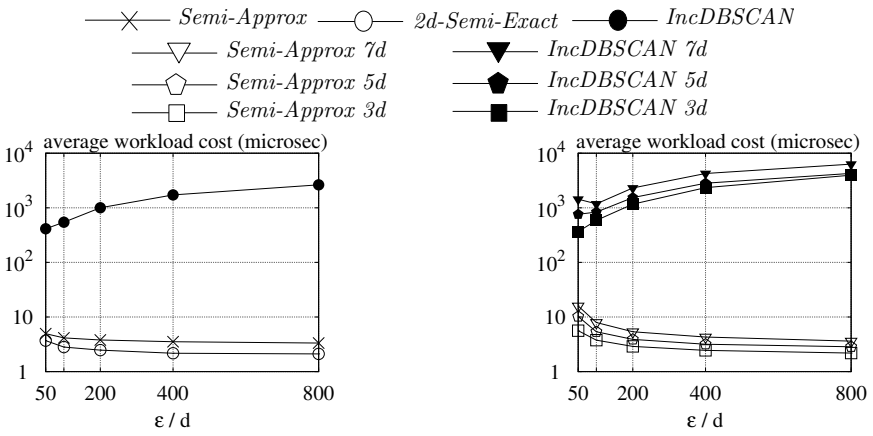
FIGURE 3.8: Performance of semi-dynamic algorithms in $d \geq 3$ dimensions

number of updates by the end of the t-th operation, and (ii) $updcost[i]$ is the overhead of the algorithm for the i-th update. Notice that query time is registered in *avgcost* but not *maxupdcost*.

Focusing on 2D space, Figure 3.7a plots the average cost of *IncDBSCAN*, *2d-Semi-Exact*, and *Semi-Approx*, whereas Figure 3.7b plots their max update cost. *2d-Semi-Exact* and *Semi-Approx* finished the workload significantly faster than *IncDBSCAN*, achieving an improvement of two orders of magnitude! Moreover, while the average cost of *IncDBSCAN* deteriorated continuously, the performance of *2d-Semi-Exact* and *Semi-Approx* remained stable throughout the workload. This is expected because *IncDBSCAN* must perform a range query per insertion (see Section 3.2), which tends to retrieve more data points as time progresses. Our solutions do not suffer from this drawback.

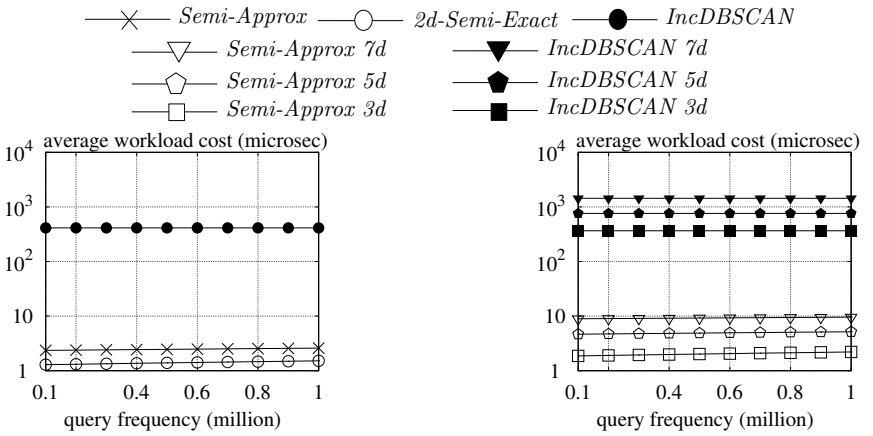
Turning to 3D space—where the competing methods are *IncDBSCAN* and *Semi-Approx*— Figure 3.8a compares their average cost and max update cost simultaneously. Figures 3.8b and 3.8c present the same results for $d = 5$ and 7, respectively. In all dimensionalities, *Semi-Approx* consistently outperformed *IncDBSCAN* by a wide margin even in logarithmic scale.

Interestingly, all the methods exhibited similar behavior when it comes to the *maxupdcost* metric.



(a) $d = 2$ (b) $d = 3, 5, 7$

FIGURE 3.9: Semi-dynamic performance vs. ϵ

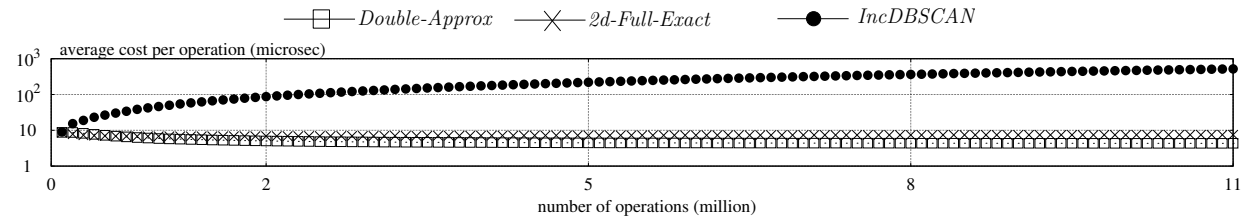


(a) $d = 2$ (b) $d = 3, 5, 7$

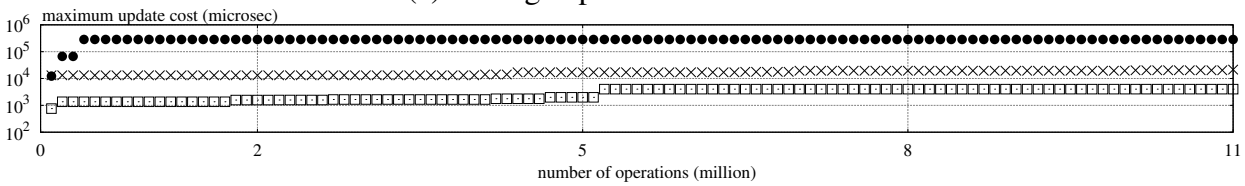
FIGURE 3.10: Semi-dynamic performance vs. f_{qry}

We will return to this issue later when we discuss the fully dynamic scenario, where contrasting phenomena will be observed.

We define an algorithm's *average workload cost* as *avgcost*(W), where W is the total number of operations in the workload of concern. The next experiment demonstrates the effect of ϵ on the cost of cluster maintenance—as discussed in Chapter 2 and [6], an algorithm of density-based clustering should be able to find clusters at different *granularities* of ϵ. Figure 3.9 shows the average workload cost of each applicable method as a function of ϵ, for $d = 2, 3, 5, 7$, respectively. It is evident that IncDBSCAN became prohibitively expensive as ϵ increases. On the other hand, our solutions actually performed even better for larger ϵ! This is not surprising because a greater ϵ actually *reduces* the



(a) Average operation cost vs. time



(b) Max update cost vs. time

FIGURE 3.11: Performance of fully-dynamic algorithms in 2D

number of edges in the grid graph, which in turn leads to substantial cost savings.

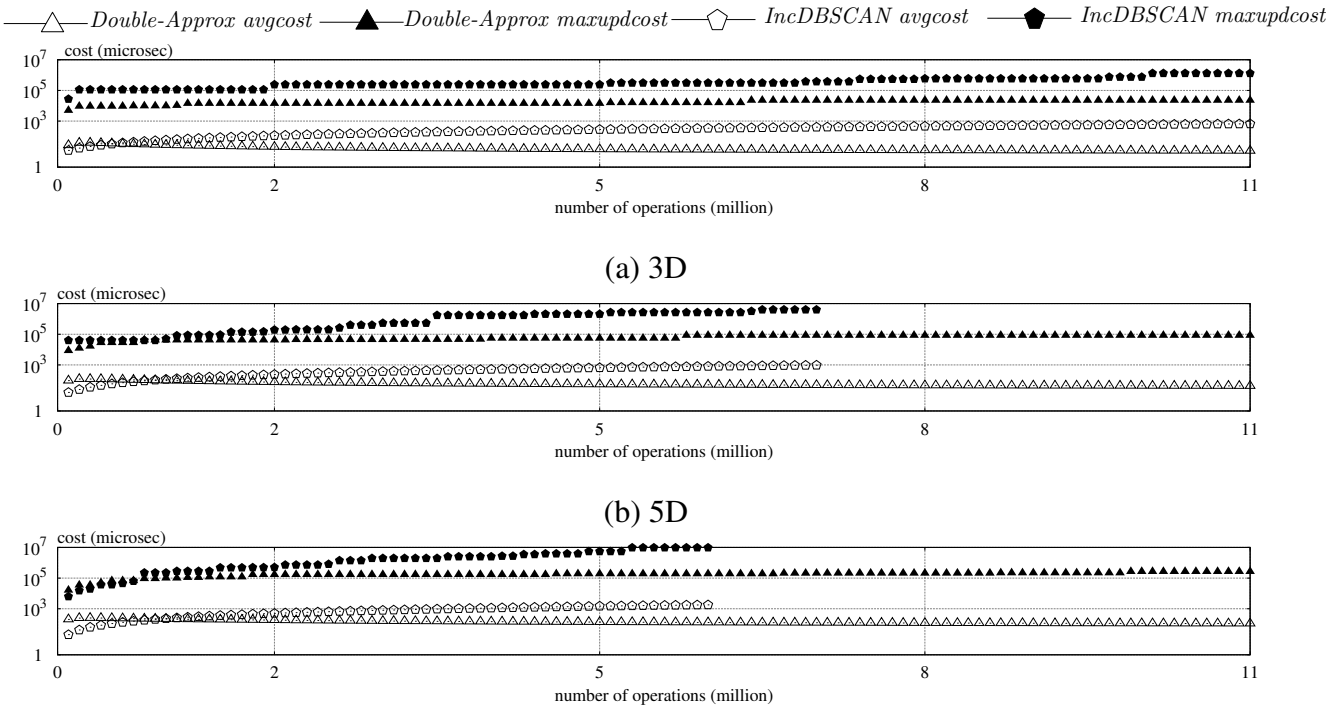
We conclude this subsection by giving the average workload cost of all methods as a function of f_{qry} in Figure 3.10. In general, query cost is negligible compared to update overhead.

3.7.3 Fully-Dynamic Results

We now proceed to evaluate the algorithms that can handle both insertions and deletions. Our strategy is similar to that in the previous subsection. *Average cost* and *max update cost* are defined in the same way as before, except that operations/updates obviously also include deletions here.

Figure 3.11 shows the results in experiments corresponding to those in Figure 3.7, with respect to *IncDBSCAN*, *2d-Full-Exact*, and *Double-Approx*. Similarly, Figure 3.12 corresponds to Figure 3.8, with respect to *IncDBSCAN* and *Double-Approx*. As before, our solutions were two orders of magnitude faster than *IncDBSCAN* in average cost. What is new, however, is that they also improved *IncDBSCAN* by nearly 10 times in max update cost as well!

What has triggered the separation in *maxupdcost*? The hardness of deletions! Recall from Section 3.2 that *IncDBSCAN* requires only one range query (to find the seed objects) in an insertion, whereas in a deletion, it demands multiple—actually perhaps many—such queries to perform BFS. This stands in sharp contrast to *Double-Approx*, which completely gets rid of BFS by novel ideas, in particular, deploying an aBCP algorithm (Lemma 3.3) to convert cluster maintenance to updating the



(a) 3D

(b) 5D

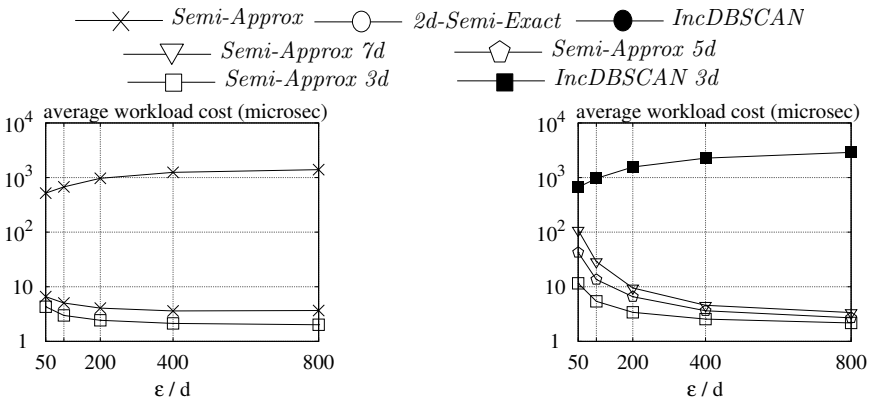
(c) 7D

FIGURE 3.12: Performance of fully-dynamic algorithms in $d \geq 3$ dimensions

CCs of the grid graph (which has only $O(n)$ edges). In *all scenarios*, our algorithms ensured processing an update in less than 0.1 seconds! The reader may have noticed that *IncDBSCAN* did not finish the 5D and 7D workloads. Indeed, we terminated it after 3 hours when its deficiencies had become apparent.

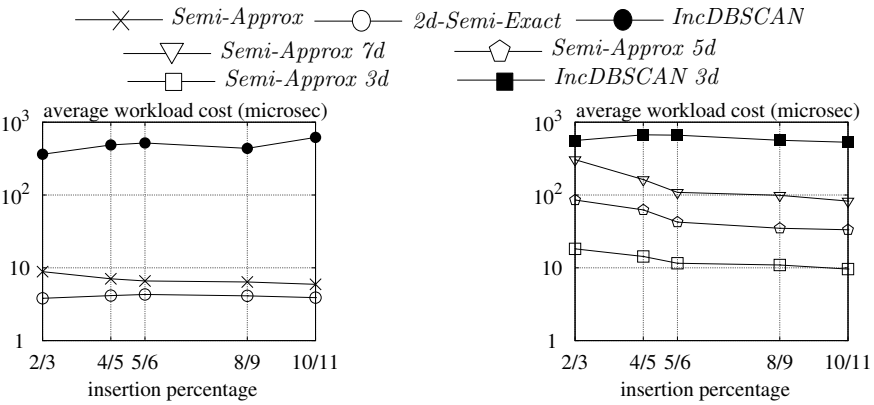
Figure 3.13 presents the results that are the counterparts of Figure 3.9, confirming that *IncDBSCAN* is essentially inapplicable for large ϵ. Note, again, that this method has no results for $d = 5$ and 7.

The last set of experiments inspected the average workload cost of these algorithms as the insertion percentage increased from 2/3 to 10/11. The results are reported in Figure 3.14. In general, the efficiency of each method improved as insertions accounted for a higher percent of the workload. Our new algorithms were the clear winners in all situations.



(a) $d = 2$ (b) $d = 3, 5, 7$

FIGURE 3.13: Fully-dynamic performance vs. ϵ



(a) $d = 2$ (b) $d = 3, 5, 7$

FIGURE 3.14: Fully-dynamic performance vs. $\%_{ins}$

3.8 Summary

This chapter has presented a systematic study on dynamic density based clustering under the theme of DBSCAN. Our findings reveal considerable new insight into the characteristics of the topic, by providing a complete picture of the computational hardness in various update schemes. Perhaps the most surprising result is that ρ-approximate DBSCAN, which was proposed to address the worst-case computational intractability of exact DBSCAN, suffers from the same hardness when both insertions and deletions are allowed. We have also shown how to eliminate the issue elegantly with a tiny relaxation, which has led to the development of ρ-double-approximate DBSCAN. Our algorithmic contributions

involve a suite of new algorithms that achieve near-constant update time in cluster maintenance essentially in all the update schemes where this is possible. The practical efficiency of our solutions has also been confirmed with extensive experiments.

3.9 Appendix

3.9.1 Proof of Theorem 3.1

We will prove the theorem first for ρ-approximate DBSCAN, and then for 2D exact DBSCAN.

Implementing the CC-structure as the *union-find* structure of Tarjan [65], we can support both the *EdgeInsert* and *CC-Id* operations in $\tilde{O}(1)$ time amortized. For the emptiness structure of every core cell, we can use the *approximate nearest neighbor* structure of Arya et al. [12], which answers an emptyness query in $\tilde{O}(1)$ time, and can be updated in $\tilde{O}(1)$ time.

Next, we prove that the algorithm processes n insertions in $\tilde{O}(n)$ time, that is, $\tilde{O}(1)$ amortized time per insertion:

- In the core-status structure, Step 1 takes $\tilde{O}(1)$ time per insertion by resorting to a standard dictionary-search structure (e.g., a binary search tree) on the non-empty cells.
- The total cost of Step 2 for the whole update sequence is $O(n)$. To see this, notice that a cell c is involved in this step only if it is an ϵ-close cell of c_{new}. Hence, with respect to the same c_{new}, c can be involved only $MinPts = O(1)$ times (after which c_{new} becomes a core cell, and will not require Step 2 again). As c has $O(1)$ ϵ-close cells, the total number of times that c is involved for *all* the c_{new} is $O(1)$.
- The same reasoning also explains that, the total cost incurred by the execution of the paragraph below Step 2 is $O(n)$ in the whole algorithm.
- In GUM, Steps 1, 1.1, and 1.2 can insert $O(n)$ edges in total, and therefore, entails $\tilde{O}(n)$ cost overall.

The query algorithm performs only $O(|Q|)$ *CC-Id* operations, and therefore, requires only $\tilde{O}(|Q|)$ time.

The above proof holds verbatim also for 2D exact DBSCAN, with the only difference that the structure of [12] should be replaced by the *2D nearest neighbor structure* of Chan [18].

3.9.2 Proof of Lemma 3.1

Suppose that A is an algorithm settling USEC-LS in $T(n)$ time. We can solve the USEC problem on a set P of n points (each red or blue) using divide and conquer as follows. Divide P using a plane ℓ orthogonal to dimension 1 into P_1 and P_2 each of which has $n/2$ points. Then, we recursively solve the USEC problem on P_1, and do the same on P_2. If either of these sub-problems returns "yes" (i.e., a red point within distance 1 from a blue point), we return "yes" immediately.

If both sub-problems return "no", we run A twice to solve two instances of USEC-LS. Divide P_1 into the set P_{1red} of red points, and the set P_{1blue} of blue points. Let P_{2red} and P_{2blue} be defined similarly with respect to P_2. The first USEC-LS instance is defined on P_{1red} and P_{2blue}, whereas the second on P_{1blue} and P_{2red}. If either instance returns "yes", we return "yes"; otherwise, we return "no".

Denote by $f(n)$ the running time of our USEC algorithm. The above description shows that

$$f(n) \quad = \quad 2f(n/2) + 2T(n)$$

with $f(n) = O(1)$ when $n = 1$. It is rudimentary to verify that when $T(n) = o(n^{4/3})$, then $f(n) = o(n^{4/3})$.

3.9.3 Proof of Theorem 3.3

Let G_ϵ be the core graph of $(\epsilon, MinPts)$ exact DBSCAN, and $G_{(1+\rho)\epsilon}$ be the core graph of $((1+\rho)\epsilon, MinPts)$ exact DBSCAN. Also, denote by $G_{\epsilon,\rho}$ the ρ-approximate core graph of ρ-double-approximate DBSCAN with the same ϵ and $MinPts$. A useful observation is that every edge in G_ϵ exists in $G_{\epsilon,\rho}$, and likewise, every edge in $G_{\epsilon,\rho}$ exists in $G_{(1+\rho)\epsilon}$.

Proof of Statement (i). Consider an arbitrary core point p_1 in C_1. Let C be the (only) cluster in $\mathscr{C}$ that contains p_1. Next, we will prove that $C_1 \subseteq C$.

Denote by S_ϵ the CC of G_ϵ containing p_1, and by $S_{\epsilon,\rho}$ the CC of $G_{\epsilon,\rho}$ containing p_1. Clearly, $S_\epsilon \subseteq S_{\epsilon,\rho}$. This means that all the core points of C_1 also belong to C.

Consider now an arbitrary non-core point p_2 in C_1. There must exist a core point $p_3 \in C_1$ such

that p_3 is covered by $B(p_2, \epsilon)$. Since $p_3 \in S_\epsilon \subseteq S_{\epsilon,\rho}$, we know that p_2 must also have been assigned to the cluster of $S_{\epsilon,\rho}$, namely, C.

Proof of Statement (ii). Consider an arbitrary core point p_1 in C. Let C_2 be the (only) cluster in $\mathscr{C}_2$ that contains p_1. Next, we will prove that $C \subseteq C_2$.

Denote by $S_{\epsilon,\rho}$ the CC of $G_{\epsilon,\rho}$ containing p_1, and by $S_{(1+\rho)\epsilon}$ the CC of $G_{(1+\rho)\epsilon}$ containing p_1. Clearly, $S_{\epsilon,\rho} \subseteq S_{(1+\rho)\epsilon}$. This means that all the core points of C also belong to C_2.

Consider now an arbitrary non-core point p_2 in C. There must exist a core point $p_3 \in C$ such that p_3 is covered by $B(p_2, (1+\rho)\epsilon)$. Since $p_3 \in S_{\epsilon,\rho} \subseteq S_{(1+\rho)\epsilon}$, we know that p_2 must also have been assigned to the cluster of $S_{(1+\rho)\epsilon}$, namely, C_2.

3.9.4 Proof of Lemma 3.3

Finding the Initial Pair. Suppose, without loss of generality, that $|S(c_1)| \leq |S(c_2)|$. For every point $p_1^* \in S(c_1)$, we run an emptiness query $\mathit{empty}(p_1^*, c_2)$. If the query returns 1 with a proof point p_2^*, we have found a witness pair (p_1^*, p_2^*), and hence, can terminate immediately. The cost is clearly that of $\tilde{O}(|S(c_1)|)$ emptiness queries.

We prove that the algorithm is correct. This is obvious if it finds a pair. Consider, instead, that it does not, in which case we are wrong only if there is a pair $(p_1, p_2) \in S(c_1) \times S(c_2)$ such that $\mathit{dist}(p_1, p_2) \leq \epsilon$. However, this means that $\mathit{empty}(p_1, c_2)$ must return 1, thus contradicting the fact that we did not found a pair.

Maintaining the Pair. We store in a list L the points that have been subsequently inserted in $S(c_1) \cup S(c_2)$ (the point ordering can be arbitrary). Each point of L will be *de-listed*—the meaning of which will be clear shortly—once, after which the point is removed from L. Furthermore, we enforce the rule that, if the witness pair is empty, L must be $\emptyset$.

Basic Operation: De-listing. This operation can only be performed when the witness pair is empty—it will attempt to find such a pair by issuing one emptiness query. For this purpose, the operation starts by removing the first point p from L; assume, without loss of generality, that $p \in S(c_1)$. It then issues $\mathit{empty}(p, c_2)$. If the query returns 1 with a proof point $p' \in S(c_2)$, (p, p') is taken as the witness pair. Otherwise, the witness pair remains empty.

Insertion. Consider that a point p has been inserted in $S(c_1)$ (the case with $S(c_2)$ is symmetric). Append p to L. If the witness pair is not empty, we do nothing else. Otherwise, p must now be the only element in L, in which case we perform a de-listing and finish.

Deletion. Consider that a point p has been deleted from $S(c_1)$ (a symmetric algorithm works for $S(c_2)$). Remove from L the entry of p (if found). If the witness pair (p_1^*, p_2^*) is not empty and $p \neq p_1^*$, the deletion does not affect the pair; and we are done. Otherwise, we proceed as follows:

1. Issue $\mathit{empty}(p_2^*, c_1)$. If it returns 1 with a proof point p', set (p', p_2^*) as the new witness pair, and return.
2. Otherwise, do the following until L is empty or the algorithm decides to return:
   2.1 Perform a de-listing.
   2.2 If the de-listing finds a witness pair, return.
3. (Now L is empty) set the witness pair to empty.

Correctness. Our algorithm is always correct if it finds a witness pair Let us look at the case where it does not. This is wrong only if there exists a pair $(p_1, p_2) \in S(c_1) \times S(c_2)$ such that $\mathit{dist}(p_1, p_2) \leq \epsilon$. At least one of p_1, p_2 must have been inserted after the initial pair was found. Without loss of generality, assume that p_2 is the one; and if both are, then assume that p_2 was de-listed after p_1.

Consider the moment when our algorithm de-listed p_2 from L. Since p_1 was present in $S(c_1)$, the query $\mathit{empty}(p_2, c_1)$ we issued must have returned a proof point p'. The witness (p', p_2) must have disappeared because p' was deleted. But in this case, our algorithm would immediately issue another $\mathit{empty}(p_2, c_1)$, which (again because p_1 was present in $S(c_1)$) must have returned another proof point. The situation repeats itself, with the consequence that we must be holding a witness pair, thus creating a contradiction.

Efficiency. Clearly, the total number of emptiness queries is at most the number of point insertions and deletions in $S(c_1) \cup S(c_2)$. This concludes the proof of the lemma.

Remark: No Materialization of L. At first glance, it may seem that a point of $S(c_1)$ (or $S(c_2)$) needs to be duplicated in L. Such duplication is space consuming if c_1 is involved in many instances of aBCP

simultaneously. In fact, L does not need to be materialized, and instead can be represented using only $O(1)$ memory.

Let us store the points of $S(c_1)$ in a list, sorted by insertion order. We can de-list these points by the sorted order, so that at any moment the points not yet de-listed—namely, those in L—constitute a suffix of the list. The suffix can be identified by remembering only the first point in the suffix, which only needs a single pointer. The same also holds for $S(c_2)$. Thus, two pointers suffice for L. To de-list a point, simply pick the point referenced by either pointer, and then shift the pointer down one position.

It is now evident that, no matter how many instances of aBCP c_1 is involved in, $S(c_1)$ is stored just once, by keeping one pointer for each instance.

3.9.5 Proof of Theorem 3.4

We prove only the update efficiency because the C-group-by query time is obvious. We will consider first ρ-approximate DBSCAN, and then 2D exact DBSCAN.

Implementing the CC-structure as the structure of Holm et al. [39], we can support *EdgeInsert*, *EdgeRemove*, and *CC-Id* all in $\tilde{O}(1)$ amortized time. For the emptiness structure, we can still use the approximate nearest neighbor structure of [12], which answers an emptyness query in $\tilde{O}(1)$ time, and can be updated in $\tilde{O}(1)$ time per insertion and deletion.

Let us now analyze the update cost of the core-status structure. Consider first the insertion of a point p_{new}. Let c_{new} be the cell of p_{new}. In the worst case, we will check all the $O(1)$ ϵ-close sparse cells c of c_{new}. As c has at most $MinPts = O(1)$ points, we need at most $O(1)$ approximate range count queries, whose total overhead is $\tilde{O}(1)$. An analogous argument shows that each deletion entails $\tilde{O}(1)$ time.

To account for the cost of GUM, we analyze how many of the following events may happen during an insertion/deletion on P:

- J_1: the number of aBCP instances created;
- J_2: the number of aBCP instances destroyed;
- K: the number of aBCP insertions/deletions.

An insertion on P can turn at most $O(1)$ points into core points—as mentioned, they must be in c_{new} or the $O(1)$ ϵ-close sparse cells of c_{new}, while each of these cells has at most $MinPts = O(1)$ points. As c has at most $O(1)$ aBCP instances, a new core point in a cell c can trigger at most $O(1)$ new aBCP instances and $O(1)$ aBCP insertions. Similarly, a deletion on P can destroy $O(1)$ aBCP instances and trigger $O(1)$ aBCP core deletions. We thus conclude that J_1, J_2 and K are all bounded by $O(1)$.

The initialization of an aBCP instance takes $\tilde{O}(1)$ time as it requires $O(1)$ emptiness queries by Lemma 3.3 and the fact that c_{new} has $O(1)$ points. Destroying an aBCP instance also takes only $O(1)$ time because it requires only discarding two pointers (see the remark in the proof of Lemma 3.3). Furthermore, by Lemma 3.3, the cost of the aBCP algorithm is proportional to K, now that each emptiness query takes $\tau = \tilde{O}(1)$ time. Thus, J_1, J_2, K all bounded by $O(1)$ indicates that GUM entails $\tilde{O}(1)$ amortized cost in each update.

Finally, the above discussion shows that an update can add/remove $O(1)$ edges of $\mathbb{G}$. Therefore, the cost from the CC-structure is $\tilde{O}(1)$ time amortized per update. We thus conclude the whole proof for ρ-double-approximate DBSCAN.

The above proof holds verbatim also for 2D exact DBSCAN, with the only difference that the structure of [12] should be replaced by the 2D nearest neighbor structure of Chan [18].

Chapter 4

External Density-Based Clustering and Multi-Dimensional Grid Graphs

4.1 Prelinmilary

Firstly, let us review some basic concepts.

External Memory (EM) Computational Model. In EM model, the machine is equipped with M words of (internal) memory, and a disk that has been formatted into *blocks*, each of which has B words. The values of M and B satisfy $M \geq 2B$. An *I/O* either reads a block of data from the disk into memory, or writes B words of memory into a disk block. The *cost* of an algorithm is measured in the number of I/Os performed. Define function $sort(n) = \Theta((n/B)\log_{M/B}(n/B))$, which is the I/O complexity of sorting n elements [4].

Multi-Dimensioanl Grid Graphs. Recall that a d-dimensional grid graph is an undirected graph $G = (V, E)$ with two properties:

- Each vertex $v \in V$ is a distinct point in space $\mathbb{N}^d$, where $\mathbb{N}$ represents the set of integers.
- If E has an edge between $v_1, v_2 \in V$, then (i) the two vertices are distinct (i.e., no self-loops), and (ii) they differ by at most 1 in their coordinates on *every* dimension.

See Figure 1.3 for two illustrative examples. For $d = O(1)$, such a graph is *sparse*, namely, $|E| = O(|V|)$, because each vertex can have a degree at most $3^d = O(1)$.

Vertex Separator. Given a positive integer $r \leq |V|$, a set $S \subseteq V$ is an *r-separator* of a d-dimensional grid graph $G = (V, E)$ if it satisfies:

- $|S| = O(|V|/r^{1/d})$
- Removing the vertices of S disconnects G into $h = O(|V|/r)$ subgraphs $G_i = (V_i, E_i)$ for $i \in [1, h]$ such that
  - $|V_i| = O(r)$;
  - No vertex of V_i is adjacent to any vertex of V_j, if $i \neq j$.
  - The vertices of V_i are adjacent to $O(r^{1-1/d})$ vertices of S (if a vertex $v \in V_i$ is adjacent to some vertex in S, v is said to be a *boundary vertex* of G_i).

Such separators are known [51, 63] to exist for any $r \in [1, |V|]$. Of special importance are M-separators, which are crucial for several fundamental graph problems, as described shortly.

4.2 Main Results

A primary contribution of this chapter is an algorithm for computing such a separator I/O-efficiently:

Theorem 4.1. *Let $G = (V, E)$ be a d-dimensional grid graph for any constant d. We can compute an M-separator, as well as the corresponding subgraphs $G_1, ..., G_h$ as stated earlier, in $O(sort(|V|))$ I/Os.*

Our proof of the theorem uses ideas different from those of [51, 63][1]. Interestingly, as a side product, our proof presents a new type of r-separators that can be obtained by a recursive binary orthogonal partitioning of $\mathbb{N}^d$, i.e., split $\mathbb{N}^d$ using an axis-parallel plane, and then recur in the "data spaces" on each side of the plane. All the vertices on all the cutting planes constitute our separator S.

[1]The r-separators of [51,63] are constructed by repetitively partitioning a d-dimensional grid graph with "surface cuts". Specifically, each cut is performed with a closed d-dimensional surface (which is a sphere in [51] and an axis-parallel rectangle in [63]). All vertices near the surface are added to the separator, while the process is carried out recursively inside and outside the surface, respectively. It remains a challenging open problem to compute such M-separators in $O(sort(|V|))$ I/Os.

Our algorithm of Theorem 4.1 works for *all* values of M, B satisfying $M \geq 2B$. This is a notable property, because of which Theorem 4.1 yields an improved algorithm for the *single source shortest path* (SSSP) problem on 2D grid graphs:

Corollary 4.1. *The SSSP problem on a 2D grid graph $G = (V, E)$ can be settled in $O(|V|/\sqrt{M} + sort(|V|))$ I/Os.*

Previously, the best I/O-efficient SSSP algorithm [36] on 2D grid graphs relies on the planar separator algorithm of [46], which is efficient only under the *tall cache* assumption $M = \Omega(B^2)$. Specifically, with that assumption, the algorithm of [36] matches the performance guarantee in Corollary 4.1. For $M = o(B^2)$, however, the I/O-complexity of [36] becomes $O((|V|/\sqrt{M}) \cdot \log_M |V|)$, which we strictly improve.

Corollary 4.1 also implies a new algorithm for *breadth first search* (BFS) on 2D grid graphs:

Corollary 4.2. *We can perform BFS on a 2D grid graph $G = (V, E)$ in $O(|V|/\sqrt{M} + sort(|V|))$ I/Os.*

The corollary nicely bridges the previous state of the art, which runs either the SSSP algorithm of [36], or the best BFS algorithm [49] for general graphs that performs $O(|V|/\sqrt{B} + sort(|V|))$ I/Os. In particular, the corollary improves the winner of those two algorithms when M is between $\omega(B)$ and $o(B^2)$.

Our next result stems from an unexpected source. It has been stated [67, 73] that the *connected components* (CCs) of a 2D grid graph $G = (V, E)$ can be computed in $O(sort(|V|))$ I/Os. This is based on the belief that a 2D grid graph has the property of being *sparse under edge contractions*. Specifically, an *edge contraction* removes an edge between vertices v_1, v_2 from G, combines v_1, v_2 into a single vertex v, replaces every edge adjacent to v_1 or v_2 with an edge adjacent to v, and finally removes duplicate edges thus produced—all these steps then create a new graph; see Figure 1.4. The aforementioned property says that, if one performs any sequence of edge contractions to obtain a resulting graph $G' = (V', E')$, G' must still be sparse, namely, $|E'| = O(|V'|)$. Surprising enough, the belief—perhaps too intuitive—seemed to have been taken for granted, such that no proof has ever been documented.

We formally *disprove* this belief:

Theorem 4.2. *There exists a 2D grid graph that is not sparse under edge contractions.*

With the belief invalidated, the best existing deterministic algorithm for computing the CCs of a 2D grid graph requires an I/O complexity that is the minimum of $O(sort(|V|) \cdot \log\log B)$ [54] and $O(sort(|V|) \cdot \log(|V|/|M|))$ [37]. Equipped with Theorem 4.1, we improve this result:

Corollary 4.3. *The CCs of a d-dimensional grid graph $G = (V, E)$ with any constant d can be computed in $O(sort(|V|))$ I/Os.*

4.2.1 Main Application: Density-Based Clustering

Density-based clustering is an important problem in data mining (see popular textbooks [33,64]). The input consists of

- A real value $\epsilon > 0$, and
- A set P of n points in $\mathbb{R}^d$, where $\mathbb{R}$ denotes the set of real values, and the dimensionality d is a constant integer at least 2.

The input defines a *neighbor graph* $\mathcal{G}$ as follows. First, $\mathcal{G}$ has n vertices, each corresponding to a distinct point in P. Second, there is an edge between two vertices (a.k.a., points) p_1, p_2 if:

- $dist(p_1, p_2) \leq \epsilon$, where function *dist* gives the distance between p_1 and p_2.

The goal of the problem is to output the connected components of $\mathcal{G}$, each of which forms a *cluster*.

Figure 4.1 illustrates an example where the distance metric is L_∞ norm. Note that there can be $\Omega(n^2)$ edges in $\mathcal{G}$ (for simplicity, no edges are given in the example, but the square as shown should make it easy to imagine which edges are present). Thus, one should not hope to solve the problem efficiently by materializing all the edges.

Using our results on d-dimensional grid graph, we prove that the problem can be settled in near-linear I/Os, when *dist* is L_∞ norm:

Theorem 4.3. *The density-based clustering problem under L_∞ norm can be settled in*

- $O(sort(n))$ *I/Os for $d = 2$ and 3;*
- $O((n/B)\log^{d-2}_{M/B}(n/B))$ *I/Os for $d \geq 4$.*

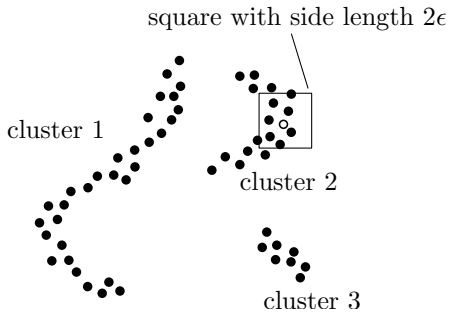


Figure 4.1: Density-based clustering. The square illustrates the value of ϵ (all the points in the square are within L_∞ distance ϵ from the white point), which determines the output of 3 clusters.

Why L_∞—especially when the "textbook metric" is L_2 norm? There are two primary reasons. First, the problem is known to be hard under L_2 norm: it demands $\Omega(n^{4/3})$ time to solve for $d \geq 3$ (by Theorem 2.1), unless the Hopcroft's problem [24] could be settled in $o(n^{4/3})$ time, which is commonly believed to be impossible [23, 24]. Consequently, L_2 norm is unlikely to admit an EM algorithm with near linear I/O complexity (otherwise, one could obtain an efficient RAM algorithm by setting M and B to constants). Theorem 4.3, therefore, separates L_∞ norm (and hence, also L_1 norm) from L_2 norm, subject to the Hopcroft-hard conjecture.

An interesting question arises: apart from $p = \infty$ and 1, is there any other L_p norm admitting an efficient algorithm for density-based clustering? Intuitively, L_∞ and L_1 norms are easy due to the orthogonal nature of a "sphere" (i.e., it is a hyper-square). Do fast algorithms exist when a sphere becomes "curvy", as is the case for almost all other values of $p > 1$? Unfortunately, till this day, the question still remains open.

However, the question disappears as long as mild approximation can be accepted. This is the second reason for studying L_∞ norm: the precise L_∞ clusters (produced by Theorem 4.3) directly serve as an $O(1)$-approximation for any L_p norm with $p > 0$, according to the formulation in Section 2.3, which we explain below.

Our approximate notion—in essence—defined an *approximate neighbor graph* $\mathcal{G}'$ as follows. As in $\mathcal{G}$, the vertex set of $\mathcal{G}'$ is just P itself. Whether there should be an edge between two vertices p_1, p_2 is decided by:

- The edge definitely exists if $dist(p_1, p_2) \leq \epsilon$.
- The edge must not exist if $dist(p_1, p_2) > (1 + \rho)\epsilon$.

- Otherwise, don't care: the edge may or may not exist.

The objective of *ρ-approximate density-based clustering* is to compute the connected components of $\mathcal{G}'$. Note that $\mathcal{G}'$ is not unique because of the don't-care option. Leveraging this flexibility, we managed to give an $O(n)$-expected-time algorithm to settle the problem, even for L_2 norm. Notice that the original, precise, clustering problem essentially sets $\rho = 0$.

If two points p_1, p_2 have L_∞-distance x for any $p > 0$, then their L_p-distance must fall in $[x, x \cdot d^{1/p}]$. This implies that, given the same ϵ, the precise neighbor graph $\mathcal{G}$ under L_∞ is also an approximate neighbor graph $\mathcal{G}'$ under L_p with $1 + \rho = d^{1/p}$. Therefore, the L_∞ clusters obtained from Theorem 4.3 constitute an $O(1)$-approximate result under L_p norm for *all* $p > 0$ *simultaneously*.

As a remark, the above observation partially explains the phenomenon that density-based clustering often yields similar results in practice regardless of the L_p norm adopted.

4.2.2 Additional Applications: Terrains, Land Surfaces, and Road Networks

To our knowledge, in the database field, I/O-efficient algorithms on d-dimensional grid graphs first appeared in 1993, when Nodine et al. [55] studied *path blocking*, i.e., how to store the edges in disk blocks (perhaps with duplication) so that every path of length B is covered by a small number of blocks.

The subsequent research seemed to have focused on 2D grid graphs, largely motivated by the analysis of *terrains* [2, 8, 9], also known as *land surfaces* [22, 44, 60, 72]. Such analysis is enabled by remote sensing, which produces a function $f : \mathbb{R}^2 \to \mathbb{R}$ that maps every point on the earth's longitude-altitude plane to an elevation. To represent the function approximately, the plane is discretized into a grid, such that function values are stored only at the grid points. Real-world objects (such as roads, paths, waterways, etc.) are represented by edges that give rise to a 2D grid graph. The length of an edge (u, v) captures the 3D distance between the points $f(u)$ and $f(v)$. Solving fundamental problems such as BFS, SSSP, and CC on such graphs is important for *flow analysis* [8, 9], *nearest-neighbor queries* [60, 72], and *navigation* [44].

It would be reasonable to assume that a 2D grid graph serves as an accurate model of the road network in an urban area. In that case, our SSSP algorithm would find use in applications that require path planning. In fact, our results can be strengthened to provide better support of these applications.

Recall that, if an edge exists between two vertices v_1, v_2 in a grid graph, their coordinates must differ by at most $c = 1$ on every dimension. The value c can actually be relaxed to an arbitrary constant, without affecting the correctness of Theorem 4.3, and Corollaries 4.1-4.3. The resulting "generalized" 2D grid graph is expected to model an urban road network with enhanced accuracy.

4.2.3 Remarks on the Tall-Cache Assumption

We note that some success has been achieved by [9, 35] in eliminating the tall-cache assumption, but only on *restricted* 2D grid graphs where the x- and y-coordinates of all vertices are confined to the range $[1, \sqrt{|V|}]$. The results in this paper do not have such a limitation, and apply to vertices of arbitrary coordinates.

An algorithm was presented in [37] to perform depth-first search (DFS) on a 2D grid graph in $O(sort(|V|) \cdot \log(|V|/|M|))$ I/Os as long as $M \geq 2B$. It can therefore be used to compute CCs with the same cost. We did not manage to improve their bound on DFS, but as far as the CC problem is concerned, Corollary 4.3 is more superior in terms of both efficiency and applicability (by supporting all constant d).

With a few exceptions (e.g., the Google file system uses a block size of $B = 64$MB [30], rendering B^2 too large a size for main memory), the tall-cache assumption can be satisfied in many practical scenarios nowadays. Nevertheless, its final elimination has been a grand wish of the theory community for a long time. We feel that it makes a fundamental contribution to prove the possibility of computing an M-separator of a d-dimensional grid graph using near-linear I/Os without the assumption. This, for instance, allows us to obtain Theorem 4.3 without "any strings attached" on the memory size.

4.3 Orthogonal Separators

We now embark on the journey towards proving Theorem 4.1, which paves the foundation for all the other results of this paper. As mentioned earlier, the proposed r-separator is obtained by recursively splitting the data space $\mathbb{N}^d$ with an axis-parallel plane. In this section, we will focus on showing the *existence* of such an "orthogonal r-separator". Our algorithm for *computing* it, as is given in the next section, re-uses many of the ideas needed to prove the existence. Clarifying those ideas first will

simplify the presentation of that algorithm considerably.

Formally, we define an *orthogonal separator* of a d-dimensional grid graph $G = (V, E)$ as a subset S of V that can be obtained by the following recursive process:

1. Initially, $S = \emptyset$.
2. Find a plane π—called a *cutting plane*—that is perpendicular to one of the d dimensions. Add to S *all* the vertices of V on π.
3. Let S' be the set of vertices on the cutting plane. Removing S' (this includes deleting the edges incident on the vertices of S') disconnects G into two subgraphs G_1 and G_2. If necessary, perform Step 2 on G_1, G_2, or both.

It is worth pointing out that, if we remove all the cutting planes used in the above process, the remaining portion of the data space is divided into disjoint (d-dimensional) rectangles, each of which will be called a *residue rectangle*. The graph induced by the vertices of V in each residue rectangle is precisely a subgraph that results from deleting all the vertices in S from G. Those subgraphs are said to be *induced by removing* S.

If an r-separator is simultaneously an orthogonal separator, we call it an *orthogonal r-separator*. Neither of the r-separators in [51, 63] is an orthogonal r-separator. The objective of this section is to prove:

Lemma 4.1. *Any d-dimensional grid graph $G = (V, E)$ admits an orthogonal r-separator for any integer $r \in [1, |V|]$.*

4.3.1 Binary Separators

Recall that an r-separator can be *multi-way* because it may induce any number $h = O(|V|/r)$ of subgraphs. In this subsection, we prove the existence of an orthogonal separator with $h = 2$:

Lemma 4.2. *Let $G = (V, E)$ be a d-dimensional grid graph satisfying*

$$|V| \geq 2^d \cdot (2d + 1)^{d+1}.$$

There exists an orthogonal separator $S \subseteq V$ such that:

- $|S| \leq (2d+1)^{1/d} \cdot |V|^{1-1/d}$
- *Removing S induces subgraphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ with $|V_i| \geq |V|/(4d+2)$ for $i = 1, 2$.*

Given a point $p \in \mathbb{N}^d$, denote by $p[i]$ its coordinate on dimension $i \in [1, d]$. Given a vertex $v \in V$, an integer x, and a dimension i, v is *on the left of x on dimension i* if $v[i] < x$, and similarly, *on the right of x on dimension i* if $v[i] > x$. We define the *V-occupancy of x on dimension i* as the number of vertices $v \in V$ satisfying $v[i] = x$.

To find the S of Lemma 4.2, our strategy is to identify an integer x and a dimension i such that (i) the V-occupancy of x on dimension i is at most $(2d+1)^{1/d} \cdot |V|^{1-1/d}$, and (ii) there are at least $|V|/(4d+2)$ points on the left and right of x on dimension i, respectively. Choosing S as the set of vertices $v \in V$ with $v[i] = x$ will satisfy the requirements of Lemma 4.2. We will prove that such a pair of x and i definitely exists.

For each $j \in [1, d]$, define y_j to be the largest integer y such that V has at most $|V|/(2d+1)$ vertices on the left of y on dimension j, and similarly, z_j to be the smallest integer z such that V has at most $|V|/(2d+1)$ vertices on the right of z on dimension j. It must hold that $y_j \leq z_j$.

Consider the axis-parallel box whose projection on dimension $j \in [1, d]$ is $[y_j, z_j]$. By definition of y_j, z_j, the box must contain at least

$$|V|\left(1 - \frac{2d}{2d+1}\right) = |V| \cdot \frac{1}{2d+1}$$

vertices. This implies that the box must contain at least $|V|/(2d+1)$ points in $\mathbb{N}^d$, namely:

$$\prod_{j=1}^{d}(z_j - y_j + 1) \quad \geq \quad \frac{|V|}{2d+1}$$

Therefore, there is at least one j satisfying

$$z_j - y_j + 1 \quad \geq \quad \left(\frac{|V|}{2d+1}\right)^{1/d}.$$

Set i to this j. Since the box apparently contains at most $|V|$ vertices, there is one integer $x \in [y_i, z_i]$ such that the V-occupancy of x on dimension i is at most

$$\frac{|V|}{|V|^{1/d}/(2d+1)^{1/d}} \quad = \quad (2d+1)^{1/d} \cdot |V|^{1-1/d}.$$

We now argue that there must be at least $|V|/(4d+2)$ vertices on the left of x on dimension i. For this purpose, we distinguish two cases:

- $x = y_i$: By definition of y_i and x, the number of vertices on the left of x on dimension i must be at least

$$\frac{|V|}{2d+1} - (2d+1)^{1/d} \cdot |V|^{1-1/d}$$

  which is at least $|V|/(4d+2)$ for $|V| \geq 2^d(2d+1)^{d+1}$.

- $x > y_i$: By definition of y_i, there are at least $|V|/(2d+1)$ vertices whose coordinates on dimension i are at most y_i. All those vertices are on the left of x on dimension i.

A symmetric argument shows that at least $|V|/(4d+2)$ vertices are on the right of x on dimension i. This finishes the proof of Lemma 4.2.

4.3.2 Weak Multi-Way Separators

In this subsection, we prove:

Lemma 4.3. *Let $G = (V, E)$ be a d-dimensional grid graph. For any positive integer r satisfying*

$$2^d \cdot (2d+1)^{d+1} \leq r \tag{4.1}$$

there exists an orthogonal separator $S \subseteq V$ with $|S| = O(|V|/r^{1/d})$ such that removing S induces $O(|V|/r)$ subgraphs each of which has at most r vertices.

Note that the lemma does not necessarily yield an r-separator because each subgraph induced by the removal of S may not have a small number of boundary vertices.

Motivated by [28], we perform the binary split enabled by Lemma 4.2 recursively until every subgraph has at most r vertices. In total, Lemma 4.2 is applied $O(|V|/r)$ times (each with a cutting plane). Denote by S the union of the separators produced by all those applications (i.e., the vertices of V on the $O(|V|/r)$ cutting planes). We complete the proof of Lemma 4.3 with:

Lemma 4.4. $|S| = O(|V|/r^{1/d})$.

Proof. Define function $f(n)$ which gives the maximum possible $|S|$ when the original graph has $n = |V|$ vertices. If $\frac{r}{4d+2} \leq n \leq r$, $f(n) = 0$. Otherwise, Lemma 4.2 indicates

$$f(n) \leq \beta \cdot n^{1-1/d} + \max_{\alpha=\frac{1}{4d+2}}^{\frac{4d+1}{4d+2}} f(\alpha n) + f((1-\alpha)n)$$

where $\beta = (2d+1)^{1/d}$. It is rudimentary to verify that $f(n) = O(n/r^{1/d})$ for $n > r$. □

Remark. Define the *minimum bounding box* of G—denoted as $MBB(G)$—as the d-dimensional rectangle whose projection on dimension $i \in [1, d]$ is $[x_i, y_i]$, where x_i (y_i, resp.) is the smallest (largest, resp.) coordinate on this dimension of the vertices in G.

Consider any subgraph G' induced by removing the S of Lemma 4.3. The above proof implies a useful *bounding box property*: each boundary vertex of G' must be on the boundary faces of $MBB(G')$.

4.3.3 Binary Boundary Separators

This subsection presents a variant of Lemma 4.2, which is crucial for us to strengthen the separator S in Lemma 4.3 into an r-separator. We say that a d-dimensional grid graph $G = (V, E)$ is *r-colored* if

- $|V| \leq r$;
- Every vertex in V is colored either black or white;
- There are at least $r^{1-1/d}$ black vertices, all of which are on the boundary faces of $MBB(G)$.

We will show:

Lemma 4.5. *Let r be a positive integer such that*

$$(32d^3)^{d(d-1)} \leq r. \tag{4.2}$$

Let $G = (V, E)$ be an r-colored d-dimensional grid graph with b black vertices. There exists a set $S \subseteq V$ satisfying:

- $|S| \leq (4d^3)^{1/(d-1)} \cdot r^{1-1/d}$
- *Removing S induces subgraphs G_1, G_2 each of which has at least $b/(32d^3)$ black vertices.*

We prove the lemma with a careful execution of the strategy in Section 4.3.1. Since $MBB(G)$ has $2d$ faces, one of them contains at least $b/(2d)$ black vertices. Fix R to be this face, which is a $(d-1)$-dimensional rectangle. Assume, without loss of generality, that R is orthogonal to dimension d.

Given a coordinate x on dimension $i \in [1, d-1]$, we define the *black R-occupancy of x on dimension i* as the number of black vertices $v \in V$ satisfying (i) v is in R, and (ii) $v[i] = x$. The reader should distinguish this notion from "V-occupancy" as defined at the beginning of Section 4.3.

We proceed differently depending on:

- *Case 1:* There does not exist a pair of x and $k \in [1, d-1]$ such that the black R-occupancy of x on dimension k is at least $\frac{b}{2d}/(4d)$.
- *Case 2:* Such a pair of x and k exists. This case can happen only for $d \geq 3$.

Case 1. For each $j \in [1, d-1]$, define y_j to be the largest integer y such that R has at most $\frac{b}{2d}/(2d)$ black vertices on the left of y on dimension j, and similarly, z_j to be the smallest integer z such that R has at most $\frac{b}{2d}/(2d)$ black vertices on the right of z on dimension j. It must hold that $y_j \leq z_j$.

Consider the axis-parallel box in R whose projection on dimension $j \in [1, d-1]$ is $[y_j, z_j]$. By definition of y_j, z_j, the box must contain at least

$$\frac{b}{2d}\left(1 - \frac{2(d-1)}{2d}\right) = \frac{b}{2d} \cdot \frac{1}{d}$$

black vertices. Therefore, there is at least one dimension $j \in [1, d-1]$ on which the projection of the box covers at least

$$\left(\frac{b}{2d^2}\right)^{1/(d-1)}$$

coordinates. Set i to this j. Since the box apparently contains at most $|V| \leq r$ vertices, there is one integer $x \in [y_i, z_i]$ such that the V-occupancy of x on dimension i is at most

$$\frac{r}{b^{1/(d-1)}} \cdot (2d^2)^{1/(d-1)} \quad \leq \quad r^{1-1/d} \cdot (2d^2)^{1/(d-1)}$$

applying $b \geq r^{1-1/d}$.

By definition of Case 1, there must be at least $b/(8d^2)$ black vertices on the left and right of x on dimension i, respectively. Therefore, setting $S = \{v \in V \mid v[i] = x\}$ fulfills the requirements of Lemma 4.5.

Case 2. Without loss of generality, suppose that $k = d - 1$. Consider a "slice" of R at coordinate x on dimension k: $R' = \{p \in R \mid p[k] = x\}$, which is a $(d-2)$-dimensional rectangle. By definition of Case 2, R' contains at least $b/(8d^2)$ black vertices.

For each $j \in [1, d-2]$, define y_j to be the largest integer y such that R' has at most $\frac{b}{8d^2}/(2d)$ black vertices on the left of y on dimension j, and similarly, z_j to be the smallest integer z such that R' has at most $\frac{b}{8d^2}/(2d)$ black vertices on the right of z on dimension j. It must hold that $y_j \leq z_j$.

Consider the axis-parallel box in R'' whose projection on dimension $j \in [1, d-2]$ is $[y_j, z_j]$. The box must contain at least

$$\frac{b}{8d^2}\left(1 - \frac{2(d-2)}{2d}\right) = \frac{b}{4d^3}$$

black vertices. Hence, there is at least one dimension $j \in [1, d-2]$ on which the projection of the box covers at least

$$\left(\frac{b}{4d^3}\right)^{1/(d-2)}$$

coordinates. Set i to this j. There is one integer $x \in [y_i, z_i]$ such that the V-occupancy of x on dimension i is at most

$$\begin{aligned} \frac{r}{b^{1/(d-2)}} \cdot (4d^3)^{\frac{1}{d-2}} &\leq r^{\frac{d^2-3d+1}{d(d-2)}} \cdot (4d^3)^{\frac{1}{d-2}} \\ &< r^{1-1/d} \cdot (4d^3)^{\frac{1}{d-2}} \end{aligned} \tag{4.3}$$

where the first inequality applied $b \geq r^{1-1/d}$.

We choose S as the set of vertices $v \in V$ with $v[i] = x$. To show that there must be at least $b/(32d^3)$ black vertices on the left of x on dimension i, we distinguish several cases:

- $x = y_i$: By (4.3) and the definitions of y_i and x, the number of black vertices on the left of x on dimension i must be at least

  $$\frac{b}{16d^3} - r^{\frac{d^2-3d+1}{d(d-2)}} \cdot (4d^3)^{\frac{1}{d-2}}$$

  which is at least $b/(32d^3)$ when r satisfies (4.2).

- $x > y_i$: The definitions of y_i and x imply at least $b/(16d^3)$ black vertices on the left of x on dimension i.

A symmetric argument shows that there must be at least $b/(32d^3)$ black vertices on the right of x on dimension i. This completes the proof of Lemma 4.5.

4.3.4 Strong Multi-Way Separators

This subsection will complete the proof of Lemma 4.1.

Large r. Let us first consider the case where r satisfies (4.2), and hence, also (4.1). To find the desired separator S, we first apply Lemma 4.3 on G. Let S' be the separator output by this application, which induces $h' = O(|V|/r)$ subgraphs. Our algorithm in Section 4.3.2 ensures that each subgraph has at most r vertices. We say that a subgraph is *bad* if it has more than

$$33d^3 \cdot (4d^3)^{1/(d-1)} \cdot r^{1-1/d}$$

boundary vertices.

Motivated by [28], we deploy Lemma 4.5 to eliminate all the bad subgraphs. Specifically, given a bad subgraph G_{bad}, we color all its boundary vertices black, and the other vertices white, after which G_{bad} becomes r-colored. Apply Lemma 4.5 to split it into G_{bad1} and G_{bad2}. The application produces a binary separator, where all the vertices are added to S'. Recur on G_{bad1} (and similarly, G_{bad2}) if it is still bad. When there are no more bad subgraphs, the current S' becomes our final separator S. Call the above process the *elimination phase*.

Lemma 4.6. *Let G_{bad} be a bad subgraph with b boundary vertices. The elimination phase generates $O(b/r^{1-1/d})$ subgraphs from G_{bad}.*

Proof. Define function $f(n)$ which gives the maximum number of subgraphs that can be generated from G_{bad} when G_{bad} has n black vertices. Set $\beta = (4d^3)^{1/(d-1)}$.

If $n \leq 33d^3\beta r^{1-1/d}$, $f(n) = 1$. Otherwise, Lemma 4.5 indicates:

$$f(n) \leq \max_{\alpha=\frac{1}{32d^3}}^{1-\frac{1}{32d^3}} \Big(f(\alpha n + \beta r^{1-1/d}) + f((1-\alpha)n + \beta r^{1-1/d}) - 1 \Big).$$

It is rudimentary to verify that $f(n) = O(n/r^{1-1/d})$ for $n > 33d^3\beta r^{1-1/d}$. □

Let b_i ($1 \leq i \leq h'$) be the number of boundary vertices of the i-th subgraph produced from the application of Lemma 4.3 (i.e., right before the elimination phase starts). From Lemma 4.4 and the fact that each vertex in a d-dimensional grid graph has degree $O(1)$, we know

$$\sum_{i=1}^{h'} b_i = O(|V|/r^{1/d}).$$

Combining this with Lemma 4.6 shows that the elimination phase introduces at most

$$O\left(\frac{|V|}{r^{1/d}} \cdot \frac{1}{r^{1-1/d}}\right) = O(|V|/r)$$

new subgraphs. Therefore, in total there are $h' + O(|V|/r) = O(|V|/r)$ subgraphs at the end.

The above analysis also indicates that the elimination phase can apply Lemma 4.5 no more than $O(|V|/r)$ times, each of which adds $O(r^{1-1/d})$ vertices into S. Therefore, the final separator S has size at most

$$|S'| + O\left(\frac{|V|}{r} \cdot r^{1-1/d}\right) = O(|V|/r^{1/d}).$$

Small r. For $r < (32d^3)^{d(d-1)}$, it suffices to manually increase r to $(32d^3)^{d(d-1)}$, and return directly the separator S obtained with the above "large-r" algorithm. This is a valid orthogonal r-separator also for the original r due to the fact that d is a constant. We now conclude the proof of Lemma 4.1.

4.4 Computing a Separator I/O-Efficiently

This section will prove Theorem 4.1 by giving an algorithm to construct an M-separator. Our proof is essentially an efficient implementation of the strategy explained in Section 4.3 for finding an orthogonal M-separator. Recall that the strategy involves two phases: (i) Lemma 4.3, and (ii) the elimination phase in Section 4.3.4. The second phase, as far as algorithm design is concerned, is trivial. Every subgraph produced by the first phase has—by definition of M-separator—$O(M)$ edges, which can therefore be loaded into memory so that the algorithm in Section 4.3.4 runs with no extra I/Os. In other words, the second phase can be accomplished in only $O(|V|/B)$ I/Os.

Henceforth, we will focus exclusively on the first phase, assuming

$$M \geq 2^d \cdot (4d+2)^{2d} \cdot B. \tag{4.4}$$

Note that this assumption is made without loss of generality as long as d is a constant. It is folklore that, in general, any algorithm assuming $M \geq cB$ for any constant $c > 2$ can be adapted to work under $M \geq 2B$, with only a constant blowup in the I/O cost.

The construction algorithm of Lemma 4.3 recursively binary splits the input graph, until all the obtained subgraphs have at most M vertices. This process can be imagined as a *split tree*, where $G = (V, E)$ is the parent of $G_1 = (V_1, E_1)$, $G_2 = (V_2, E_2)$ if the splitting of G spawns G_1 and G_2. The split is *balanced* in the sense that both $|V_1|$ and $|V_2|$ are at least $|V|/(4d+2)$. Hence, the split tree has a height of $O(\log(|V|/M))$.

It should be rather rudimentary to implement the algorithm in $O((|V|/B)\log(|V|/M))$ I/Os—note that the logarithm base is 2—by implementing the splitting of $G = (V, E)$ in $O(|V|/B)$ I/Os. Our goal, as stated in Theorem 4.1, is to lower the cost by a factor of $\Theta(\log(M/B))$ to $O(sort(|V|))$.

We achieve the purpose by performing $2^{\Omega(\log(M/B))}$ splits on G still in $O(|V|/B)$ I/Os, thus effectively generating $\Omega(\log(M/B))$ levels of the split tree underneath G. This is reminiscent of the bulkloading algorithm of Agarwal et al. [1] for the kd-tree. However, unlike the kd-tree where a split involves mainly finding the median from a sorted list, the split process in our context is much more sophisticated. This renders our construction algorithm to be drastically different from that of [1].

4.4.1 One Split

Given a d-dimensional grid graph $G = (V, E)$ with $|V| > M$, we in this subsection describe an algorithm for performing one split on $G = (V, E)$ that finishes in *sublinear* I/Os, assuming that certain preprocessing has been done. This algorithm will play an essential role in our final solution.

Recall that, given a coordinate x on dimension $i \in [1, d]$, the V-occupancy of x is the number of vertices $v \in V$ with $v[i] = x$. We now extend this concept to an interval $\sigma = [x_1, x_2]$ on dimension i: the *average V-occupancy* of σ equals

$$\frac{|\{v \in V \mid v[i] \in \sigma\}|}{x_2 - x_1 + 1}.$$

Preprocessing Assumed. Prior to invoking the algorithm below, each dimension $i \in [1, d]$ should have been partitioned into at most s disjoint intervals—called *slabs*—where

$$s = (M/B)^{1/d}. \tag{4.5}$$

A slab σ of dimension i is said to *cover* a vertex $v \in V$ if $v[i] \in \sigma$. We call σ *heavy* if it covers more than $|V|/(4d + 2)$ vertices. Our algorithm demands an important *heavy-singleton property*:

If a slab $\sigma = [x_1, x_2]$ of any dimension is heavy, it must hold that $x_1 = x_2$, namely, σ is a *singleton slab* containing only a single coordinate.

All the slabs naturally define a d-dimensional *histogram* H. Specifically, H is the d-dimensional grid with at most s^d cells, each of which is a d-dimensional rectangle whose projection on dimension $i \in [1, d]$ is a slab on that dimension. For each cell ϕ of H, the following information should already be available:

- A *vertex count*, equal to the number of vertices $v \in V$ that ϕ contains (i.e, the point v falls in ϕ). Denote by $\phi(V)$ the set of these vertices.
- *d vertex lists*, where the i-th ($1 \leq i \leq d$) one sorts all the vertices of $\phi(V)$ by dimension i. This means that a vertex $v \in \phi(V)$ is duplicated d times. We store with each copy of v all its $O(1)$ adjacent edges.

All the vertex counts are kept in the memory. The sorted vertex lists in all the cells, on the other hand, occupy $O(s^d + |V|/B) = O(|V|/B)$ blocks in the disk.

Given a slab σ on any dimension, we denote by $\sigma(V)$ the set of vertices covered by σ. The vertex counts in H allow us to obtain $|\sigma(V)|$ precisely, and hence, the average V-occupancy of σ precisely, without any I/Os. Define

$$K = \max_{\text{non-singleton } \sigma} |\sigma(V)| \tag{4.6}$$

Note that the maximum ranges over all *non-singleton* slabs of *all* dimensions.

As in Section 4.3.1, our aim is to find a dimension i and a coordinate x such that (i) the V-occupancy of x is at most $(2d + 1)^{1/d}|V|^{1-1/d}$, and (ii) at least $|V|/(4d + 2)$ vertices are on the left and right of x on dimension i, respectively. Our algorithm will perform $O((M/B)^{1-1/d} + K/B)$ I/Os.

Algorithm. Suppose that the slabs on dimension i are numbered from left to right, i.e., the leftmost one is numbered 1, the next 2, and so on. For dimension $j \in [1, d]$, let y_j be the largest integer y such that at most $|V|/(2d + 1)$ points are covered by the slabs on this dimension whose numbers are *less than* y, and similarly, let z_j be the smallest integer z such that at most $|V|/(2d + 1)$ points are covered in the slabs on this dimension whose numbers are *greater than* z. It must hold that $y_j \leq z_j$.

Let R be the d-dimensional rectangle whose projection on dimension i is the union of the slabs numbered $y_j, y_j + 1, ..., z_j$. As R contains at least $|V|/(2d + 1)$ vertices, its projection on at least one dimension covers at least $(|V|/(2d + 1))^{1/d}$ coordinates. Fix i to be this dimension. Note that the projection of R on dimension i (i.e., an interval on the dimension) has an average V-occupancy of at most $(2d + 1)^{1/d}|V|^{1-1/d}$. Therefore, at least one of the slabs numbered $y_i, y_i + 1, ..., z_i$ on dimension i has an average V-occupancy at most $(2d + 1)^{1/d}|V|^{1-1/d}$. Let σ be this slab.

It thus follows that at least one coordinate x within σ has V-occupancy of at most $(2d+1)^{1/d}|V|^{1-1/d}$. If σ is a singleton slab, then x is the (only) coordinate contained in σ. Otherwise, to find such an x, we scan the vertices of $\sigma(V)$ in ascending order of their coordinates on dimension i. This can be achieved by merging the vertex lists of all the at most s^{d-1} cells in σ—more specifically, the lists sorted by dimension i. The scan takes

$$O(s^{d-1} + |\sigma(V)|/B) = O((M/B)^{1-1/d} + K/B)$$

I/Os, by keeping a memory block as the reading buffer for each cell in σ.

To prove the algorithm's correctness, we first argue that at least $|V|/(4d + 2)$ vertices are on the left of x on dimension i. By $|V| > M$ and (4.4), it holds that

$$(2d + 1)^{1/d}|V|^{1-1/d} \leq \frac{|V|}{4d + 2}.$$

This implies that σ—the slab which x comes from—cannot be heavy. Therefore, by definition of y_i, there must be at least

$$\frac{|V|}{2d + 1} - \frac{|V|}{4d + 2} = \frac{|V|}{4d + 2}$$

vertices in the slabs of dimension i whose numbers are less than y_i. All those vertices are on the left of x on dimension i. A symmetric argument shows that at least $|V|/(4d + 2)$ vertices are on the right of x on dimension i.

4.4.2 $2^{\Omega(\log(M/B))}$ Splits

Let $G = (V, E)$ be a d-dimensional grid graph with $|V| > M$ that is stored as follows. First, V is duplicated in d lists, where the i-th ($1 \leq i \leq d$) one sorts all the vertices $v \in V$ by dimension i. Second, each copy of v is associated with all the $O(1)$ edges adjacent to v.

In this section, we present an algorithm that achieves the following purpose in $O(|V|/B)$ I/Os: recursively split G using the *one-split algorithm* of Section 4.4.1 such that, in each resulting subgraph, the number of vertices is at most

$$\max\left\{M, O\left(\frac{|V|}{2^{\Omega(\log(M/B))}}\right)\right\}.$$

but at least $M/(4d + 2)$.

Define

$$t = (M/B)^{1/d}/2$$

Partition each dimension $i \in [1, d]$ into disjoint intervals (a.k.a., *slabs*) $\sigma = [x_1, x_2]$ satisfying two conditions:

- If σ covers more than $|V|/t$ vertices, it must hold that $x_1 = x_2$, i.e., σ is a singleton slab.
- If σ is not the rightmost slab on this dimension, more than $|V|/t$ vertices $v \in V$ satisfy $v[i] \in [x_1, x_2 + 1]$.

These conditions can be understood intuitively as follows. To create a slab of dimension i starting at coordinate x_1, one should set its right endpoint x_2 ($\geq x_1$) as large as possible, provided that the slab still covers at most $|V|/t$ points. But such an x_2 does not exist if x_1 itself already has a V-occupation of more than $|V|/t$; in this case, create a singleton slab containing only x_1. It is easy to obtain these slabs in $O(|V|/B)$ I/Os from the vertex list of V sorted by dimension i.

Proposition 4.1. *Each dimension has less than* $2t$ *slabs.*

Proof. The union of any two consecutive slabs must cover more than $|V|/t$ vertices. Consider the following pairs of consecutive slabs: (1st, 2nd), (3rd, 4th), ..., leaving out possibly the rightmost slab. A vertex is covered by the union of at most one such pair. Therefore, there can be at most

$$\left\lfloor \frac{|V|}{\lfloor |V|/t \rfloor + 1} \right\rfloor \leq t - 1$$

pairs, making the number of slabs at most $2(t-1)+1 = 2t-1$. □

Proposition 4.2. *On any dimension, a non-singleton slab covers at most $|V|/t$ vertices.*

Proof. Follows immediately from how the slabs are constructed. □

Construct the histogram H on G as defined in Section 4.4.1. This can be accomplished in $O(|V|/B)$ I/Os. To understand this, observe that the total number of cells in the histogram is at most $(2t)^d \leq M/B$, which allows us to allocate one memory block to each cell. Using these blocks as writing buffers, we can create all the cells' vertex lists on a dimension by scanning the vertex list of V on the same dimension only once.

Recursive One-Splits. We invoke the one-split algorithm on G—noticing that all its preprocessing requirements have been fulfilled—which returns a coordinate x and dimension i. The I/O cost is $O((M/B)^{1-1/d} + |V|/(tB))$ I/Os, because Proposition 4.2 ensures that the value of K in (4.6) is at most $|V|/t$.

The pair x and i defines a separator S, which consists of all the vertices $v \in V$ with $v[i] = x$. Removing S splits G into $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$. Let σ be the slab on dimension i containing x. Extracting S from σ requires $O(1 + |S|/B + |\sigma(V)|/B) = O(|S|/B + |V|/(tB))$ I/Os.

We will then recursively apply the one-split algorithm on G_1 and G_2, respectively, before which, however, we need to prepare their histograms H_1, H_2. If σ is singleton, H_1 and H_2 can be obtained trivially with no I/Os: H_1 (or H_2, resp.) includes all the cells of H on the left (or right, resp.) of x on dimension i.

If σ is non-singleton, each cell ϕ in σ needs to be split (at x on dimension i) into ϕ_1, ϕ_2, whose information is not readily available yet. We can produce the information of all such ϕ_1, ϕ_2 by inspecting each ϕ as follows:

1. Assign the vertices in ϕ—if not in S—to ϕ_1 or ϕ_2.

2. Prepare the d sorted lists of ϕ_1 and ϕ_2 by splitting the corresponding lists of ϕ.

As there are $O((M/B)^{1-1/d})$ cells in σ, the above steps finish in $O((M/B)^{1-1/d} + |V|/(tB))$ I/Os.

If $|V_1| > M$ (or $|V_2| > M$), we now apply the one-split algorithm on G_1 (or G_2, resp.)—descending one level from G in the split tree—which is recursively processed in the same manner. The recursion

ends after we have moved

$$\ell \;=\; \lfloor (\log_{4d+2} t) - 1 \rfloor \tag{4.7}$$

levels down in the split tree from G. It can be verified that $\ell \geq 1$ (applying (4.4)) and $2^\ell = O(t)$.

Recall that the one-split algorithm requires the heavy-singleton property to hold. We now prove that this property is always satisfied during the recursion. Let $G' = (V', E')$ be a graph processed by the one-split algorithm. Since G' is at most ℓ levels down in the split tree from G, we know (by the fact that each split is balanced) that

$$|V'| \;\geq\; \frac{|V|}{(4d+2)^\ell}$$

which together with (4.7) shows

$$\frac{|V'|}{4d+2} \geq \frac{|V|}{t}.$$

Therefore, a heavy slab σ' of any dimension for G' must contain more than $|V|/t$ vertices. On the other hand, σ' must be within a slab σ defined for G, which thus also needs to cover more than $|V|/t$ vertices. By our construction, σ must be singleton, and therefore, so must σ'.

Finally, it is worth pointing out that each split will generate $O((M/B)^{1-1/d})$ cells, and hence, demands the storage of this many extra vertex counts in memory. This is fine because the total number of vertex counts after $2^\ell = O(t)$ splits is $O((M/B)^{1-1/d} \cdot t) = O(M/B)$.

Bounding the Total Cost. The one-split algorithm is invoked at most 2^ℓ times in total. By the above analysis, the overall I/O cost is

$$\begin{aligned} & O\left(\frac{|S|}{B} + \left(\left(\frac{M}{B}\right)^{1-1/d} + \frac{|V|}{tB}\right) \cdot 2^\ell\right) \\ = \;& O\left(\frac{|S|}{B} + \left(\left(\frac{M}{B}\right)^{1-1/d} + \frac{|V|}{tB}\right) \cdot t\right) \\ = \;& O\left(\frac{|S|}{B} + \frac{M}{B} + \frac{|V|}{B}\right) = O\left(\frac{|V|}{B}\right) \end{aligned}$$

utilizing two facts: (i) every vertex v contributes to the $|S|/B$ term at most once—once included in a separator, v is removed from further consideration in the rest of the recursion, and (ii) a non-singleton slab of any histogram throughout the recursion is within a non-singleton slab of H (the histogram of G), and hence, covers no more than $|V|/t$ vertices by Proposition 4.2.

4.4.3 The Overall Algorithm

We are ready to describe how to compute an M-separator on a d-dimensional grid graph $G = (V, E)$ in $O(sort(|V|))$ I/Os which, according to the discussion at the beginning of Section 4.4, will complete the proof of Theorem 4.1.

First, sort the vertices of V d times, each by a different dimension, thus generating d sorted lists of V. We associate with each copy of v all its $O(1)$ edges. The production of these lists takes $O(sort(|V|))$ I/Os.

We now invoke the algorithm of Section 4.4.2 on G. For each subgraph $G' = (V', E')$ thus obtained, we materialize it into d sorted lists, where the i-th ($1 \leq i \leq d$) one sorts V' by dimension i, ensuring that each copy of a vertex is stored along with its $O(1)$ edges. This can be done in $O(|V'|/B)$ I/Os as follows. Recall that the algorithm maintains a histogram of at most M/B cells. By allocating a memory block as the writing buffer for each cell, we can generate the sorted list of V' on a dimension by one synchronous scan of the corresponding vertex lists of all cells for the same dimension. The cost is $O(M/B + |V'|/B) = O(|V'|/B)$ because $|V'| \geq M/(4d + 2)$.

Finally, if $|V'| > M$, we recursively apply the algorithm of Section 4.4.2 on G', noticing that the preprocessing requirements of the algorithm have been fulfilled on G'.

Now we prove that the total cost of the whole algorithm is $O(sort(|V|))$. One application of the algorithm of Section 4.4.2 on a graph $G' = (V', E')$ costs $O(|V'|/B)$ I/Os, i.e., each vertex in V' is amortized $O(1/B)$ I/Os. The vertex can be charged $O(\log_{M/B}(|V|/M))$ times, adding up to $O(sort(|V|))$ I/Os overall for all vertices.

4.5 Density-Based Clustering

In this section, we attend to the density-based clustering problem that motivated this work in the first place. Besides establishing Theorem 4.3, the proposed algorithm, as presented in Section 4.5.1, demonstrates an elegant geometric application of our results on d-dimensional grid graphs. The algorithm leverages Corollary 4.3, which we prove in Section 4.5.2.

4.5.1 Proof of Theorem 4.3

Maxima/Minima. Let us first mention a relevant result on the *maxima/minima problem*. Let P be a set of n distinct points in $\mathbb{R}^d$. A point $p_1 \in P$ *dominates* another $p_2 \in P$ if $p_1[i] \geq p_2[i]$ for all dimensions $i \in [1, d]$—recall that $p[i]$ denotes the coordinate of p on dimension i. The *maxima set* of P is the set of points $p \in P$ such that p is not dominated by any point in P. Conversely, the *minima set* of P is the set of points $p \in P$ such that p does not dominate any point in P. A point in the maxima or minima set is called a *maximal* or *minimal* point of P, respectively.

In EM, both the maxima and minima sets of P can be found in $O(sort(n))$ I/Os for $d = 2, 3$, and $O((n/B)\log^{d-2}_{M/B}(n/B))$ I/Os for $d \geq 4$ [62].

Clustering. Recall that the input to the problem involves (i) a real value $\epsilon > 0$, and (ii) a set P of n points in $\mathbb{R}^d$.

We impose an arbitrary regular grid $\mathbb{G}$ on $\mathbb{R}^d$, where each cell is an axis-orthogonal d-dimensional square with side length ϵ. Assign each point $p \in P$ to the cell of $\mathbb{G}$ containing it; if p happens to be on the boundary of multiple cells, assign it to all those (at most $2^d = O(1)$) cells. For each cell ϕ of $\mathbb{G}$, denote by $\phi(P)$ the set of points assigned to ϕ. If $\phi(P)$ is non-empty, ϕ is a *non-empty* cell. There can be obviously at most n non-empty cells, all of which can be easily found in $O(sort(n))$ I/Os.

It is clear that two points assigned to the same cell ϕ must belong to the same cluster. This allows us to "sparsify" P by computing the clusters at the cell level. For this purpose, we define a graph $G = (V, E)$ as follows:

- Each vertex V corresponds to a non-empty cell.
- Two different vertices (a.k.a. cells) $\phi_1, \phi_2 \in V$ are connected by an edge if and only if there exists a point $p_1 \in \phi_1(P)$ and a point $p_2 \in \phi_2(P)$ such that $dist(p_1, p_2) \leq \epsilon$.

We will explain later how to generate G efficiently, but a crucial observation at the moment is that G is a d-dimensional grid graph. To see this, embed the grid $\mathbb{G}$ naturally in a space $\mathbb{N}^d$ with one-one mapping between the cells of $\mathbb{G}$ and the points of $\mathbb{N}^d$. It is easy to verify that there can be an edge between two non-empty cells ϕ_1 and ϕ_2 *only if* their coordinates differ by at most 1 on every dimension.

Now we can solve the clustering problem easily by computing the CCs (connected components) of G. For each connected component, collect the union of $\phi(P)$ for each vertex (i.e., cell) ϕ therein. The union corresponds to precisely one final cluster of P. Corollary 4.3 permits us to achieve the purpose in $O(sort(n))$ I/Os.

We now discuss the generation of G. We say that a non-empty cell ϕ is *sparse* if $|\phi(P)| \leq B$; otherwise, ϕ is *dense*. Also, another cell ϕ' is a *neighbor* of ϕ if the two cells differ in their coordinates by at most 1 on every dimension (in the embedded space $\mathbb{N}^d$). Note that a cell has less than $3^d = O(1)$ neighbors.

The non-empty neighbors of all non-empty cells can be produced in $O(sort(n))$ I/Os as follows. For each non-empty cell ϕ, generate $3^d - 1$ pairs (ϕ, ϕ') for all its neighbors ϕ', regardless of whether ϕ' is empty. Put all such pairs together, and join them with the list of non-empty cells to eliminate all such pairs (ϕ, ϕ') where ϕ' is empty. The non-empty neighbors of each non-empty cell can then be easily derived from the remaining pairs.

Define the *neighbor point set* of a non-empty cell ϕ—denoted as N_ϕ—to be the *multi-set* that gathers the $\phi'(P)$ of all non-empty neighbors ϕ' of ϕ. We emphasize that N_ϕ is a multi set because a point may belong to the $\phi'(P)$ of several ϕ'. Since we already have the non-empty neighbors of all non-empty cells, it is easy to create the N_ϕ of all ϕ in $O(sort(n))$ I/Os. In doing so, we also ensure that the points of N_ϕ are sorted by which $\phi'(P)$ they come from. Note that all the neighbor point sets occupy $O(n/B)$ blocks in total.

Given a non-empty cell ϕ, we now elaborate on how to obtain its edges in G. This is easy if ϕ is sparse, in which case we can achieve the purpose by simply loading the entire $\phi(P)$ in memory and scanning through N_ϕ. The I/O cost of doing so for all the sparse cells is therefore $O(n/B)$.

Consider instead ϕ to be a dense cell. We examine every non-empty neighbor ϕ' of ϕ, in ascending order of the appearance of $\phi'(P)$ in N_ϕ. Let us assume—without loss of generality due to symmetry—that the coordinate of ϕ is at most that of ϕ' on every dimension of $\mathbb{N}^d$. We determine whether there is an edge in G between ϕ and ϕ' by solving three d-dimensional maxima/minima problems, each on no more than $|\phi(P)| + |\phi'(P)|$ points:

1. Find the maxima set Σ_1 of $\phi(P)$, and the minima set Σ_2 of $\phi'(P)$.

2. Construct a set Π of points as follows: (i) add all points of Σ_1 to Π, and (ii) for each point $p \in \Sigma_2$, decrease its coordinate by ϵ on every dimension, and add the resulting point to Π.

3. If Π contains two points with the same coordinates, declare *yes* (i.e., there is an edge between ϕ and ϕ'), and finish. This implies the existence of $p_1 \in \Sigma_1$ and $p_2 \in \Sigma_2$ with $p_1[i] + \epsilon = p_2[i]$ for all $i \in [1, d]$.

4. Find the minima set Σ_3 of Π.

5. If any point of Σ_1 is absent from Σ_3, declare *yes*; otherwise, declare *no*.

To see the correctness, suppose first that there should be an edge. Then, there must be a maximal point p_1 of $\phi(P)$ and a minimal point p_2 of $\phi'(P)$ that have L_∞ distance at most ϵ. Let p_2' be the point shifted from p_2 after decreasing its coordinate by ϵ on all dimensions; p_2' either is dominated by p_1 or coincides with p_1. It follows that p_1 will not appear in Σ_3 if the execution comes to Step 5, prompting the algorithm to output yes. Similarly, one can show that if there should not be an edge, the algorithm definitely reports no.

For $d = 2, 3$, running the above algorithm for all dense cells ϕ and their non-empty neighbors ϕ' entails I/O cost (applying the aforementioned result of [62] on the minima/maxima problem)

$$\sum_{\text{dense } \phi, \text{ neighbor } \phi'} O(sort(|\phi(V)| + |\phi'(V)|)) = O(sort(n))$$

using the fact that each cell ϕ can be a neighbor of less than $3^d = O(1)$ dense cells. This proves the first bullet of Theorem 4.3, while the second bullet can be established in the same fashion using the $d \geq 4$ result of [62], all subject to Corollary 4.3, which we prove right away in the next subsection.

4.5.2 Proof of Corollary 4.3

Given a d-dimensional grid graph $G = (V, E)$, apply Theorem 4.1 to compute an M-separator S, as well as its induced subgraphs $G_1 = (V_1, E_1)$, ..., $G_h = (V_h, E_h)$ where $h = O(|V|/M)$. For each $i \in [1, h]$, define G_i^+ as an *extended subgraph* whose

- Vertices include (i) those in V_i and (ii) the separator vertices (i.e., vertices in S) that are adjacent to any boundary vertices of G_i. There are $O(M^{1-1/d})$ such separator vertices, i.e., same order as the number of boundary vertices.

- Edges include (i) those in E_i, and (ii) the edges between the boundary vertices of G_i and separator vertices. G_i^+ has $O(M)$ edges in total.

All these graphs can be generated in $O(sort(|V|))$ I/Os.

Construct a graph $G' = (V', E')$ with $V' = S$ as follows. First, E' includes all the edges in E among the separator vertices of S. $O(|S|) = O(|V|/M^{1/d})$ edges are added this way. Second, we add to E' additional edges that reflect the connectivity of the separator vertices within each extended subgraph. Specifically, for each $i \in [1, h]$, load into memory G_i^+ and compute its CCs. If a CC contains $x \geq 2$ separator vertices, add to E' $x - 1$ edges that form a tree connecting those vertices. The total number of edges inserted to E' in the second step is $O((|V|/M) \cdot M^{1-1/d}) = O(|V|/M^{1/d})$. Both steps can be done in $O(|V|/B)$ I/Os.

Invoke the algorithm of [54][2] to compute the CCs of G' in

$$\begin{aligned} & O\left(\frac{|E'|}{B} \log_{M/B} \frac{|E'|}{B} \cdot \log\log B\right) \\ = \; & O\left(\frac{|V|}{BM^{1/d}} \log_{M/B} \frac{|V|}{B} \cdot \log\log B\right) = O(sort(|V|)) \end{aligned}$$

I/Os. Label the vertices of V' (i.e., S) so that vertices in a CC receive the same unique label.

Finally, for $i \in [1, h]$, load G_i^+ into memory again. For each non-separator vertex v_i, give it the same label as any separator vertex that v_i can reach in G_i^+. If no such separator vertex exists, v_i is in a CC that does not involve any separator vertex; all the vertices in the CC are thus given a new label. Doing so for all i entails $O(|V|/B)$ extra I/Os.

4.6 New Results on 2D Grid Graphs

This section will concentrate on $d = 2$. Section 4.6.1 will demonstrate additional applications of our Theorem 4.1 by revisiting the SSSP and BFS problems and proving Corollaries 4.1 and 4.2. Section 4.6.2 will disprove the "sparsity under edge contraction" belief by establishing Theorem 4.2.

[2] In general, the algorithm of [54] finds the CCs of a graph $G = (V, E)$ in $O(sort(|V|) + sort(|E|) \log\log(|V|B/|E|))$ I/Os.

4.6.1 SSSP and BFS

Consider a grid graph $G = (V, E)$ where each edge in E is associated with a non-negative weight. Given two vertices v_1, v_2, a *path* from v_1 to v_2 is a sequence of edges in E that allows us to walk from v_1 to v_2 without leaving the graph. The *length* of a path is the sum of the weights of all its edges. The *shortest path* from v_1 to v_2 is a path from v_1 to v_2 with the smallest length; the length of the path is the *shortest distance* from v_1 to v_2.

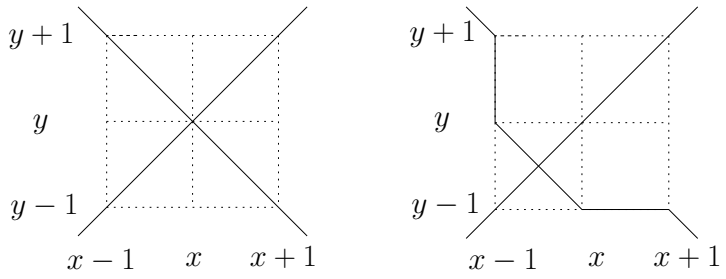
In the SSSP problem, besides G, we are also given a source vertex v_{src}, and need to output the shortest paths and distances from v_{src} to all the other vertices in V. In particular, all the shortest paths must be reported space-economically in a *shortest path tree* where (i) each node corresponds to a distinct vertex in V, (ii) v_{src} is the root, and (iii) the shortest path from v_{src} to any other vertex v in G goes through the same sequence of vertices as in the path from v_{src} to v in the tree. The tree should be stored in the disk using the *child adjacency format* where each node is associated with a list of its children.

Consider an M-separator S of G and its $h = O(|V|/M)$ subgraphs $G_1, ..., G_h$. Given a separator vertex $v \in S$, its *adjacent set* is the set of all G_i ($i \in [1, h]$) such that E has an edge between v and at least one vertex in G_i. Arge et al. [7] proved that the SSSP problem can be solved in $O(|V|/\sqrt{M} + sort(|V|))$ I/Os, as long as S fulfills the following *separator-decomposition* requirement:

> S has been divided into $g = O(|V|/M)$ disjoint subsets $S_1, ..., S_g$ such that the vertices in each S_i ($1 \leq i \leq g$) have the same adjacent set.

Our objective is to strengthen Theorem 4.1 to satisfy the requirement in $O(sort(|V|)$ I/Os.

Let S and $G_1, ..., G_h$ be the separator and subgraphs that Theorem 4.1 returns for G. Recall that our algorithm of Theorem 4.1 recursively performs binary splits, where the cutting planes are just vertical/horizontal lines segments in $\mathbb{N}^2$. It must hold that (i) separator vertices can fall only on these line segments, and (ii) all the vertices of each G_i ($i \leq [1, h]$) are contained in one of the residue rectangles (see definition at the beginning of Section 4.3). This property motivates a simple algorithm for dividing S to satisfy the separator-decomposition requirement. First, label the subgraphs arbitrarily from 1 to h. For each vertex $v \in S$, generate a *label list* that sorts in ascending order the labels of the subgraphs in the adjacent set of v. The list has length $O(1)$. We now partition S into disjoint subsets, where the vertices in each subset have the same label list. The aforementioned property implies that



(a) Before (b) After

FIGURE 4.2: Contortion within a square

there are only $O(|V|/M)$ subsets. The partitioning can be easily done by sorting in $O(sort(|V|))$ I/Os, thus establishing Corollary 4.1.

The BFS problem is, essentially, an instance of SSSP on a grid graph where all edges have the same weight. In particular, the shortest path tree corresponds to the *BFS tree*. Corollary 4.1 immediately implies Corollary 4.2.

4.6.2 Disproving Edge-Contraction Sparsity

This subsection serves as a proof of Theorem 4.2. Recall that a graph $G = (V, E)$ is sparse if $|E| \leq c|V|$, for some constant $c > 0$. Given any integer $m \geq 2$, we will design a grid graph that can be edge-contracted into a clique of m vertices. The clique is not sparse when $m > 2c + 1$. Thus, regardless of the choice of c, there is always a grid graph that is not sparse under edge contraction.

Before proceeding, let us point out a basic geometric fact that will be useful in our design. Let $p_1 = (x_1, y_1)$ and $p_2 = (x_2, y_2)$ be two distinct points in $\mathbb{R}^2$ such that x_1, y_1, x_2, y_2 are all even integers. Let ℓ_1 be the line with slope 1 passing p_1, and ℓ_2 be the line with slope -1 passing p_2. Then, the intersection of ℓ_1 and ℓ_2 must be a point whose coordinates on both dimensions are integers.

Given integers i, j satisfying $i \in [0, m-1]$ and $j \in [0, m-2]$, function $F(i, j)$ returns the point $(1000m \cdot i, 100j)$ in $\mathbb{R}^2$. Call these $m(m-1)$ points *cornerstones*.

For each pair of $i \in [0, m-1]$ and $j \in [0, m-2]$ satisfying $i \leq j$—there are $m(m-1)/2$ such pairs—define a *wedge path* between cornerstones $F(i, j)$ and $F(j+1, i)$ as follows. Shoot a ray with slope 1 emanated from $F(i, j)$, and a ray with slope -1 emanated from $F(j+1, i)$. Let p be the intersection of the two rays; p must have integer coordinates. The wedge path consists of two segments: the first

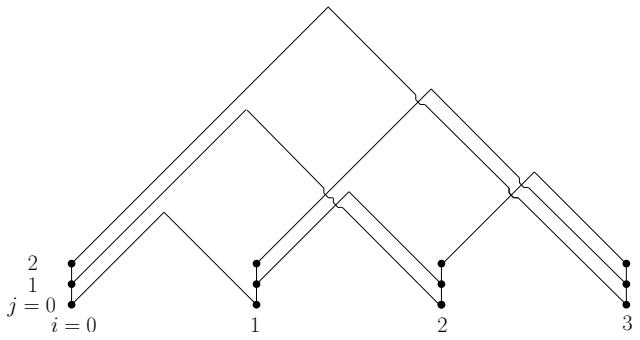


Figure 4.3: The designed grid graph for $m = 4$ (the black points are the cornerstones; the other vertices are dotted along the curves, but are omitted for clarity)

one connects $F(i, j)$ and p, while the second connects p and $F(j + 1, i)$.

The above definition has yielded $m(m - 1)/2$ wedge paths. Two such paths may intersect each other; and the intersection point has integer coordinates—a property that is *not* desired. Next, we will contort some paths a little to ensure the following guarantee: any two resulting paths are either disjoint or intersect only at a point with *fractional* coordinates on both dimensions.

Let P_{intr} be the set of intersection points among the wedge paths. For each point $p = (x, y)$ in P_{intr}, place a square $[x - 1, x + 1] \times [y - 1, y + 1]$ centered at p. The constants 1000 and 100 in the definition of $F(i, j)$ ensure that: (i) the $|P_{intr}|$ squares are disjoint from each other, and (ii) all of them are above the line $y = 100(m - 2)$, i.e., higher than all cornerstones.

Focus now on one such square, as shown on Figure 4.2a, where the two lines illustrate the intersecting wedge paths. We contort one of the two paths as shown in Figure 4.2b, so that the two paths now intersect at the point $(x - 1/2, y - 1/2)$. Apply the same contortion in all squares.

For each $i \in [0, m - 1]$, we add a *vertical path* from cornerstone $F(i, 0)$ through $F(i, m - 2)$. These m paths and the $m(m - 1)/2$ wedge paths (possibly contorted) give rise to the edges in our grid graph G—notice that every path uses only segments each connecting two points whose coordinates are integers differing by at most 1 on each dimension. To complete the graph with vertices, we simply place a vertex at every point p of $\mathbb{R}^2$ such that (i) p has integer coordinates on both dimensions, and (ii) p is on one of those $m + m(m - 1)/2$ paths. See Figure 4.3 for such the final G with $m = 4$.

It remains to explain how to perform edge contractions to convert G into a clique of m vertices. First, contract every vertical path into a "super vertex". Between each pair of super vertices, there remains a sequence of edges corresponding to one unique wedge path. The $m(m-1)/2$ edge sequences

do not share any vertices except, of course, the super vertices. Contracting each wedge path down to the last edge gives the promised clique.

4.7 Summary

This chapter has proved that any d-dimensional grid graph $G = (V, E)$ admits a vertex separator that (i) resembles the well-known multi-way vertex separator of a planar graph, and (ii) can be obtained solely by dividing the space recursively with perpendicular planes, and collecting the vertices on those planes. Furthermore, we have shown that such separators can be computed in $O(sort(|V|))$ I/Os, even if the memory can accommodate only two blocks.

A major application of the above findings is that they lead to an algorithm that performs density-based clustering in d-dimensional space with near-linear I/Os, when the distance metric is L_∞ norm—and hence, also $O(1)$-approximate density-based clustering with the same I/O bound under L_p norm for any $p > 0$. Our techniques also lead to improved results on three fundamental problems: CC, SSSP, and BFS. Specifically, the CC problem has been settled in $O(sort(|V|))$ I/Os for any d-dimensional grid graph $G = (V, E)$. Our improvement on SSSP and BFS, however, is less significant, and concerns only small values of M.

We close the chapter with two open questions. First, is it possible utilize our separator-computation algorithm to improve the I/O complexity of the DFS algorithm in [37]? Second, does BFS on a 2D grid graph require $\Omega(|V|/\sqrt{M})$ I/Os in the worst case, thus making the result of Corollary 4.2 optimal?

Chapter 5

Conclusions

DBSCAN is an effective technique for density-based clustering, which is very extensively applied in data mining, machine learning, and databases. However, currently there has not been clear understanding on its theoretical computational hardness. All the existing algorithms suffer from a time complexity that is quadratic to the dataset size n when the dimensionality d is at least 3.

In this thesis, we have shown that, unless very significant breakthroughs (ones widely believed to be impossible) can be made in theoretical computer science, the DBSCAN problem requires $\Omega(n^{4/3})$ time to solve for $d \geq 3$ under the Euclidean distance. This excludes the possibility of finding an algorithm of near-linear running time, thus motivating the idea of computing approximate clusters. Towards that direction, we proposed ρ-approximate DBSCAN, and proved both theoretical and experimentally that the new method has excellent guarantees both in the quality of cluster approximation and computational efficiency.

The exact DBSCAN problem in dimensionality $d = 2$ is known to be solvable in $O(n \log n)$ time. We have further enhanced that understanding by showing how to settle the problem in $O(n)$ time, provided that the data points have already been pre-sorted on each dimension. In other words, coordinating sorting is in fact the hardest component of the 2D DBSCAN problem. The result immediately implies that, when all the coordinates are integers, the problem can be solved in $O(n \log \log n)$ time deterministically, or $O(n \sqrt{\log \log n})$ expected time randomly.

Moreover, we have presented a systematic study on dynamic density based clustering under the theme of DBSCAN. Our findings reveal considerable new insight into the characteristics of the topic,

by providing a complete picture of the computational hardness in various update schemes. Perhaps the most surprising result is that ρ-approximate DBSCAN, which was proposed to address the worst-case computational intractability of exact DBSCAN, suffers from the same hardness when both insertions and deletions are allowed. We have also shown how to eliminate the issue elegantly with a tiny relaxation, which has led to the development of ρ-double-approximate DBSCAN, and designed a suite of new algorithms that achieve near-constant update time in cluster maintenance essentially in all the update schemes where this is possible.

To support datasets that can not fit in main memory, we studied the density-based clustering and its related graph problems in external memory model. We proved that any d-dimensional grid graph $G = (V, E)$ admits a vertex separator that (i) resembles the well-known multi-way vertex separator of a planar graph, and (ii) can be obtained solely by dividing the space recursively with perpendicular planes, and collecting the vertices on those planes. Then, we showed that such separators can be computed in $O(sort(|V|))$ I/Os, even if the memory can accommodate only two blocks.

A major application of the above findings is that they lead to an algorithm that performs density-based clustering in d-dimensional space with near-linear I/Os, when the distance metric is L_∞ norm—and hence, also $O(1)$-approximate density-based clustering with the same I/O bound under L_p norm for any $p > 0$. Our techniques also lead to improved results on three fundamental problems: CC, SSSP, and BFS. Specifically, the CC problem has been settled in $O(sort(|V|))$ I/Os for any d-dimensional grid graph $G = (V, E)$. Our improvement on SSSP and BFS, however, is less significant, and concerns only small values of M.

References

[1] P. K. Agarwal, L. Arge, O. Procopiuc, and J. S. Vitter. A framework for index bulk loading and dynamization. In *Proceedings of International Colloquium on Automata, Languages and Programming (ICALP)*, pages 115–127, 2001.

[2] P. K. Agarwal, L. Arge, and K. Yi. I/O-efficient batched union-find and its applications to terrain analysis. 7(1):11, 2010.

[3] P. K. Agarwal, H. Edelsbrunner, and O. Schwarzkopf. Euclidean minimum spanning trees and bichromatic closest pairs. *Discrete & Computational Geometry*, 6:407–422, 1991.

[4] A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM (CACM)*, 31(9):1116–1127, 1988.

[5] A. Andersson, T. Hagerup, S. Nilsson, and R. Raman. Sorting in linear time? *Journal of Computer and System Sciences (JCSS)*, 57(1):74–93, 1998.

[6] M. Ankerst, M. M. Breunig, H.-P. Kriegel, and J. Sander. OPTICS: Ordering points to identify the clustering structure. In *Proceedings of ACM Management of Data (SIGMOD)*, pages 49–60, 1999.

[7] L. Arge, G. S. Brodal, and L. Toma. On external-memory mst, SSSP and multi-way planar graph separation. *J. Algorithms*, 53(2):186–206, 2004.

[8] L. Arge, J. S. Chase, P. N. Halpin, L. Toma, J. S. Vitter, D. Urban, and R. Wickremesinghe. Efficient flow computation on massive grid terrain datasets. *GeoInformatica*, 7(4):283–313, 2003.

[9] L. Arge, L. Toma, and J. S. Vitter. I/O-efficient algorithms for problems on grid-based terrains. *ACM Journal of Experimental Algorithmics*, 6:1, 2001.

[10] S. Arya and D. M. Mount. Approximate range searching. *Computational Geometry*, 17(3-4):135–152, 2000.

[11] S. Arya and D. M. Mount. A fast and simple algorithm for computing approximate euclidean minimum spanning trees. In *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1220–1233, 2016.

[12] S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, and A. Y. Wu. An optimal algorithm for approximate nearest neighbor searching fixed dimensions. *Journal of the ACM (JACM)*, 45(6):891–923, 1998.

[13] K. Bache and M. Lichman. UCI machine learning repository, 2013.

[14] N. Beckmann, H. Kriegel, R. Schneider, and B. Seeger. The R*-tree: An efficient and robust access method for points and rectangles. In *Proceedings of ACM Management of Data (SIGMOD)*, pages 322–331, 1990.

[15] C. Böhm, K. Kailing, P. Kröger, and A. Zimek. Computing clusters of correlation connected objects. In *Proceedings of ACM Management of Data (SIGMOD)*, pages 455–466, 2004.

[16] B. Borah and D. K. Bhattacharyya. An improved sampling-based DBSCAN for large spatial databases. In *Proceedings of Intelligent Sensing and Information Processing*, pages 92–96, 2004.

[17] P. Bose, A. Maheshwari, P. Morin, J. Morrison, M. H. M. Smid, and J. Vahrenhold. Space-efficient geometric divide-and-conquer algorithms. *Computational Geometry*, 37(3):209–227, 2007.

[18] T. M. Chan. A dynamic data structure for 3-d convex hulls and 2-d nearest neighbor queries. *Journal of the ACM (JACM)*, 57(3), 2010.

[19] V. Chaoji, M. A. Hasan, S. Salem, and M. J. Zaki. Sparcl: Efficient and effective shape-based clustering. In *Proceedings of International Conference on Management of Data (ICDM)*, pages 93–102, 2008.

[20] M. de Berg, O. Cheong, M. van Kreveld, and M. Overmars. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, 3 edition, 2008.

[21] M. de Berg, C. Tsirogiannis, and B. T. Wilkinson. Fast computation of categorical richness on raster data sets and related problems. pages 18:1–18:10, 2015.

[22] K. Deng, X. Zhou, H. T. Shen, Q. Liu, K. Xu, and X. Lin. A multi-resolution surface distance model for *k*-nn query processing. *The VLDB Journal*, 17(5):1101–1119, 2008.

[23] J. Erickson. On the relative complexities of some geometric problems. In *Proceedings of the Canadian Conference on Computational Geometry (CCCG)*, pages 85–90, 1995.

[24] J. Erickson. New lower bounds for Hopcroft's problem. *Discrete & Computational Geometry*, 16(4):389–418, 1996.

[25] M. Ester. Density-based clustering. In *Data Clustering: Algorithms and Applications*, pages 111–126. 2013.

[26] M. Ester, H. Kriegel, J. Sander, M. Wimmer, and X. Xu. Incremental clustering for mining in a data warehousing environment. In *Proceedings of Very Large Data Bases (VLDB)*, pages 323–333, 1998.

[27] M. Ester, H. Kriegel, J. Sander, and X. Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Proceedings of ACM Knowledge Discovery and Data Mining (SIGKDD)*, pages 226–231, 1996.

[28] G. N. Frederickson. Fast algorithms for shortest paths in planar graphs, with applications. *SIAM Journal of Computing*, 16(6):1004–1022, 1987.

[29] J. Gan and Y. Tao. DBSCAN revisited: Mis-claim, un-fixability, and approximation. In *Proceedings of ACM Management of Data (SIGMOD)*, pages 519–530, 2015.

[30] S. Ghemawat, H. Gobioff, and S. Leung. The google file system. pages 29–43, 2003.

[31] A. Gunawan. A faster algorithm for DBSCAN. Master's thesis, Technische University Eindhoven, March 2013.

[32] A. Guttman. R-trees: a dynamic index structure for spatial searching. In *Proceedings of ACM Management of Data (SIGMOD)*, pages 47–57, 1984.

[33] J. Han, M. Kamber, and J. Pei. *Data Mining: Concepts and Techniques*. Morgan Kaufmann, 2012.

[34] Y. Han and M. Thorup. Integer sorting in 0(n sqrt (log log n)) expected time and linear space. In *Proceedings of Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 135–144, 2002.

[35] H. J. Haverkort. I/O-optimal algorithms on grid graphs. *CoRR*, abs/1211.2066, 2012.

[36] H. J. Haverkort and L. Toma. I/O-efficient algorithms on near-planar graphs. *J. Graph Algorithms Appl.*, 15(4):503–532, 2011.

[37] J. Her and R. S. Ramakrishna. An external-memory depth-first search algorithm for general grid graphs. *Theoretical Computer Science*, 374(1-3):170–180, 2007.

[38] G. R. Hjaltason and H. Samet. Distance browsing in spatial databases. *ACM Transactions on Database Systems (TODS)*, 24(2):265–318, 1999.

[39] J. Holm, K. de Lichtenberg, and M. Thorup. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. *Journal of the ACM (JACM)*, 48(4):723–760, 2001.

[40] D. G. Kirkpatrick and S. Reisch. Upper bounds for sorting integers on random access machines. *Theoretical Computer Science*, 28:263–276, 1984.

[41] M. Klusch, S. Lodi, and G. Moro. Distributed clustering based on sampling local density estimates. In *Proceedings of the International Joint Conference of Artificial Intelligence (IJCAI)*, pages 485–490, 2003.

[42] Z. Li, B. Ding, J. Han, and R. Kays. Swarm: Mining relaxed temporal moving object clusters. *Proceedings of the VLDB Endowment (PVLDB)*, 3(1):723–734, 2010.

[43] B. Liu. A fast density-based clustering algorithm for large databases. In *Proceedings of International Conference on Machine Learning and Cybernetics*, pages 996–1000, 2006.

[44] L. Liu and R. C. Wong. Finding shortest path on land surface. In *Proceedings of ACM Management of Data (SIGMOD)*, pages 433–444, 2011.

[45] E. H.-C. Lu, V. S. Tseng, and P. S. Yu. Mining cluster-based temporal mobile sequential patterns in location-based service environments. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 23(6):914–927, 2011.

[46] A. Maheshwari and N. Zeh. I/O-efficient planar separators. *SIAM Journal of Computing*, 38(3):767–801, 2008.

[47] S. Mahran and K. Mahar. Using grid for accelerating density-based clustering. In *Proceedings of IEEE International Conference on Computer and Information Technology (CIT)*, pages 35–40, 2008.

[48] J. Matousek. Range searching with efficient hiearchical cutting. *Discrete & Computational Geometry*, 10:157–182, 1993.

[49] K. Mehlhorn and U. Meyer. External-memory breadth-first search with sublinear I/O. In *Proceedings of European Symposium on Algorithms (ESA)*, pages 723–735, 2002.

[50] B. L. Milenova and M. M. Campos. O-Cluster: Scalable clustering of large high dimensional data sets. In *Proceedings of International Conference on Management of Data (ICDM)*, pages 290–297, 2002.

[51] G. L. Miller, S. Teng, and S. A. Vavasis. A unified geometric approach to graph separators. In *Proceedings of Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 538–547, 1991.

[52] D. Moulavi, P. A. Jaskowiak, R. J. G. B. Campello, A. Zimek, and J. Sander. Density-based clustering validation. In *International Conference on Data Mining*, pages 839–847, 2014.

[53] D. M. Mount and E. Park. A dynamic data structure for approximate range searching. In *Proceedings of Symposium on Computational Geometry (SoCG)*, pages 247–256, 2010.

[54] K. Munagala and A. G. Ranade. I/O-complexity of graph algorithms. In *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 687–694, 1999.

[55] M. H. Nodine, M. T. Goodrich, and J. S. Vitter. Blocking for external graph searching. In *Proceedings of ACM Symposium on Principles of Database Systems (PODS)*, pages 222–232, 1993.

[56] M. M. A. Patwary, D. Palsetia, A. Agrawal, W. Liao, F. Manne, and A. N. Choudhary. A new scalable parallel DBSCAN algorithm using the disjoint-set data structure. In *Conference on High Performance Computing Networking, Storage and Analysis*, page 62, 2012.

[57] T. Pei, A.-X. Zhu, C. Zhou, B. Li, and C. Qin. A new approach to the nearest-neighbour method to discover cluster features in overlaid spatial point processes. *International Journal of Geographical Information Science*, 20(2):153–168, 2006.

[58] A. Reiss and D. Stricker. Introducing a new benchmarked dataset for activity monitoring. In *International Symposium on Wearable Computers*, pages 108–109, 2012.

[59] S. Roy and D. K. Bhattacharyya. An approach to find embedded clusters using density based techniques. In *Proceedings of Distributed Computing and Internet Technology*, pages 523–535, 2005.

[60] C. Shahabi, L. A. Tang, and S. Xing. Indexing land surface for efficient knn query. *PVLDB*, 1(1):1020–1031, 2008.

[61] G. Sheikholeslami, S. Chatterjee, and A. Zhang. Wavecluster: A wavelet based clustering approach for spatial data in very large databases. *The VLDB Journal*, 8(3-4):289–304, 2000.

[62] C. Sheng and Y. Tao. Finding skylines in external memory. In *Proceedings of ACM Symposium on Principles of Database Systems (PODS)*, pages 107–116, 2011.

[63] W. D. Smith and N. C. Wormald. Geometric separator theorems & applications. In *Proceedings of Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 232–243, 1998.

[64] P.-N. Tan, M. Steinbach, and V. Kumar. *Introduction to Data Mining*. Pearson, 2006.

[65] R. E. Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM (JACM)*, 22(2):215–225, 1975.

[66] R. E. Tarjan. A class of algorithms which require nonlinear time to maintain disjoint sets. *Journal of Computer and System Sciences (JCSS)*, 18(2):110–127, 1979.

[67] L. Toma and N. Zeh. I/O-efficient algorithms for sparse graphs. In *Algorithms for Memory Hierarchies, Advanced Lectures*, pages 85–109, 2002.

[68] C.-F. Tsai and C.-T. Wu. GF-DBSCAN: A new efficient and effective data clustering technique for large databases. In *Proceedings of International Conference on Multimedia Systems and Signal Processing*, pages 231–236, 2009.

[69] M. Varma and A. Zisserman. Texture classification: Are filter banks necessary? In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 691–698, 2003.

[70] W. Wang, J. Yang, and R. R. Muntz. STING: A statistical information grid approach to spatial data mining. In *Proceedings of Very Large Data Bases (VLDB)*, pages 186–195, 1997.

[71] J.-R. Wen, J.-Y. Nie, and H. Zhang. Query clustering using user logs. *ACM Transactions on Information Systems (TOIS)*, 20(1):59–81, 2002.

[72] S. Xing, C. Shahabi, and B. Pan. Continuous monitoring of nearest neighbors on land surface. *Proceedings of the VLDB Endowment (PVLDB)*, 2(1):1114–1125, 2009.

[73] N. Zeh. I/O-efficient graph algorithms. Technical report, Dalhousie University, 2002.