MASTER'S THESIS 2023

Investigating Optimization Techniques for Algorithms in Radars

Joseph Atalla, Joakim Mörling

ISSN 1650-2884
LU-CS-EX: 2023-43

DEPARTMENT OF COMPUTER SCIENCE
LTH | LUND UNIVERSITY

EXAMENSARBETE
Datavetenskap

LU-CS-EX: 2023-43

Investigating Optimization Techniques for Algorithms in Radars

Optimering av algoritmer för Radar

Joseph Atalla, Joakim Mörling

Investigating Optimization Techniques for Algorithms in Radars

(An effort to improve the execution time of the signal processing chain)

Joseph Atalla
jo2605at-s@student.lu.se

Joakim Mörling
jo1732mo-s@student.lu.se

August 28, 2023

Master's thesis work carried out at Axis Communications AB.

Supervisors:
Henrik Lehtonen
Santhosh Nadig
Aras Papadelis
Jonas Skeppstedt, jonas.skeppstedt@cs.lth.se
Examiner:

Per Andersson, per.andersson@cs.lth.se

Abstract

This thesis analyzes bottlenecks in a proposed radar processing chain and attempts to investigate various optimization techniques for such bottlenecks. One bottleneck, in particular, was deemed bigger than the others and is the focus of this master thesis. The bottleneck is a clustering algorithm called DBSCAN. When more input is introduced, the algorithm scales non-linearly, reducing the time available for other processes.

A known mitigation strategy for this problem is to down-sample the data, guaranteeing consistent performance. However, this approach has the undesirable side effect of reducing the radar's clustering accuracy. This thesis investigates alternative solutions that directly address the core problem, aiming for higher performance. Some different algorithms were tested, and a new algorithm was found to increase the throughput by over 30 times. Attributes of different clustering algorithms have also been mapped.

Keywords: Radars, DBSCAN, Clustering, Optimization, Embedded systems, ρ-approximate DBSCAN

Acknowledgements

We want to express our gratitude to all those who have contributed developing of this master thesis. First and foremost, we thank our supervisors, who provided invaluable guidance, feedback, and support throughout the research process. Without the experienced advice from Aras Papadelis we would have lost track of what is essential. These findings were possible with the immense help from Henrik Lehtonen with cross-compilation, all discussions, helpful suggestions with report writing, and the attention to detail in the math and results. We want to thank Santhosh Nadig for his insightful advice.

A special thanks to our colleague Sebastian Fors, for helping us find the real-world datasets in our time of need.

We want to thank our supervisor from LTH, Jonas Skeppstedt, who fully believed in us throughout the process.

We would want to thank the authors Gan and Tao [18] for letting us use their binary code; without it, our results would not have been as far-reaching, and the scope of this thesis would have been smaller.

Finally, we would like to thank our families for their love, encouragement, and patience during the long hours of research and writing. Without their support, this project would not have been possible.

Thank you all for your contributions and support.

Contents

Chapter 1
Introduction

1.1 Introduction

Historical catastrophes are undeniably tragic due to the loss of life they entail, but they have occasionally led to advancements in various scientific fields. One field that witnessed significant progress due to World War II is technology, particularly in developing computer science and radar systems. Today, radars serve many purposes, with some of their applications being object detection in vehicles and security cameras.

This paper delves into the use of radars in security camera systems. It explores various ways to optimize their performance by potentially improving the time or memory complexity of one of their most resource-intensive algorithms. The radar under investigation is an FMCW radar supplied to us by the hosting company. In most situations, radar units perform flawlessly in object detection and data handling. However, the radar units' mode of operation relies on detecting moving objects. In real-world scenarios, numerous moving objects, such as swaying tree leaves or severe weather, can place a considerable strain on the object detection functionality of the units. This stress can result in reduced performance.

1.2 Background

1.2.1 A brief history of radars

The word radar stands for *Radio Detection and Ranging*. While the invention of radars was credited to Christian Hülsmeyer in the early 1900s, his invention did not include a system to compute distances [4]. The computation of distances in radar was later introduced in World War II [9]. Since then, radars have continuously been developed and improved in hardware and software aspects.

1.2.2 FMCW radar

This thesis's type of radar in deployment is Frequency Modulated Continuous Wave (FMCW). As the name suggests, the transmitted signal changes frequency during its duration, beginning at frequency f_0 and increasing linearly to f_1. Such a signal is called a chirp. Nearby objects reflect the transmitted signal and a receiver is used to detect the reflection. A single chirp can be used to measure the distance to the target, and by transmitting a series of chirps and measuring the phase difference between the received signals, the target's radial velocity can be calculated. Multiple receiver antennas can be used to determine the polar angle of the target with respect to the radar. The polar angles of the target can be deduced by measuring the phase difference between the antennas for the signal reflected by a target. More details on signal processing come in the following sections.

After receiving the signal, the radar uses the transmitted and the received chirps' frequencies and phase shifts to extract information about the targets, such as distance, velocity, and position. The following presents further processing done by the radar after the signal is received.

1.2.3 IF signaling

The chirp signals can be challenging to process directly due to their high frequencies, as it is difficult and expensive to implement suitable electronic components [40]. The radar system introduces a frequency mixer to overcome these challenges. The purpose of the frequency mixer is to use the received chirps and a generated signal with a frequency that is slightly different and mix them. As a result, two new signals are generated. The first signal is the sum of the two frequencies and is filtered out, while the other contains the difference of the frequencies [36]. The latter is called the Intermediate Frequency signal (IF signal). This process can be seen in Figure 1.1 [46] in which $\boldsymbol{Tx}$ Antenna transmits the signal and $\boldsymbol{Rx}$ Antenna receives the signal. Since the radar is modulated, the transmitted signal changes its frequency over time, which is why the same sinusoidal signal presented in the diagram would provide a slightly different frequency signal from the one passed by the $\boldsymbol{Rx}$ Antenna. The IF signal contains vital information for the object the signal was reflected by. This makes it worthwhile as it is at a lower frequency and is applicable for further processing in the radar unit.

1.2.4 Fast Fourier Transform

The "chirp" signal, when reflected by multiple targets, poses a challenge in isolating the Intermediate Frequency (IF) signals in the time domain. The IF signal becomes an amalgamation of these simultaneously received signals. The signals can be represented in the frequency domain using the Fourier Transform (FT) to address this issue [40].

The FT is an essential tool in radar signal processing, especially in analyzing and interpreting IF signals. This mathematical technique enables the conversion of a time-domain signal, which depicts how a signal evolves, into a frequency-domain representation [36]. This new representation highlights the distinct frequency components constituting the original signal, allowing for easier isolation and analysis.

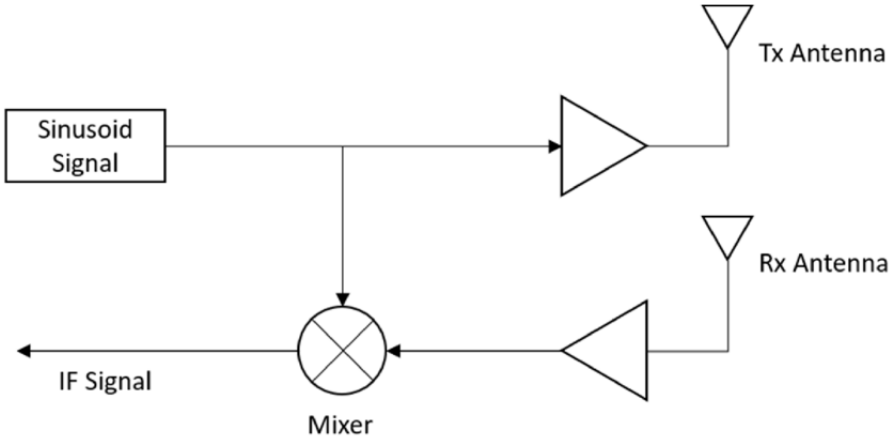


Figure 1.1: Frequency mixer in radar units.

The fastest algorithm to compute a Fourier transformation as of today is the Fast-Fourier Transform (FFT) [15], first published in 1965 [14]. The Fast Fourier Transform (FFT) algorithm efficiently computes the discrete Fourier Transform. The FFT has become an indispensable tool for working with discrete signals in digital signal processing. It uses the inherent symmetries and redundancies in the Fourier Transform computation to significantly reduce the required arithmetic operations. As a result, the FFT algorithm exhibits a computational complexity of $O(n \log n)$, where n is the number of data points in the signal. This efficiency makes the FFT particularly well-suited for processing large datasets and real-time applications [8].

1.2.5 Radar Cross-Section

In practice, the received signals are weaker than the transmitted ones. Many factors cause this, such as signal power scaling ($\frac{1}{r^4}$ scale) but also the Radar Cross-Section (RCS). RCS measures an object's ability to reflect radar signals in the direction of the radar receiver. It is an important parameter that characterizes the detectability of an object by radar systems. The radar range equation can calculate the power of the received signal (P_r), as presented below.

$$P_r = \frac{P_t G_t \sigma A_e}{(4\pi)^2 r^4}$$

Where σ is RCS, r is the distance to the target, P_t is the power of the transmitted signal, G_t is the gain of the transmitted signal, and A_e is the receiving aperture which is related to the gain of the receiving antenna and the signal wavelength. Using this equation, we can deduce the RCS of a target:

$$P_r = \frac{P_t G_t \sigma A_e}{(4\pi)^2 r^4} \implies \sigma = \frac{(4\pi)^2 r^4 P_r}{P_t G_t A_e}$$

[37]

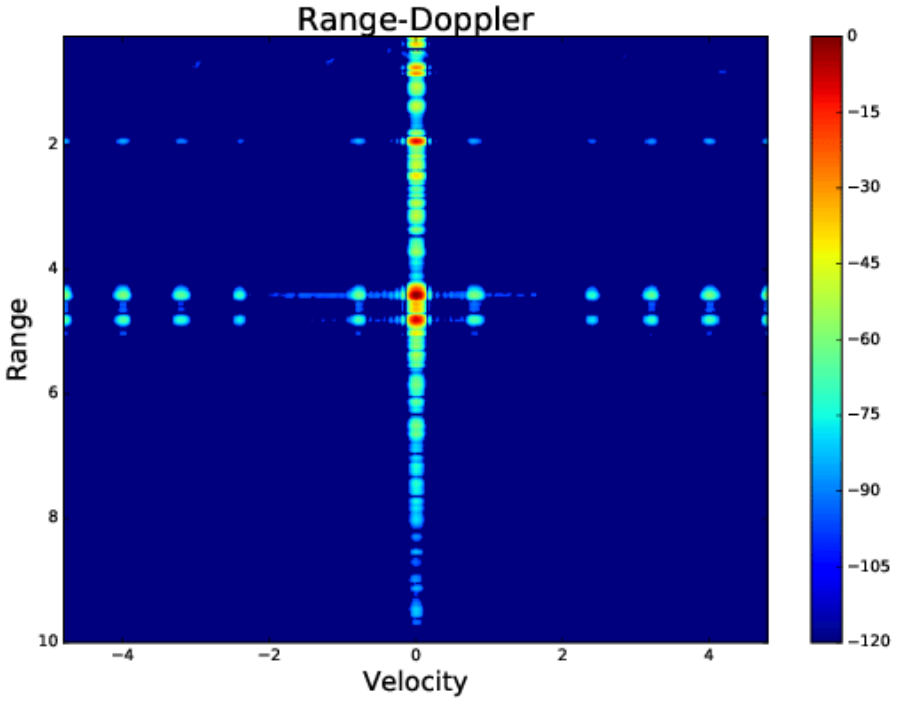


Figure 1.2: Example of a Range-Doppler plot.

1.2.6 Range-Doppler plot

A so-called Range-Doppler plot can be produced using the IF signals processed with FFT. A Range-Doppler plot represents the relative velocity (to the radar) on the x-axis and the range (object distance) on the y-axis. An example of a Range-Doppler plot can be seen in Figure 1.2 [41]. Each point in the Range-Doppler plot represents the received power for an object with a given distance and relative velocity. The point represents random noise if no object exists with that combination of range and velocity.

1.2.7 Point clouds

The radar implements a detector to filter out noisy or weak points. The rest are considered detections that supply the radar with points on the Range-Doppler plot. A point cloud is a set of points on the Range-Doppler graph containing coordinates for the points and other radar information, including the RCS and azimuth (angle).

1.2.8 Radar processing chain

Data processing in the radar device is handled in a chain where processing steps run sequentially. This processing chain runs once every frame in the radar unit, so, for instance, if the processing frequency is 10 Hz, the frame deadline would be 100 ms. Figure 1.3 shows an example of an overview of the processing chain that can be used on the radar device used for this thesis. As can be seen, the raw data is first sent as input to FFT in the hardware, which is then

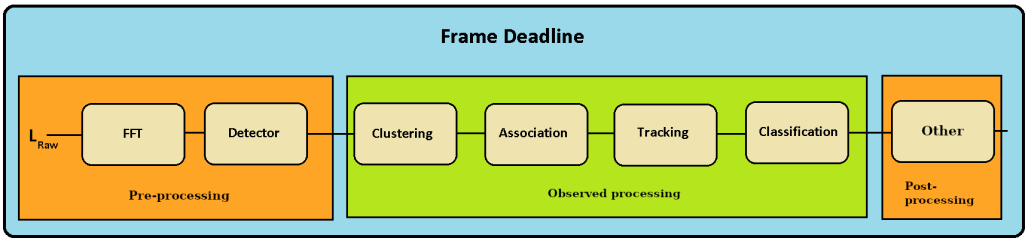


Figure 1.3: Radar processing chain.

processed by the detector in one software process. Then steps such as clustering, association, tracking, and classification are run sequentially in another software process.

1.3 Problem formulation

1.3.1 Frame deadline problem

Given the example in Section 1.2.8, there is a deadline for processing each frame within 100 ms. The processing deadline is one of the main topics that must be handled when implementing radar data processing, as it enforces a tight constraint on the embedded system. If an algorithm or a processing step takes progressively longer than others, it reduces the time allotted for other steps to finish. Having time taken from other processes is construed as a bottleneck. Section 2.1 covers the concept of bottlenecks. This problem can partially be solved by down-sampling the data.

1.3.2 Down-sampling problem

When the radar detects many moving objects, the various processing steps start to take longer and may exceed the 100 ms frame deadline. A solution that the hosting company actively investigates is to down-sample the dataset. In this case, down-sampling becomes a shallow solution because it results in worse clustering precision. Optimizing the algorithm is a better solution to remove the need to down-sampling the input data potentially. However, even if the down-sampling factor can be reduced but not entirely removed due to optimization, it will be an overall software improvement.

1.3.3 Goal

This thesis aims to evaluate the existing algorithms by identifying which ones are a bottleneck and looking at potential enhancements.

1.3.4 Research questions

The following presents the research questions (RQ) we formulated to achieve the goals of this thesis.

RQ1. Are there any possibilities to improve the time complexity of the current algorithms?

RQ2. What are the advantages and disadvantages of the various implementations of the algorithms?

1.3.5 Limitations

A vast number of candidate algorithms could be used in the signal processing for the radars of the type used in this thesis, where going through each one of the algorithms would not be possible. Therefore, we set some limitations before we searched algorithms. In addition, we excluded mathematical algorithms and advanced machine-learning techniques as out of the scope of this thesis.

Difference in hardware

In the embedded system, the radar shares processing power with other software components on the unit. The system also uses a different CPU architecture compared to a standard computer. These may provide different results compared to a desktop CPU (x86-64).

1.3.6 Iterative improvement process

This section presents a generalized representation of the improvement process for the algorithm optimizations (not only for determining bottlenecks).

1. Identify holdup — localize the processing steps that burden the chain during heavy loads to avoid exceeding the frame deadline.
2. Profile current implementation — Measure time spent and reason about time complexity.
3. Search publications — If an algorithm has no room for improvement, then publications with other algorithms will be searched for.
4. Implement an improved algorithm if there is one. Otherwise, make small incremental improvements to the current algorithm.
5. Measure the new algorithm with the previous one as the base case. Return to the previous step until achieving satisfactory performance.

Chapter 2
Bottlenecks and optimization

2.1 Bottlenecks

The radar unit processing chain contains many steps, such as multiple FFT executions, clustering, and more. As the radar unit runs the processing chain at ten fps, each execution has a deadline of 100 ms to finish. When the algorithm does not meet the deadline, the burden falls on some other algorithms, delaying or skipping the consecutive frames. We consider these algorithms bottlenecks, as they are the ones that take the most time in the radar processing chain, and the following steps can only start once they have finished.

2.2 Background on optimization

There are many aspects to look at when it comes to optimization. The following points illustrate aspects of how software can be optimized.

2.2.1 Memory complexity

One of the common ways to optimize algorithms is by reduction of memory usage. One can do this by reducing the amount or size of stored data. Another possible memory optimization is by reducing the number of heap allocations.

In this context, memory optimizations may not be the way to go. The radar units contain 500-1000 MB of RAM, with most of that memory not used while running the unit, meaning that the memory optimizations by tightly packing data would barely make a difference. Nonetheless, it would be possible to alter the code in a way that may reduce cache misses by taking advantage of spatial or temporal locality. *Temporal locality* is a memory property in which the same memory address will most likely be reused. *Spatial locality* is a memory property that proposes that the following memory address will be accessed next. This may

be done by reusing the same data structure for all frames (not creating a new one every frame) or packing all possible variables together in memory. Packing memory reduces the memory access time as cache memory is used instead of RAM, which improves the throughput of the software.

2.2.2 Time complexity

Optimizing time complexity will be the focus of this paper. Time complexity is one of the most generalized representations of algorithm speed, making it independent of hardware specifications. Improving time complexity can be done in multiple ways. It can be as simple as using a data structure with improved search time complexity, but it can also be using an improved variation of the algorithm. Of course, in many cases, using a data structure goes hand-in-hand with improving the algorithm.

Another aspect to discuss when it comes to improving time complexity may lie in a numerical analysis perspective. This is because one common way to optimize algorithms today is by estimating the results instead of computing them precisely, and this is considered one drawback of using such optimizations. In many applications, estimated results are acceptable. On the other hand, not all optimizations necessarily affect the results. Thus, it is crucial to monitor accuracy when making algorithm changes.

2.2.3 Other optimizations

When it comes to optimization, it is sometimes the case that a program can be improved in other ways than just improving the algorithm's time complexity or memory usage. The compiler can optimize some programs by adding the "-O3" flag when compiling with the GNU compiler. Other programs can be optimized by implementing a multi-core approach. Also, there may be more mathematical changes that we can do. For instance, the number of instructions can be optimized by reducing the need for division and square root instructions, as they use more clock cycles than other simpler arithmetic operations such as addition and multiplication. If possible, we may use a squared quantity $x^2 = y$ in an algorithm to avoid computing the square root $x = \sqrt{y}$.

Data layouts

There are generally two ways to store data sequentially when using data structures. AoS stands for Array of Structures, and as the name suggests, the data is stored by storing the structures that contain their attributes in an array. The other method, SoA, stands for Structure of Arrays, which means that one structure contains multiple arrays representing each attribute. Neither layout can be considered generally more efficient than the other when it comes to optimization, but depending on how the data is used, the correct data layout would help reduce cache misses, which will reduce execution time [45]; sometimes, even a combination of the two may be the best.

2.3 Determining bottlenecks

Now, the question lies in what algorithms are mostly suited for optimization. We can determine this by observing the software processing chain monitoring and comparing the constituent algorithms' respective executions and call stacks. We can do this introspection for the algorithms already implemented on the radar system since it is supported by one of the materials mentioned below. In addition, it is possible to measure different parts of the processing chain on a dataset and retrieve statistics for its respective steps, which we then use for determining the bottlenecks in the radar system.

2.3.1 Material

Perf

Perf is a performance profiler for Linux. It supports hardware counters, tracepoints, and much more. It is one of the most common profiling tools [44].

Hotspot

A graphical user interface for visualizing *perf* output. Among the features are flame graphs that make it easy to see the relative time spent in each level of the call stack and who the callers and callees for any given function are. Hotspot is an open-source project available on GitHub [26].

Probe

An internal tool supplied to us by the hosting company makes it possible to measure the execution time of different parts of the processing chain. In Figure 2.1, we can see the maximum and average time spent in each part of the code and the amount of times run (triggers).

2.4 Hotspot and probe statistics

Figure 2.1 shows the flame graph for the processing chain of the radar unit. Detailed modules we blurred for confidentiality reasons. The data in Table 2.1 and the flame graph we retrieved from the same recording. The recording is called *F3* and is the most computationally heavy dataset we know of. The dataset, along with other ones, will be discussed further on.

2.5 Bottleneck analysis

The flame graph in Figure 2.1 shows that the processing steps that take the most time are classification and clustering. One can also observe that classification takes longer than clustering. A more detailed overview of this one can view in Table 2.1. When the processing chain is pushed to its limits, it shows that the significant bottleneck is clustering, with 31% of the time in the chain and a maximum time of 82.35 ms and an average time of 46.1 ms.

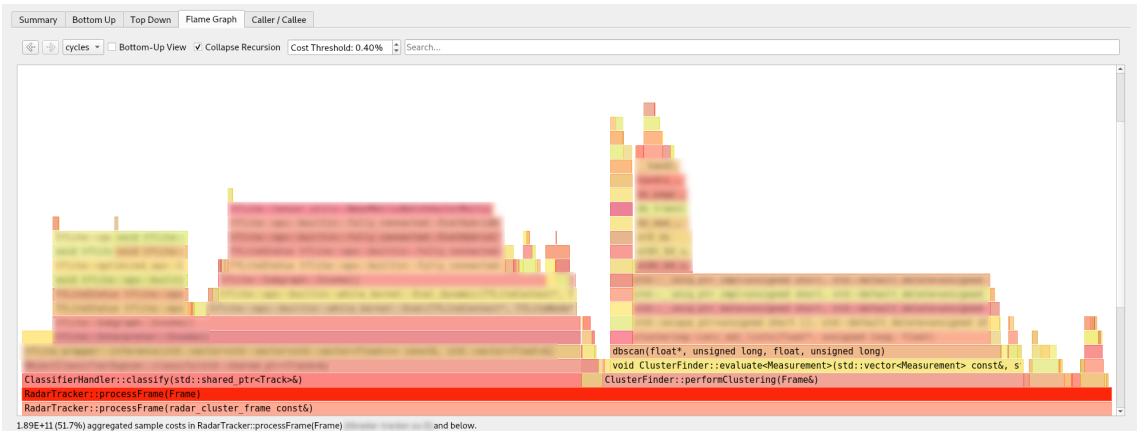


Figure 2.1: Flame graph when running the *F3* dataset on the radar. DBSCAN and classification take the most amount of time.

(Milliseconds)	Total time	Max	Avg time	Triggers
Clustering	106823	82.35	46.1	2318
Association	2558.1	3.3	1.1	2317
Tracking Prediction	1069.1	1.1	0	147167
Track Update	7314.5	6	0.05	102792
Classification	223184	32.7	5.9	37841

Table 2.1: Example of probe statistics taken a run of the *F3* dataset on the radar.

The other measurements have a lower average and maximum time while having more triggers, meaning they take more time because they run more often. That is why total time is not a complete measurement, as we would also have to compare the number of triggers for the result to show an even time distribution. On the other hand, the reality is that the algorithms run different amounts of time, and the chain as a whole needs to be optimized. Since the decision is between classification and clustering, and classification is done using machine learning, we decided to optimize clustering.

2.6 Determined bottleneck

As a bottleneck, we deduced that clustering possesses the most considerable potential among the discussed procedures in the radar system. According to internal documents about the clustering, it is viewed as a bottleneck, and a downsampling method was introduced that would remove input points if the total number of points in a frame was more than four hundred. Additionally, a time target for clustering was set to $50\,\mathrm{ms}$, with some measurements after downsampling reaching over $60\,\mathrm{ms}$. In other words, the frames with the most points in the recording we probed on would not be run since they would take too long to process.

Chapter 3
Clustering algorithms

This chapter covers, in detail, how the clustering algorithm is currently optimized and the prospective improvements.

3.1 Clustering

One of the key differences between humans and computers is that humans can quickly identify density-based clusters. When using signal-processing units, the points created are discrete, which will create multiple points from the same object with no connection to each other. For this reason, point clustering is introduced to establish connections among such points. A point cluster is a group of points in a graph that can be identified as one object, as seen in Figure 3.1. One can use multiple algorithms to solve the clustering problem, some of which include k-means and *DBSCAN*. Such algorithms are selected for usage depending on application, efficiency, and limitations.

3.2 Original DBSCAN

This section will cover the clustering algorithm called **DBSCAN**. DBSCAN is widely used for applications that require no previous knowledge of the number of clusters.

3.2.1 Definition

DBSCAN stands for Density-based spatial clustering of applications with noise. The DBSCAN algorithm is a clustering algorithm that finds the nearest neighbors within an ϵ distance of each other and then identifies clusters among them containing at least $minPts$ neighboring points. This process, one can see in Figure 3.2.

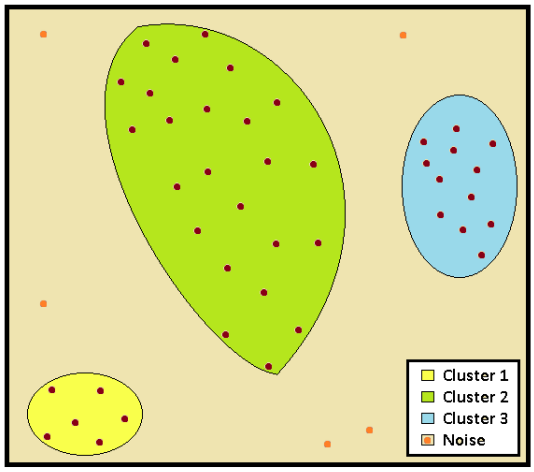


Figure 3.1: An example of a dataset with three clusters.

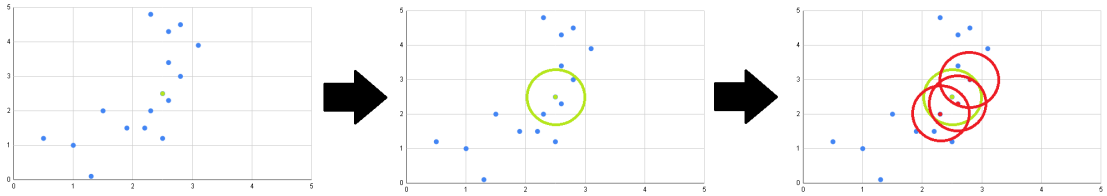


Figure 3.2: DBSCAN cluster creation.

3.2.2 Misconception about time complexity

It was long believed that the algorithm had a lower time complexity than it had since the original paper claimed it to be $O(n \log n)$ [16]. Unfortunately, this has caused so much confusion that major text books [31, 33, 42] and papers [3, 6, 13, 28, 30, 39, 49] have restated this claim [17]. The correct time complexity of this algorithm is, in fact, $O(n^2)$. This is because the distances are calculated from every point to every point, as seen in Figure 3.3.

3.2.3 The algorithm

For some conditions, $O(n^2)$ is acceptable, but if the algorithm runs with increasingly more input points, it is preferred to be as efficient as possible. More precisely, what DBSCAN does is that it first identifies a core point by calculating the number of neighbors it has within a radius of ϵ, counting them as border points. It then labels them as a cluster and propagates further using said neighbors until there are no more new neighbors within a distance of ϵ that were not visited. This process can be viewed in Figure 3.2, where first, a point is selected in the window to the left, then bordering points are counted in the middle window, and lastly, propagating throughout the neighbors in the last window to the right. The pseudo-code in Algorithm 1 shows the DBSCAN algorithm implementation[1]. Finally, points considered outliers, i.e., not part of a cluster, are labeled as noise.

[1]In the algorithm, the "visited" implementation is open to interpretation.

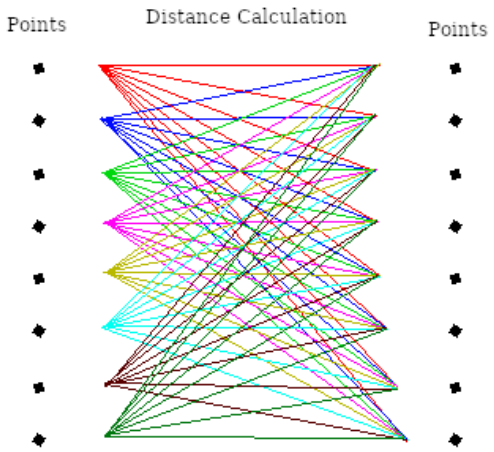


Figure 3.3: Visualization of distance calculations for every point.

The clustering part of the algorithm in DBSCAN is reasonably quick by itself, as it skips all visited points. The part of the algorithm with the highest time complexity is in the method *RangeQuery*. There have been numerous implementations as a result of the high time complexity. One implementation only uses a $distanceTo()$ function, which is the least effective way to implement it because it continuously calculates the distances every time it is called. A more standard way to implement it is by using a distance matrix that stores the pairwise distances. Nonetheless, both of these implementations for *RangeQuery* have a time complexity of $O(n^2)$ since $distanceTo()$ calculation is just converted into memory access, which likewise takes time. This thesis will investigate various ways to improve the *RangeQuery*.

3.2.4 Optimization of DBSCAN

The hosting company has already optimized the algorithm compared to the algorithms in the base form. It uses less overhead and avoids doing unnecessary computations. Nonetheless, these improvements do not improve the time complexity of DBSCAN. The time complexity means the radar struggles when more points are introduced. This algorithm will be referred to as Current-DBSCAN from now on.

3.3 G13 DBSCAN

The G13 DBSCAN algorithm is a grid-based algorithm that improves DBSCAN's time complexity, changing it from $O(n^2)$ to $O(n \log n)$. Gunawan initially presented it in 2013, hence the name G13 [20]. Since the algorithm is grid-based, a data structure is required for the grid. The data structure we chose for this is a hash table because insertion and retrieval are $O(1)$.

3.3.1 The algorithm

This algorithm is divided into four sections; partitioning, core identification, cluster formation by core merging, and border point identification.

Algorithm 1 Original DBSCAN algorithm [16].

```
1:  procedure DBSCAN(𝒟, ε, MinPts)
2:      C ← 0
3:      for each p in 𝒟 do
4:          if p = visited then
5:              continue
6:          𝒩 ← RangeQuery(𝒟, p, ε)
7:          if |𝒩| < MinPts then
8:              label p ← NOISE
9:              continue
10:         C ← C + 1
11:         label p ← C
12:         while 𝒩 ≠ ∅ do
13:             q ← pop from 𝒩
14:             if label q = NOISE then
15:                 label q ← C
16:             if q = visited then
17:                 continue
18:             label q ← C
19:             𝒩_N ← RangeQuery(𝒟, q, ε)
20:             if |𝒩_N| ≥ MinPts then
21:                 𝒩 ← 𝒩 ∪ 𝒩_N
22:     return labels(𝒟)
```

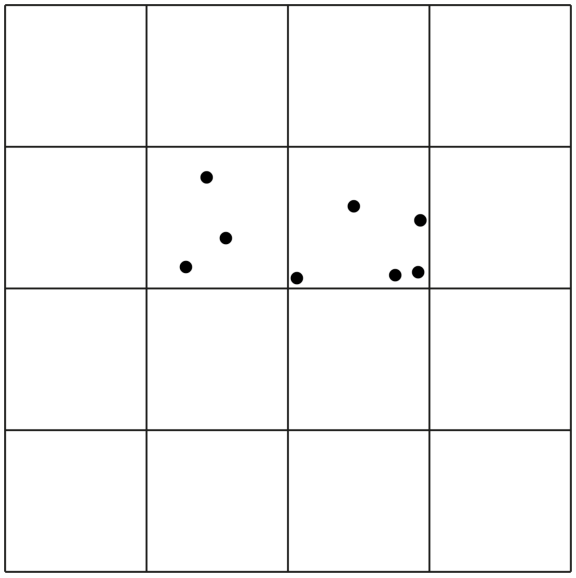


Figure 3.4: Eight points in a grid with sixteen cells.

3.3.2 Partitioning

For the first section, we form a grid $\mathcal{T}$, in which each cell in the grid has sides of length $\epsilon/\sqrt{2}$. Given that each cell is a square, and ϵ is the maximum distance in which a neighbor is counted, that can be used as the square's diagonal, providing the maximum possible distance within the square. This would form a right triangle with two equal sides. Using the Pythagorean theorem, we can calculate[2] the sides s; $s^2 + s^2 = \epsilon^2 \iff s^2 = (\epsilon^2)/2 \iff s = \pm\epsilon/\sqrt{2}$. The side lengths are taken advantage of when identifying core points.

To create the grid which holds all cells, we used a hash table that maps each point in the graph $\mathcal{G}(i, j)$ to a cell, $\mathcal{G} : \mathbb{R}^2 \to \mathbb{Z}$. Each cell $c \in \mathcal{T}$ may contain points. In Figure 3.4, we see that most cells are empty; however, the cells in the middle of the grid have three and five points respectively in them. As mentioned earlier, the hash table stores the cells and the points pertaining to them.

One assumption we make is that no point in $\mathcal{P}$, the set of all points, falls directly on the edge of a cell. This would cause the point to be counted twice, which is not intended. In Figure 3.5, one can see that two adjacent cells would compete over the ownership of the red point unless this is handled somehow. One way to mitigate this is to check if any point exists on the edge, and if it does, one could move $\mathcal{T}$ infinitesimally to one side or the other, so the point landed near the edge instead of on it. Since the implementation is a hash table, it does not cause any issues because the hash function rounds up the coordinates of the points deterministically.

A point $p \in \mathcal{P}$ has either the label *Core* or *Non-core*. $\mathcal{P}(c)$ denotes the set of points the cell c covers. A cell c is considered non-empty if $|\mathcal{P}(c)| > 0$, and the cell is considered core if any point in c is labeled as core.

[2]Negative distances are not used, so we are only interested in the positive results of square root.

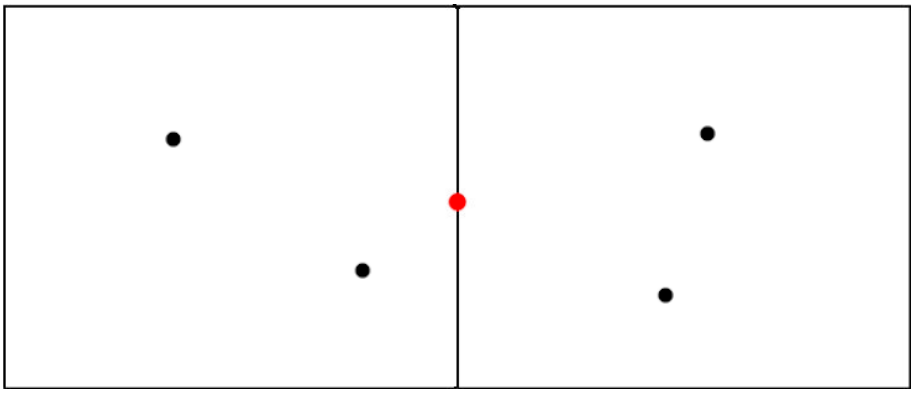


Figure 3.5: Scenario where a point (red) is on the edge between two adjacent cells.

3.3.3 Determining core points

The side lengths secure a certain amount of core points with minimal computation because all points within cells will have distances less than ϵ, meaning that if the number of points within one cell reaches or exceeds ***minPts***, such points will immediately be labeled as core points. However, if this condition is not fulfilled, it is required to iterate through all points in the cell and manually calculate the distances between those points and points in neighboring cells up to two steps away from the current cell. As shown in Figure 3.6, the neighbors of the neighbors are also needed because a single cell has a side length slightly smaller than ϵ. There is a possibility that points are coordinated near the edges of the cells[3]. Neighboring cells' function is thus defined as $\mathcal{N}(c) = \{c_n | (c, c_n) \in T, \exists p \in c \exists p_n \in c_n : |p - p_n| < \frac{3\epsilon}{2}\}$. This ensures that all core points are found and is still more efficient than the original DBSCAN because the maximum number of neighboring cells is constant (21 cells). The constant time for looking up neighboring cells and iterating over each point ensures a time complexity of $O(n)$ for this step, where n is ***minPts***. The algorithm for this step is provided in Algorithm 2 [20].

3.3.4 Merging core points

The third section of this algorithm is merging. All core cells in $\mathcal{T}$ are denoted by $\mathcal{S}_{core}$. Given a core cell $c_1 \in \mathcal{S}_{core}$ and one of its neighboring core cells c_2, the merging step is:

- If $|p_1 - p_2| < \epsilon$ where $p_1 \in \mathcal{P}(c_1)$ and $p_2 \in \mathcal{P}(c_2)$ then c_1, c_2 are part of the same cluster $\mathfrak{C}(c_1) \cup \mathfrak{C}(c_2)$ where the $\mathfrak{C}$ function denotes all points that are part of that cluster.
- Otherwise, $\mathfrak{C}(c_1)$ and $\mathfrak{C}(c_2)$ are two separate clusters.

Fortunately, this step is fast whenever two neighboring core cells have a reasonable amount of points in them. Unfortunately, performing this step with brute force is still considered $O(n^2)$ in the worst case. However, Gunawan showed an approach that gives a time complexity of $O(n \log n)$ [20]. In Figure 3.7, two core neighboring cells contain all the points in the grid. A nested loop looking at each $p_1 \in c_1$ and comparing the distance to each $p_2 \in c_2$ will resemble the *RangeQuery* in Algorithm 1. Gunawan showed that this subproblem could be

[3]The image is just for demonstration purposes and is not to scale.

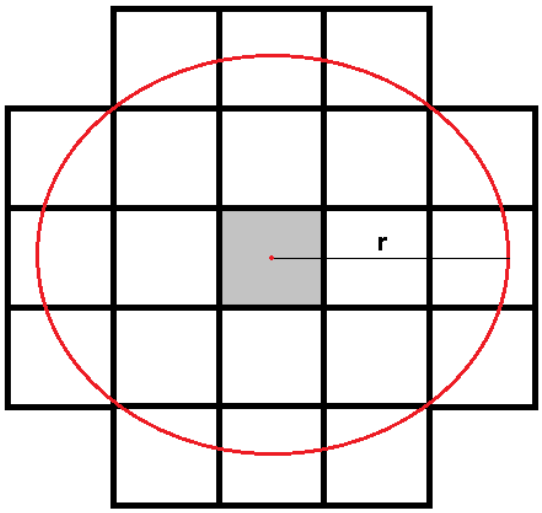


Figure 3.6: Neighboring cells.

The radius r represents the radius from the center of the current cell. This would be calculated as ϵ added with half the length of a cell diagonal, which is $\frac{3}{2}\epsilon$. This radius is chosen to illustrate why 2-step neighbors are needed for the calculation.

Algorithm 2 G-13 DBSCAN: Determining Core Points.

```
1:  procedure DETERMINECOREPOINTS(𝒯, ϵ, minPts)
2:      for each non − empty cell c in 𝒯 do
3:          if 𝒫(c) ≥ minPts then
4:              for each point p in c do
5:                  label p ← CORE
6:          else
7:              for each point p in c do
8:                  nPts ← 0
9:                  for each cell c_𝒩 in 𝒩(c) do
10:                     for each point q in c_𝒩 do
11:                         if dist(p, q) ≤ ϵ then
12:                             nPts ← nPts + 1
13:                             if nPts ≥ minPts then
14:                                 label p ← CORE
15:                                 break
16:                 if nPts ≥ minPts then
17:                     break
```

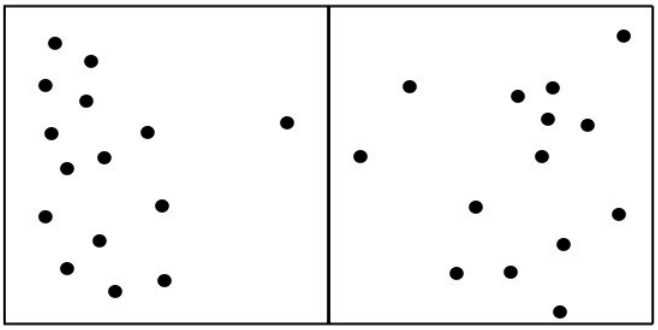


Figure 3.7: Scenario where all points are in two cells c_1 and c_2.

G13 Algorithm	Merge Time Complexity	Merge Data Structure
Brute-force	$O(n^2)$	No data structure
Voronoi Diagram	$O(n \log n)$	Voronoi diagram (V-Diagram)
Delaunay triangulation	$O(n \log n)$	Dual-graph of V-Diagram → Minimal Spanning Tree
R-tree query	$O(n^2)$	Rectangle Tree
Radix sort	$O(nd)$	Divide and Conquer [7, 18]

Table 3.1: G13 algorithms with their respective time complexities and data structures for merging core points.

regarded as a bichromatic closest pair (BCP) problem, where each p_1 will find the closest p_2 in the set $\mathcal{P}(c_2)$. Going through every point p_1 in $\mathcal{P}(c_1)$ gives $O(n)$ and for example, using a Voronoi diagram to query the closest point takes $O(\log n)$ giving a running time complexity of $O(n \log n)$.

Using a Voronoi diagram is one of many ways to solve a BCP problem. We have investigated the alternatives shown in Table 3.1.

Voronoi diagram

A Voronoi diagram can be constructed in $O(n \log n)$ time. After the diagram is constructed, we query the diagram for each point in the opposite cell.

Delaunay triangulation

A Delaunay graph can be constructed using a Voronoi diagram corresponding to its dual graph. For this graph, there exists a *Minimum Spanning Tree* (MST). The MST has the shortest edge crossing between the two cells. A search algorithm like *Depth First Search* can be used [5].

R-tree query

Much like some implementations of the original DBSCAN implementing a Rectangle tree to reduce the average run time to $\Theta(n \log(n))$, the idea is the same for G13. For each cell duo, we construct an R-tree, costing some overhead but decreasing average query time potentially. An implementation of this was programmed and tested. However, the results could have been better. We reasoned that constructing one big R-tree for all the points, instead of for each pair of neighboring points, was preferable, and we abandoned further investigation.

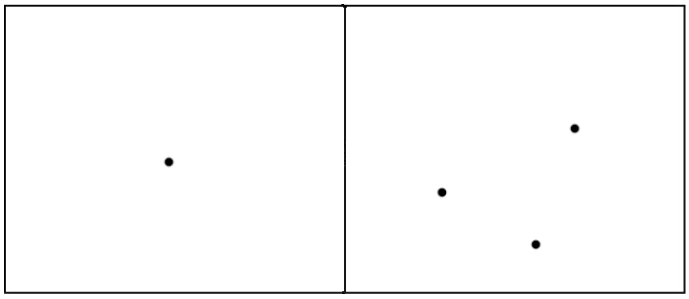


Figure 3.8: The left cell is the current cell, and the right cell is the neighbor and merging candidate.

Radix sort

Sorting with radix requires $O(nd)$ where n is the size of the dataset, and d is the number of digits for the largest number. When this sort is done on the x-axis, we can create strips for each y-axis element that satisfy $dist(p1_y, p2_y) < \epsilon$. This assumes that the points are integers. If they are not, a transformation to an integer could be made. A point like $(32.123, 5, 8530)$ with $\epsilon = 0.5$ could be transformed to the integer point $(32123, 5853)$ where $\epsilon = 500$. Kirkpatrick and Reisch showed in 1983 that integers in any range can be sorted in $O(n)$ time using radix sort (with a few assumptions) [27]. Now, solving a unit-spherical emptiness checking problem, Gan and Tao managed to achieve linear time for merging [18].

Brute-force

Our contribution to improving the G13 algorithm is introducing a brute-force implementation with less overhead to better work for embedded systems. In contrast to the original DBSCAN that does a *RangeQuery* for each point globally, our implementation does this *RangeQuery* locally. In Figure 3.8, the neighboring cell only has a few points. Going through each point and deciding whether to merge the clusters would go quickly for our current cell. However, in Figure 3.9, we see a busy cell with many points in both our current cell and the neighbor. Here, even if we have localized the brute force, the worst case is having to go through all the points on the left and compare it to all the points to the right, and much time for the whole algorithm is going to be spent doing these $O(n^2)$ computations. Nonetheless, we reason that the worst case will not be a problem for most average grids.

3.3.5 Determining border points

The final step in this algorithm is shown in the pseudo-code in Algorithm 3. This step decides how to handle the non-core points. Given a non-core point $p_1 \in \mathcal{P}_{non-core}$, if $\exists p_2 \in \mathcal{P}_{core}(dist(p_1, p_2) < \epsilon)$ then the point is considered a border point and is assigned to the closest cluster. Else, the point is noise.

Remark. Whenever we label a point as a border point, this means that there is *at least* one cluster that this point could be part of. We decide to take the cluster closest to the point if there is more than one cluster. However, this is not specified in the original algorithm and could therefore give different results depending on the order of the input data. One example is in Figure 3.10, where the difference between DBSCAN and G13 implementation

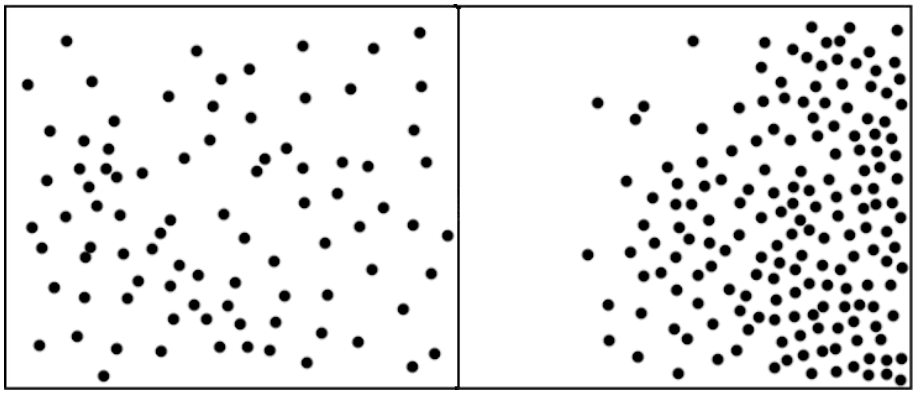


Figure 3.9: Example of a busy merge between the current cell to the left and the neighboring candidate cell on the right.

is straightforward. Here, every point is the same except one in the middle. G13 includes this point in the right cluster because the closest core point is to the right side, while regular DBSCAN, given the input order, chose the left-sided core point because it came first in the queue.

Algorithm 3 G-13 DBSCAN: Determining Border Points.

```
1:  procedure determineBorderPoints(𝒯, ϵ)
2:      for each non − empty cell c in 𝒯 do
3:          for each point p in c do
4:              if label p ≠ CORE then
5:                  q ← NULL
6:                  for each cell c_N in 𝒩(c) do
7:                      p_tmp ← NearestCorePoint(p, c_N)
8:                      if dist(p, p_tmp) ≤ dist(p, q) then
9:                          q ← p_tmp
10:                 if q ≠ NULL then
11:                     label p ← label q
12:                 else
13:                     label p ← NOISE
```

3.3.6 Shortcomings

The G13 algorithm manages to satisfy a lower worst-case time complexity than the other algorithms that came before. However, this algorithm only works in two dimensions, which limits its use cases. The radix implementation of G13 even delivers a linear time complexity. Nevertheless, it comes with the requirement that all points be integers. Next, we will look at an algorithm that builds upon G13 to achieve a lower time complexity and works in higher dimensions.

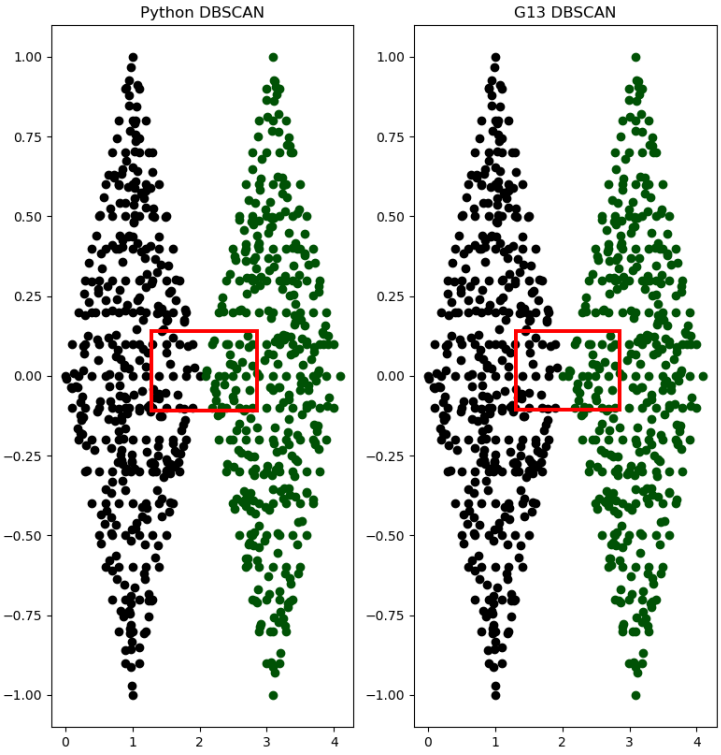


Figure 3.10: Scenario where regular DBSCAN and G13 differ.

3.4 ρ-approximate DBSCAN

Gan and Tao first introduced ρ-approximate DBSCAN in 2015 [17]. The idea is to introduce an approximation parameter ρ besides $minPts$ and ϵ. By introducing this variable, Gan and Tao reduce the time complexity for lower dimensions ($d \leq 7$) to $O(n)$. For the proof and derivation of the time complexity, we refer to Gan and Tao's newer, more comprehensive paper [18].

3.4.1 Changes from G13 implementation

Instead of a point $q \in \mathcal{P}$ needing a minimal number of points within a distance of ϵ to be considered core, it now requires a distance of $\epsilon(1+\rho)$. With this change, the connectivity inside a cluster also changes, where there is no longer any guarantee that all the points inside a cluster are density-reachable from any given $q \in \mathfrak{C}(\mathcal{G}(q))$. For a cluster $\mathfrak{K}$ to have connectivity, it is required that any two points $p_1, p_2 \in \mathfrak{K}$ has a point $p \in \mathfrak{K}$ such that p_1 and p_2 are density-reachable from p. For two points to be density-reachable from each other, there must be a sequence of core points where each new point is within ϵ from the previous one. This definition is no longer valid as it was in G13. In Figure 3.11, there is a clear cluster between four points. However, the fifth point is not within the ϵ range but within the $\epsilon(1+\rho)$ range. This new definition means points may or may not be counted in the cluster. The fifth point is now ρ-approximate density-reachable.

This change is helpful because it allows for an effective *RangeQuery* method in the merging step. Aside from this, there are no changes to G13 in the 2D case.

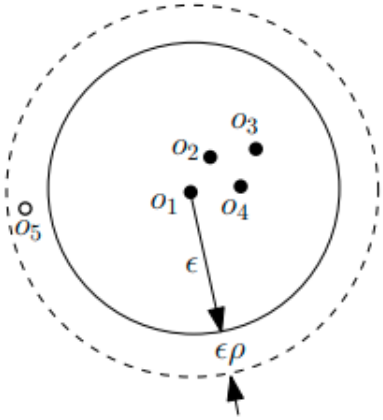


Figure 3.11: Picture borrowed from [18] that shows that all points inside ϵ are counted, and points between $\epsilon(1+\rho)$ and ϵ might be counted in the *RangeQuery*.

Sandwich theorem

From Figure 3.11 and the explanation, we can see that points inside $\epsilon(1+\rho)$ could be counted or not, while all points inside ϵ will be in the cluster. The sandwich quality guarantee assures that if the clusters change because of the approximation, then it will, in the worst case, result in the same clusters as running regular DBSCAN with ϵ as $\epsilon(1+\rho)$.

Theorem 3.4.1 (Sandwich Quality Guarantee)**.** *Proved and defined in [18].*

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

Where C_1 *as the set of clusters of DBSCAN with parameters*$(\epsilon, \textbf{\textit{minPts}})$, C_2 *as the set of clusters of DBSCAN with parameters*$(\epsilon(1+\rho), \textbf{\textit{minPts}})$*, and* C *as an arbitrary set of clusters that is a legal result of* $(\epsilon, \textbf{\textit{minPts}}, \rho)$*-approx-DBSCAN.*

Remark. The sandwich theorem makes no guarantees that all points will be in the same clusters as running regular DBSCAN with $\epsilon_{new} = \epsilon(1+\rho)$, only that if a cluster is merged, it would be the same merged cluster as ϵ_{new}. An example of this can be viewed in Figure 3.12, where we can see a different result between the two algorithms when $\epsilon = 1$; however, this difference disappears when a slightly higher ϵ value is selected for DBSCAN. In other words, the value of ϵ was not ideal in the first place. In this case, the two clusters on the left were merged by ρ-approximate DBSCAN, but that does not need to happen; it could also be clusters that would differ between $\epsilon = 1.01$ and $\epsilon = 1$ for regular DBSCAN but remain as the same as DBSCAN with $\epsilon = 1$.

Approximate range counting

The big difference between G13 and ρ-approximate is how the *RangeQuery* is done. Instead of dividing up the grid $\mathcal{G}$ into static cells with the same size, a quad-tree-like structure is created where each layer has cells with decreasing width until the side length is at most $\epsilon\rho/\sqrt{d}$, where

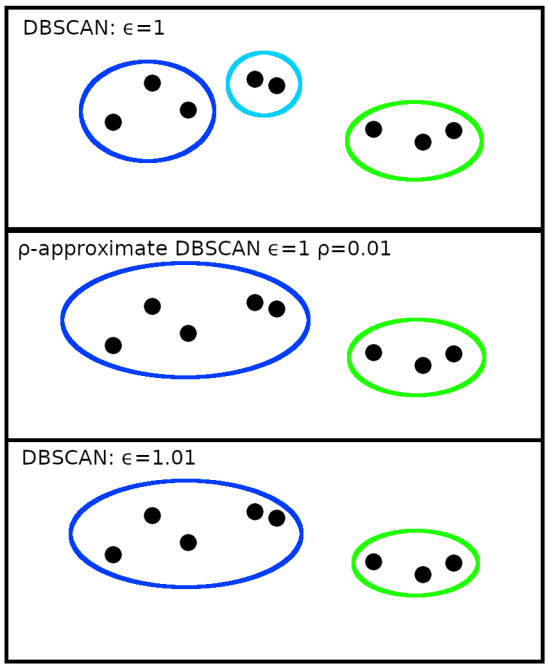


Figure 3.12: A fictional example of how the approximate *RangeQuery* could change the clusters if a poor ϵ is selected.

d is the number of dimensions. This will create a hierarchy $\mathcal{H}$ where the amount of levels is $\max\{1, 1 + \lceil \log_2(\frac{1}{\rho}) \rceil\}$, which is $O(1)$ in the number of input points [18]. The tree takes linear time to construct, and a query will be, at most, a traversal of the tree from the root to a leaf, which takes constant time. This is done for each point, giving a total time complexity of $O(n)$ in the end.

Remark. The tree is quad tree-like because, depending on the number of dimensions, it can be viewed as the quadrants in a coordinate system, giving at most 2^d number of non-empty cells per level of the tree structure. This creates four squares for two dimensions, eight cubes for three dimensions, and sixteen tesseracts for four dimensions, etc. Here we can also see the inherent problem with increasing the number of dimensions, which exponentially increases the overhead if we decide actually to build all the parts of the tree.

For each point, the query will look like this:

- If the current cell is outside the range, ignore it.
- If the cell is fully covered by the radius $\epsilon(1 + \rho)$, add $|\mathcal{P}(C)|$ to the answer.
- Otherwise, check if the cell is a leaf cell in $\mathcal{H}$, then add $|\mathcal{P}(C)|$ to the answer, else recursively go through each child node.

The number of points will be returned. If there are points, it means that a point exists in a neighboring cell that is part of the same cluster, so then merge; else, if no points are found, they are separate clusters.

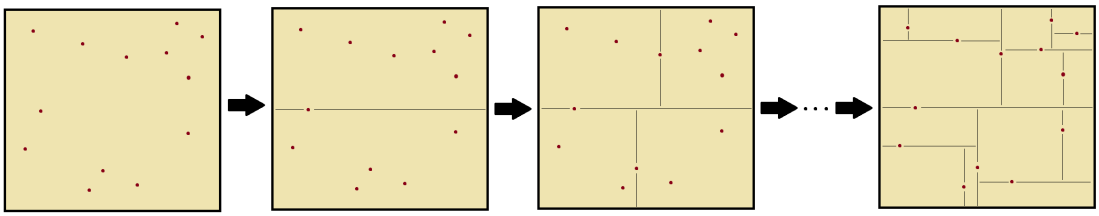


Figure 3.13: Visualization of building a *k*-d tree.

3.5 KDT-DBSCAN

Another DBSCAN technique that claims minimal time complexity and works well in larger dimensions is presented below. A *k*-d tree (*k*-dimensional tree) is a multidimensional binary tree structure that not only provides an $O(n \log n)$ time complexity but also creates the possibility of clustering over data in more than two dimensions. This solution creates the potential for improving radars by increasing the number of parameters that can be used in clustering. The *k*-d tree data structure is used to find the nearest neighbor. However, this makes it close enough to the purpose of DBSCAN's *RangeQuery* that it is possible to adapt *k*-d tree to it [32].

3.5.1 Building the tree

The idea behind *k*-d tree is to perform partitions using the median point in each dimension. There are multiple ways to build a tree, and each method has drawbacks. The most standard way to build the tree is as follows. First, the data is partitioned at the median in the first dimension while assigning the median point as the root, then partition each sub-partition in the second dimension (separately) while assigning the median points there as the two children nodes and then continue performing the same partitions for each dimension until there are no more points to partition (leaf nodes)[4]. This concept will build the binary search tree (BST), which will be utilized later [32]. The BST can be illustrated in Figure 3.13. In addition, the tree equivalent of this can be seen in Figure 3.14.

It is not easy to efficiently build the tree because the median continuously needs to be calculated for each partitioning. The mentioned method usually uses an efficient sorting technique to find the median, such as merge-sort or quick-sort, and then picks the median. These two techniques are optimized list sorting algorithms with $O(n \log(n))$ time complexity. Afterward, it splits the list at the median (excluding the median) and re-sorts the sub-lists based on the other dimensions for the sub-partitions.

Balancing the tree

Another method is to use the input data directly and balance the tree at the end. This method is also considered inefficient as it is challenging to balance a *k*-d tree after building it because the depth depends on multidimensional values.

[4] After the last dimension is reached, it starts over from the first one again.

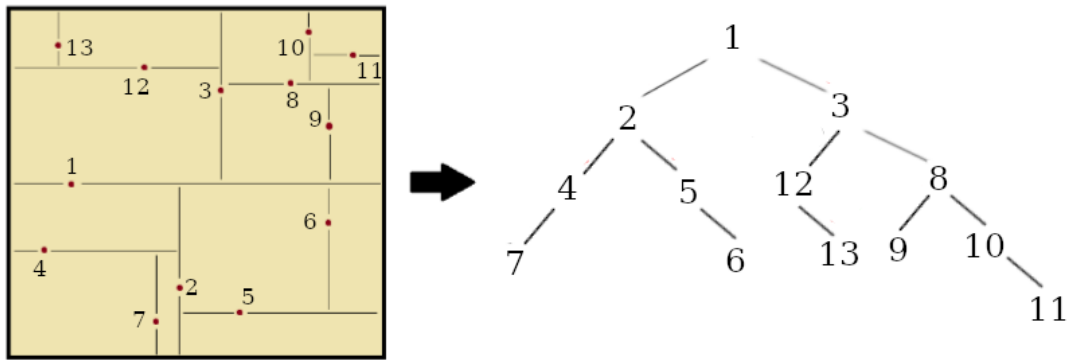


Figure 3.14: BST visualization of k-d tree.

Figure 3.15: List visualization when tree is being built for 3D points.

Building with samples

One method that may be used to build the tree efficiently is by taking a fixed number of samples from the data and finding the median of these points. The advantage of this method is that the performance is independent of input size, meaning it will almost always take the same amount of time. However, variation of execution time may increase when there is a higher number of points[5]. Unfortunately, this method also has drawbacks, as it does not guarantee a balanced tree.

Additionally, this process still requires sorting or traversing the lists when split. This may be better visualized in Figure 3.15. In this figure, given that the median is selected, the list should be correctly sorted and split up to be able to go further down the stack (taking $O(n)$ time complexity instead of $O(1)$).

[5]Variation in execution time may be improved by taking $k \log_{10} n$ amount of samples instead where n is the number of points, and k is a constant that is chosen in advance. We have decided not to do this accuracy improvement as the radars do not use an extremely high number of points.

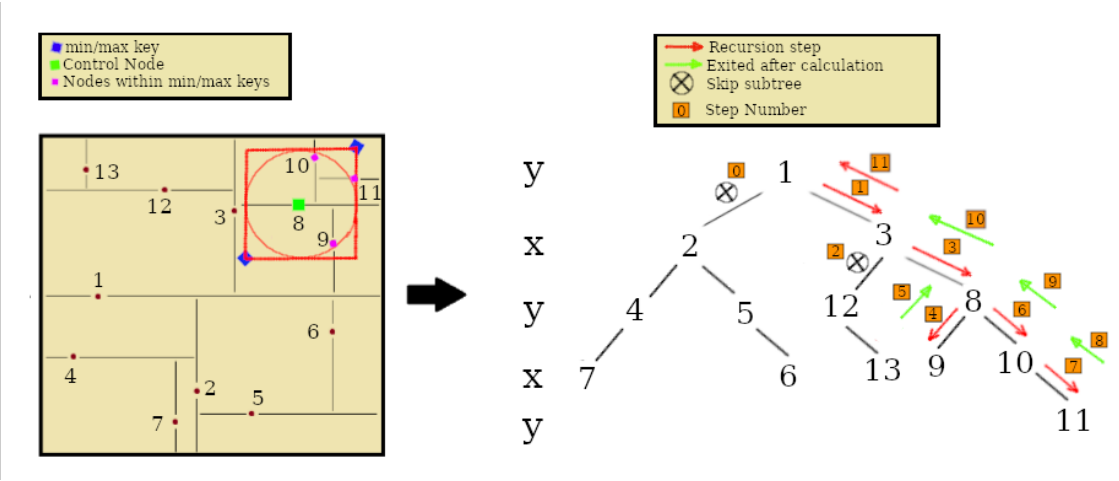


Figure 3.16: *RangeQuery* for *k*-d tree.

3.5.2 *RangeQuery*

After building the tree, a *RangeQuery* is performed recursively. However, before that, two points called ***minKey*** and ***maxKey*** are used to make two decisions by using their split coordinates[6]; if the subtree should be traversed and a node should be used for calculation. These points are derived from the node coordinates for which we want to find neighbors. To calculate ***minKey***, we subtract ϵ from all dimensions, and for ***maxKey***, we add ϵ to all dimensions.

Once we have the ***minKey*** and ***maxKey*** values, the code traverses the tree and checks if a subtree should be explored based on these points. Only points within ***minKey*** and ***maxKey*** split coordinates have their distances calculated [32]. The KDT-DBSCAN *RangeQuery* can be seen in Algorithm 4 and can be visualized in Figure 3.16 (derived from Figure 3.14). As demonstrated, distance in this context was calculated four times, whereas the original DBSCAN would perform the calculation twelve times.

[6]Split coordinate is defined as the coordinate value of the current dimension of the tree.

Algorithm 4 k-d tree.

```
1:  procedure REC_KDT(minKey, maxKey, current, control, 𝒩, ϵ, depth)
2:      dim_cnt ← number of dimensions
3:      dim ← depth mod dim_cnt
4:      greq ← current value in dim ≥ minKey value in dim
5:      leeq ← current value in dim ≤ maxKet value in dim
6:      if greq and leeq then
7:          if dist(current, control) < ϵ then
8:              𝒩 ← 𝒩 ∪ current
9:      if greq and curr.has_left then
10:         REC_KDT(minKey, maxKey, current.left, control, 𝒩, ϵ, depth + 1)
11:     if leeq and curr.has_right then
12:         REC_KDT(minKey, maxKey, current.right, control, 𝒩, ϵ, depth + 1)
13: procedure KDT(root, control, ϵ)
14:     maxKey ← control coordinates + {ϵ, ..., ϵ}
15:     minKey ← control coordinates − {ϵ, ..., ϵ}
16:     𝒩 ← ∅
17:     REC_KDT(minKey, maxKey, root, control, 𝒩, ϵ, 0)
18:     return 𝒩
```

Chapter 4
Methodology

This chapter covers the materials used, the kinds of datasets we examined, and how we conducted the tests.

4.1 Material

Most of the material used for this paper is software. However, there is still physical material used. Firstly, we needed to test the algorithms on the radar units; therefore, the hosting company provided us with two radar units. In addition, the hosting company provided two computers, one of which we ran the tests on. The specifications are as follows; its CPU is AMD Ryzen 9 3900X clocked at **3.9GHz** with 32 GB RAM. An ARM Cortex-A53 CPU powers the radar — this CPU is a dual-issue in-order superscalar computer that implements the ARMv8-A instruction set.

4.2 Testing

One of the limitations of the various algorithms is testing. Profiling on the actual units takes a long time, is difficult to automate, and can be tedious when implementing and debugging the algorithms. For this reason, the tests are divided into two sections; synthetic tests and benchmarks on real data. The DBSCAN algorithms that were used are shown in Table 4.1.

Gan and Tao [1] have permitted us to use their latest binary file containing the different algorithms they have developed to be used in comparison with the algorithms we have developed, and the DBSCAN variant provided to us by the hosting company. The binary was compiled by "GCC: (Ubuntu 4.8.4-2ubuntu1 14.04.3) 4.8.4", and the flag used was "-O3". This binary includes a plethora of DBSCAN algorithms. The ones we have used for our comparisons are marked as ***G&T*** in Table 4.1. Our programs have been compiled with "GCC:

[1]The original authors of "DBSCAN revisited: Mis-claim, un-fixability, and approximation".

Algorithm	Description	Compiled by
1996 DBSCAN	The original DBSCAN algorithm.	***J&J***
Current-DBSCAN	Proprietary implementation of DBSCAN, based on the original algorithm.	***J&J***
Brute-G13 DBSCAN	Our G13 implementation utilizes brute force when merging cores.	***J&J***
Voronoi-G13 DBSCAN	G13 that utilizes Voronoi diagram.	***G&T***
Delaunay-G13 DBSCAN	G13 with Delaunay merging.	***G&T***
R-Tree DBSCAN	R-Tree *RangeQuery*.	***G&T***
ρ-approximate DBSCAN	Approximation with $\rho = 0.01$.	***G&T*** and ***J&J***
Exact DBSCAN	Radix sort.	***G&T***
KDT-DBSCAN	1996 DBSCAN with k-d tree *RangeQuery* implementation	***J&J***

Table 4.1: Algorithms used with summary.

(Debian 10.2.1-6) 10.2.1 20210110" with the flag "-O3". These are referred to as ***J&J*** in the table.

4.2.1 Synthetic testing

In the first section of the testing, synthetic tests are performed without using the radar units, and then one chosen implementation is used on the radar. Performance-wise, these tests are expected to differ slightly compared to the actual benchmarks on real data, but this will provide a more comprehensive visualization of the algorithms' differences and their execution times. Another reason for implementing such tests is to contain tests that are both simple and quick to run. One example of a test is in Figure 3.10 where the *twodiamond* test was used.

With the assumption that speed improvements from small changes can carry over to the radar unit, we can iteratively go through changes from our computers and see if the improvement carries across to the radar unit. During the development of the algorithms, these tests also greatly aided the debugging of our DBSCAN implementations.

The following introduces a step-by-step walk-through of how the synthetic tests are carried out. In these steps, the n variable represents the number of times the tests are to be repeated.

1. Test Original DBSCAN on datasets.
2. Test Current-DBSCAN on datasets.
3. Test Gan and Tao's algorithms on datasets.
4. Record execution times and save the results to the database.
5. Repeat from Step 1 n times.
6. Implement and debug proposed algorithm.
7. Test new DBSCAN algorithm on datasets.
8. Record execution times and save the results to the database.
9. Repeat from Step 7 n times.
10. Repeat from Step 6 for other algorithms until done.

4.2.2 Benchmarks of real datasets

The latter tests that will be performed are benchmarks for the real datasets mentioned in Section 4.3.1. These benchmarks are done on the desktop computer and the radar. This is because profiling the algorithms on only the radar would take too long. We would have to run them many times to get stable measurements, and that would be for each algorithm and dataset. Performing these runs on the computer saves time and would make it possible to automatize the whole process easily. Ultimately, one algorithm is used on the radar to compare with the Current-DBSCAN. The following presents the step-to-step workflow of how the benchmarks are carried out on the desktop computer.

1. Test Original DBSCAN on datasets.
2. Test Current-DBSCAN on datasets.
3. Test Gan and Tao's algorithms on datasets.
4. Benchmark execution times and save the results to the database.
5. Repeat from Step 1 n times.
6. Test new DBSCAN algorithm on datasets.
7. Benchmark execution times and save the results to the database.
8. Repeat from Step 6 n times.
9. Repeat from Step 6 for other algorithms until done.

Even when automated, this process requires time before we get results, which is the primary reason we try to improve the desktop computers before we move and test the implementation on the radar unit.

4.2.3 Multidimensional synthetic data

To test the scalability of the various implementations of DBSCAN, a selection of datasets with Gaussian clusters [24] were used to guarantee proper trends in performance. As can be seen in Table4.2, these datasets increase in number of points and ϵ when there is an increase in dimensions, but it is guaranteed that there are $k = 9$ clusters for all of them ($minPts = 2$ for all of them). Even though they have different dimensions, these datasets have similar clusters; one example can be seen in Figure 4.1 for the *dim3* dataset.

4.2.4 Avoiding cache misses

An aspect to consider is avoiding cache misses when we measure the results. When we run the algorithms, we want to have the data hot in the cache when measuring. We do this by running the algorithm one time extra in the beginning before we start to measure. This way, variance from initial cache misses and page faults are less likely to affect our measurements.

Name	Dim.	$\|Points\|$	ϵ
dim2	2	2025	30000
dim3	3	2701	30000
dim4	4	3376	30000
dim5	5	4051	30000
dim6	6	4725	30000
dim7	7	5400	30000
dim8	8	6075	30000
dim9	9	6751	40000
dim10	10	7425	40000
dim11	11	8100	40000
dim12	12	8774	40000
dim13	13	9450	50000
dim14	14	10125	50000

Table 4.2: Gaussian Cluster datasets with increasing dimensions.

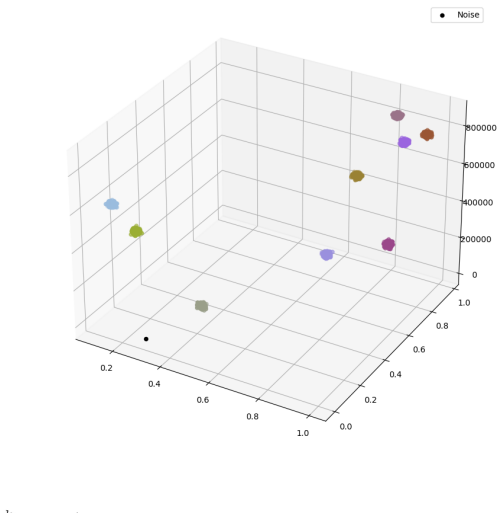


Figure 4.1: KDT-DBSCAN clustering on *dim*3.

dataset	Frame Count	Max Point Count
H2	1145	1674
Crowd	3085	5159
F3	2952	7149
Knutpunkten	969	1098
Roundabout	1852	4011
Vattenhallen	2504	3596

Table 4.3: Real-world Data chosen from Recording Database.

4.3 Data

For the development and validation of the new versions of DBSCAN that we implemented, we decided to use some real-world data and some synthetic datasets that other papers have created for measuring their clustering algorithms.

4.3.1 Real-world dataset

Our hosting company has a database of radar recordings. These recordings vary in size, length, and point intensity. See Appendix B.1 for more details. For this thesis, the required datasets need a high number of points. Table[2] 4.3 shows the recording datasets used, including their frame counts and maximum number of points within that recording.

4.3.2 Synthetic dataset

A selection of datasets have been chosen as inputs for the algorithms. These datasets vary in size, cluster sizes, point concentration, and amount of noise. The datasets include the ones shown in Table 4.4. Essentially, these datasets are text files where every new line is a 2D point that is passed to the DBSCAN algorithms in the same way as the real-world datasets. Another aspect to consider is that these datasets correctly cluster using different ϵ and ***minPts*** values, as they scatter differently. The respective DBSCAN parameters are given in Table 4.4, including their respective file sizes. These parameters are estimated from our end through trial and error. Hence, finding other suitable parameter values that may give different results is possible. In addition, Figure 4.2 shows some listed datasets to visualize better what they represent.

[2]For description, see Appendix B.1 and Figure B.1.

Dataset	Description	$\|Points\|$	ϵ	$minPts$
spiral	Spiral-shaped clusters [12]	312	3	3
compound	Cluster of Zahn's Compound [47]	399	1.5	8
aggregation	Clustering aggregation problem example [19]	788	1	5
twodiamonds	Two diamonds with a connected middle point [43]	800	0.15	8
wingnut	Two closely-connected clusters [11]	1016	0.25	7
chameleon	dataset from chameleon clustering [25]	1971	25	15
a2	Synthetic 2d data from [23]	7500	1000	10
birch	Random sized clusters in random locations [48]	100000	1000	3
GanTaoGen1	Generated datasets using Gan and Tao's seed spreader in the binary [18]	4000	1000	100
GanTaoGen2	Generated datasets using Gan and Tao's seed spreader in the binary [18]	10000	1000	100
GanTaoGen3	Generated datasets using Gan and Tao's seed spreader in the binary [18]	15000	1000	100
GanTaoGen4	Generated datasets using Gan and Tao's seed spreader in the binary [18]	20000	2000	100

Table 4.4: The datasets used for direct testing.

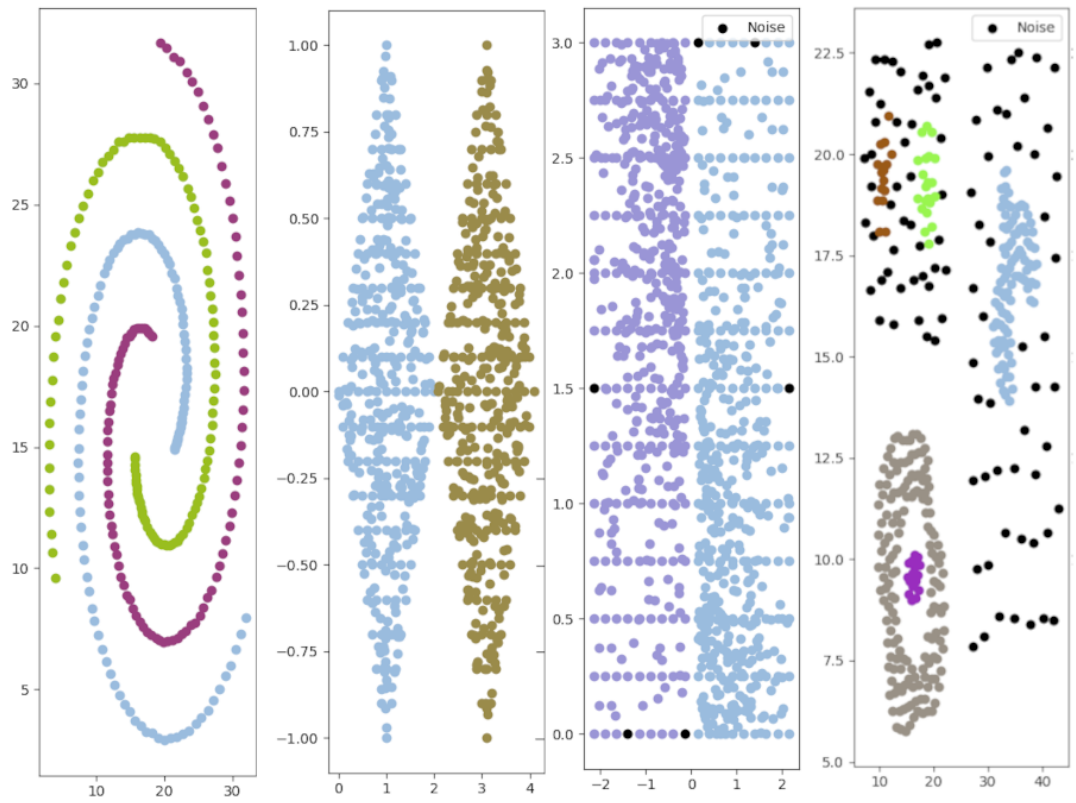


Figure 4.2: Shows *spiral*, *twodiamonds*, *wingnut* and *compound* datasets, in that order.

Chapter 5
Implementation

5.1 G13

G13 was implemented following the steps in the original paper. Merging was not specified and, therefore, developed by us. No optimizations were made — an R-tree when merging cells was tested without satisfactory results. Lastly, creating a Voronoi diagram was attempted but not finished.

5.2 ρ-approximate DBSCAN

Several performance improvements have been made to the algorithm. The focus has been on optimizing the two-dimensional case with a smaller embedded system processor in mind. The general observation was that processing takes longer time than fetching data. So, the goal was to calculate data once and then store it. One example is the lookups done when getting the neighbors. The lookups were done only 21 times for every cell, but the time increased and became a bottleneck. Because of this, getting the neighbors of each cell was calculated first when needed and, after that, stored so that it could easily be fetched when needed.

The same reasoning was applied to the quad tree-like structure. If the number of points is lower than a constant, we brute force the result instead of building the tree. If we must build the tree, we build it once; then we store it for reuse.

When building the tree, we stop dividing into smaller cells whenever only one point is left. In the *RangeQuery* later, we look for if a leaf node has one point in it and then calculate the distance between the two points. Otherwise, there is a risk that more computations are spent on dividing up this point into smaller and smaller cells instead of calculating the distance between two points.

When *RangeQuery* is run, we do not wait for the exact number it returns. If it is the case that the result is non-zero, then we know the two cells should be merged and go right ahead.

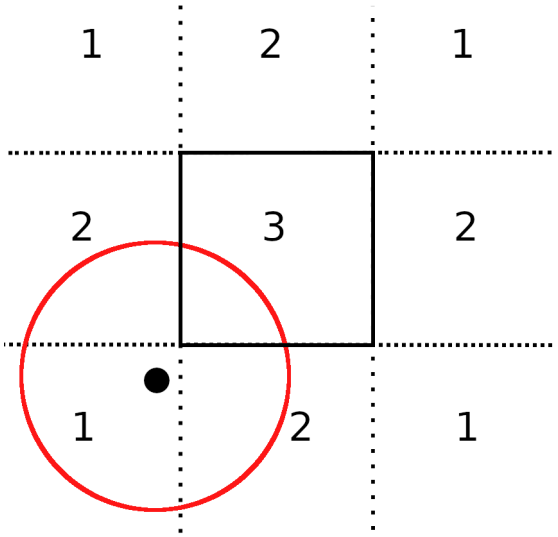


Figure 5.1: Illustration showing the three different choices when deciding if a point is disjoint, intersects, or fully covered.

Even though we have eliminated most lookups in the hash table, it is the main bottleneck for the current implementation. We have experimented with using different hash tables where we have chosen Boost's "unordered_map" as we found it to be, on average, faster compared to the standard library one [38].

The assignment of cluster IDs was changed compared to the implementation from G13 to be done after all clusters are done, so only one iteration through all points is needed.

Optimizing for two dimensions is covered in 5.2.1.

5.2.1 Intersect and fully covered functions

This section covers how we implemented the "outside range" and "fully covered" functions, mentioned in Section 3.4.1.

In Figure 5.1, we can see the three different placements of a point in relation to one of the cells. The first are the points closest to the corners, the second are the points closest to the lines, and the third are points inside the cell.

Whenever a point is closest to the corner of a cell, also seen in Figure 5.1, the first case, then we only need to calculate if the closest corner is within ϵ the distance to determine if the radius intersects the cell. To see if the point's radius fully covers the cell, the opposite corner must be within the distance.

If a point is closest to a line, which would be the second case in Figure 5.1, then a vector that is orthogonal to the line in relation to the point can be created, as seen in Figure 5.2, and its magnitude will be the closest distance. The closest line is used for the intersect function, and the opposite side's line is used for the fully covered function, similar to the corner case method.

Lastly, whenever a point is inside a cell, as is the third case in Figure 5.1, it is guaranteed to be fully covered. Since the diagonal of the widest square is ϵ, and the point's radius is at the least ϵ.

The disjoint function is the converse of the intersect function.

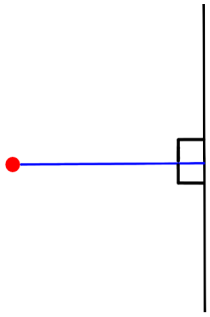


Figure 5.2: A red point, and the corresponding vector in blue that is orthogonal to the black line.

5.3 KDT-DBSCAN

For implementing KDT-DBSCAN, there are two main components; building the tree and performing *RangeQuery*. *RangeQuery* is straightforward as follows Algorithm 4. On the other hand, efficiently building the tree remains tricky. As seen in the article [32], the method for building the tree is not explicitly described.

5.3.1 Implementing *BuildTree*

Each method of implementing *BuildTree* has a disadvantage. For now, attempting to enhance the mechanism for determining the median has the unintended consequence of altering the process of splitting the list on each level. As a result, the sorting mechanism needs to be more balanced. In this case, `std::sort` in the C++ standard library utilizes `introsort`. `Introsort` is a hybrid sorting algorithm that guarantees a time complexity of $O(n \log(n))$. This algorithm starts with `quicksort` and uses `heapsort` when the recursion depth is too high [29].

Data layout

As Section 2.2.3 mentions, choosing the proper data layout is vital to achieving the best execution time possible. In this context, SoA would be more efficient because it packs similar data tighter together, decreasing the number of cache misses and avoiding passing unrelated data.

Index arrays

There are challenges when implementing the SoA approach. One of the main challenges is the one faced when attempting to sort the coordinates according to dimension, and solving this challenge may introduce a new kind of overhead. Fortunately, this can be solved by using index arrays. In this context, two main index arrays can be used. Given that the indices will be unchanged, they can represent a node in the tree. Therefore, when implementing the lefts and rights lists, they can be index arrays instead of arrays of pointers that point to the node, and they may have a value of -1 to represent the absence of the left/right child. In addition,

index arrays are taken advantage of when passing indices through function calls. In other words, `vector<Int>& vec` is passed instead of `vector<shared_ptr<Node»& vec`.

Chapter 6
Results

This chapter will cover the results from the different benchmarks created. The algorithms we implemented were created iteratively. First "1996 DBSCAN" and "Brute-G13" were implemented, and after that, we created "KD-Tree" and "Rho". Therefore, it is important to remember that these benchmarks have been used iteratively to improve performance further, one algorithm after another, first the former and then the latter algorithms.

When presenting the ARM processor results, we will also present a red dotted line that is the max amount of time that DBSCAN is allowed to spend clustering, namely 50 ms as determined in Section 2.6.

In the following results, if any given algorithm is not mentioned for some data point, the algorithm crashes during testing for that data point.

6.1 Synthetic tests

This section shows the data statistics and visualized results for the synthetic tests. We ran the tests eleven times, but the first test was discarded (for cache-misses, see 4.2.4). We discard the first result to provide a more accurate result. The time presented in the figures is the average time from these tests.

6.1.1 Execution time statistics

Figure 6.1 shows a bar graph representing the average execution time values for each implementation for each dataset. The point coordinates were rescaled to have a constant $\epsilon = 1000$ over all datasets because some algorithms require integer values, and some use precise coordinates. Normalizing the point coordinates should not affect execution times for the clustering algorithms. However, the datasets still scatter differently, with different ***minPts*** and different amounts of noise. This is discussed further in Chapter 7.

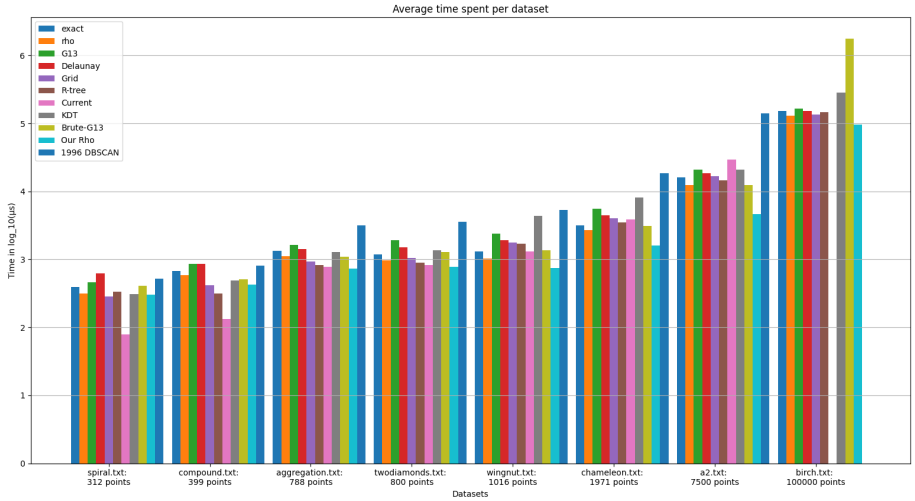


Figure 6.1: Synthetic Data-Execution Time Bar Graph.

Results depending on ϵ values

In Figure 6.2, we see the dataset *a2* and how different ϵ affect the execution times for the algorithms. The minimum number of points used is ten. The ϵ test and $minPts$ test below in Section 6.1.1 has the *a2* dataset chosen because it has the highest amount of points where all the algorithms still run.

Results depending on *minPts* values

Figure 6.3 presents how execution time is affected by varying $minPts$. The test is carried out the same way as in the ϵ test, where $\epsilon = 1000$ and the dataset is a2.

Synthetic results on radar

Figure 6.4 present synthetic results for Current-DBSCAN and our ρ-approximate implementations on the radar system for different kinds of dataset sizes, where the deadline for the dataset of $50\,\text{ms}$ is shown.

6.2 Benchmark

In this section, the real-world datasets (mentioned in Section 4.3.1) are used to benchmark the algorithms, and the results from each algorithm are plotted.

6.2.1 Number of points vs execution Time

The raw benchmark results are not discussed but can be found in Appendix B. Since there are too many points measured to make a straightforward plot using execution time on the y-axis, and the number of frames on the x-axis, an aggregate of a close number of points is used

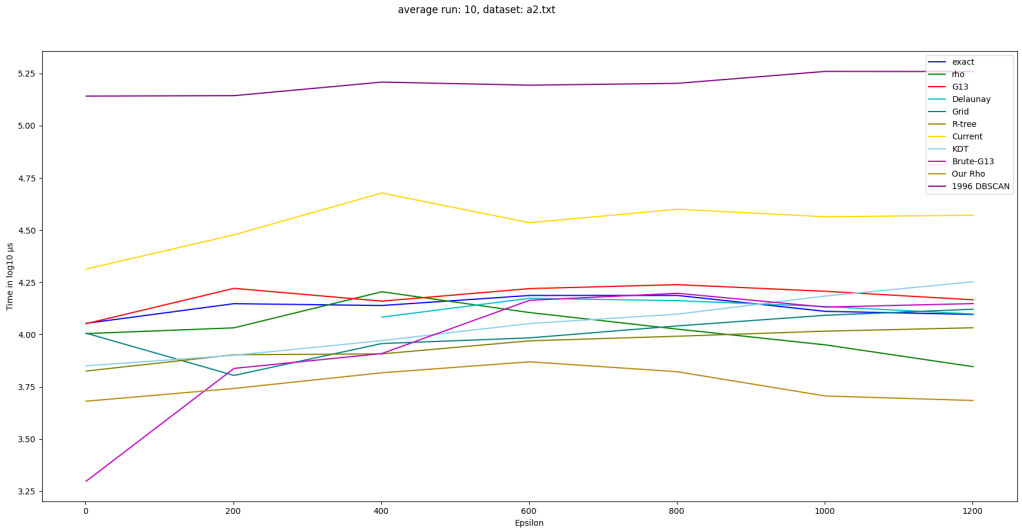


Figure 6.2: The average run time for the algorithms of the dataset a2 with increasing ϵ.

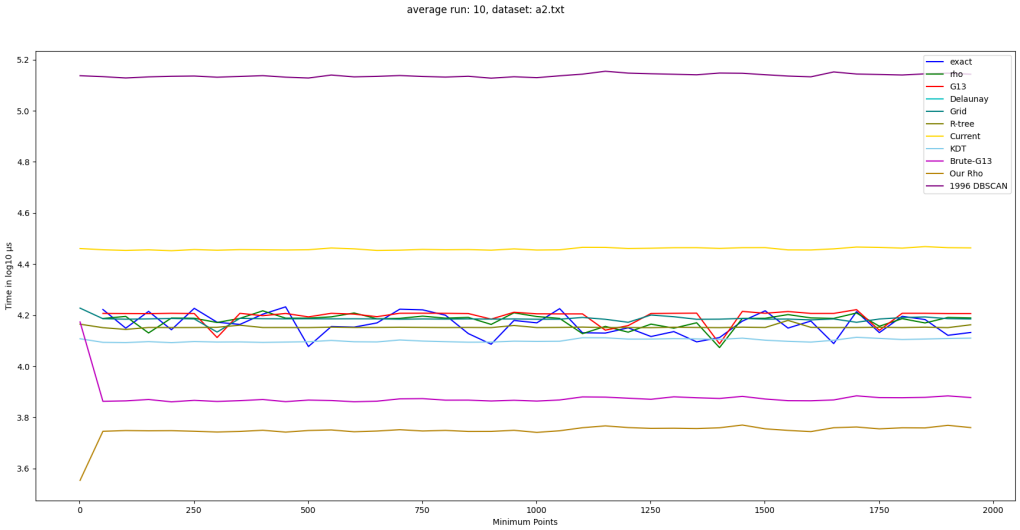
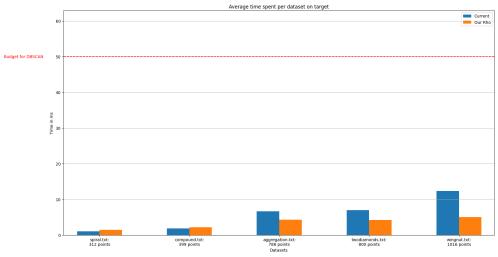
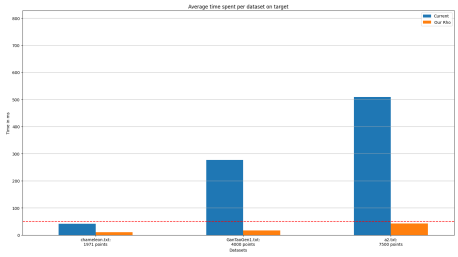


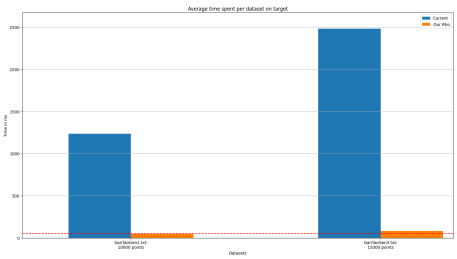
Figure 6.3: The average run time for the algorithms of the dataset a2 with different *minPts*.



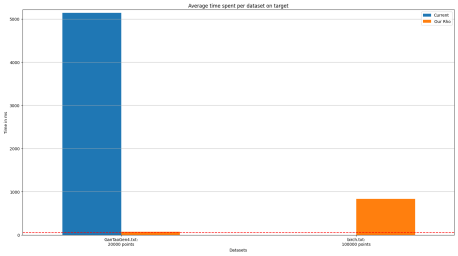
(a) Results for the small datasets on radar.



(b) Results for the medium datasets on radar.



(c) Results for the big datasets on radar.



(d) Results for the huge datasets on radar.

Figure 6.4: Radar synthetic results.

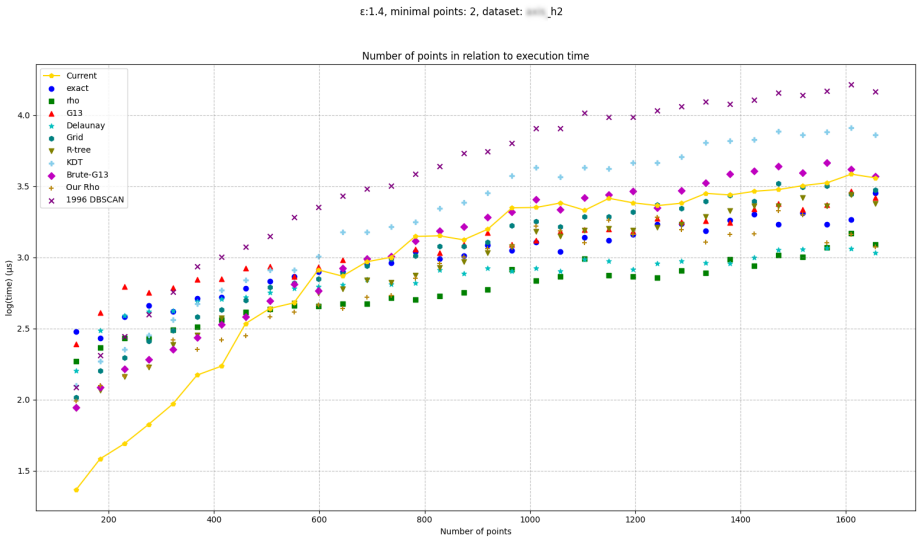


Figure 6.5: *H2* dataset.

instead. The average is calculated for each algorithm. It is also noteworthy that $\log_{10}(\mu s)$ is used for the execution time to improve the scale on the graph.

Results on PC

In Figure 6.5 the *H2* dataset is shown. The results are more varying from data point to data point compared to Figure 6.6. Figure 6.7 also has more even results but with clear distinctions in time between the algorithms. It is also the biggest dataset compared to Figure 6.8, the smallest dataset, viewed from the number of points per frame, with slightly more than 1000 points on the busiest frames. Lastly, both *Vattenhallen*, in Figure 6.9, and *Roundabout* in Figure 6.10 has the line for "Current" not drawing throughout the plots. The reason is that some frames may have, for example, 4000 points and others 3000 points, but no frames contain 3500 points. When this happens, some data points are missing, meaning the line is cut between these data points. In other words, this is to be expected.

Results on radar

The first result is from the *H2* recording site in Figure 6.11. Here we can see that at around 500 points, our algorithm becomes more viable. Next, Figure 6.12 shows the *F3* recording where the number of points starts at around 2800 points and still is within budget at around 7000 points. Figure 6.13 shows *Crowd* where we see a similar trend. *Knutpunkten* in Figure 6.14 is similar to *H2*, not having many frames with a high number of points, and with the same intersection at around 500 points. *Roundabout* in Figure 6.15 and *Vattenhallen* in Figure 6.16 both have some lines from the interpolation not showing up but with the same general trends as the previous datasets.

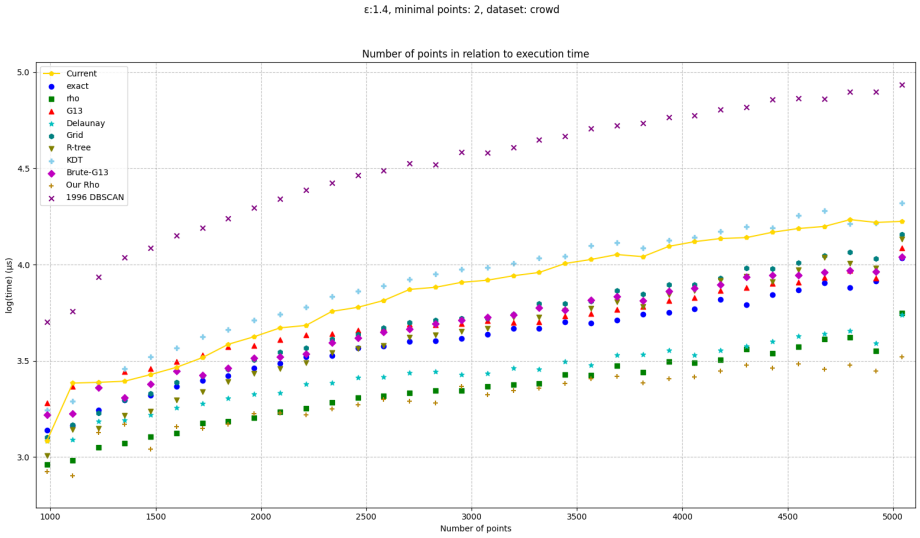


Figure 6.6: *Crowd* dataset.

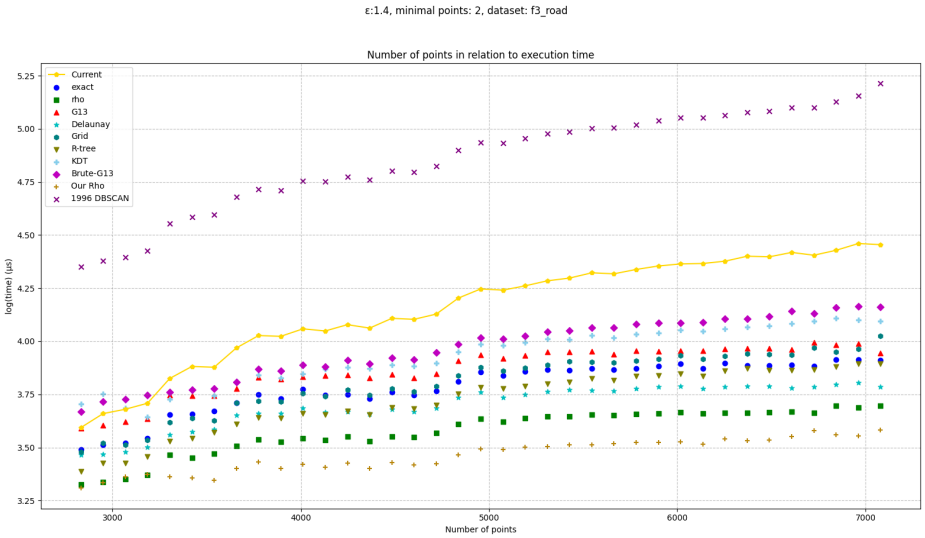


Figure 6.7: *F3* dataset.

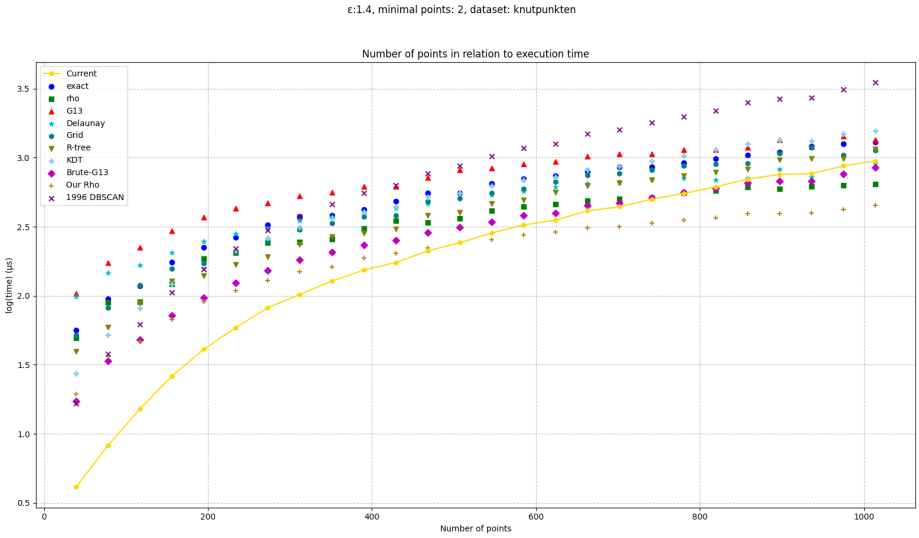


Figure 6.8: *Knutpunkten* dataset.

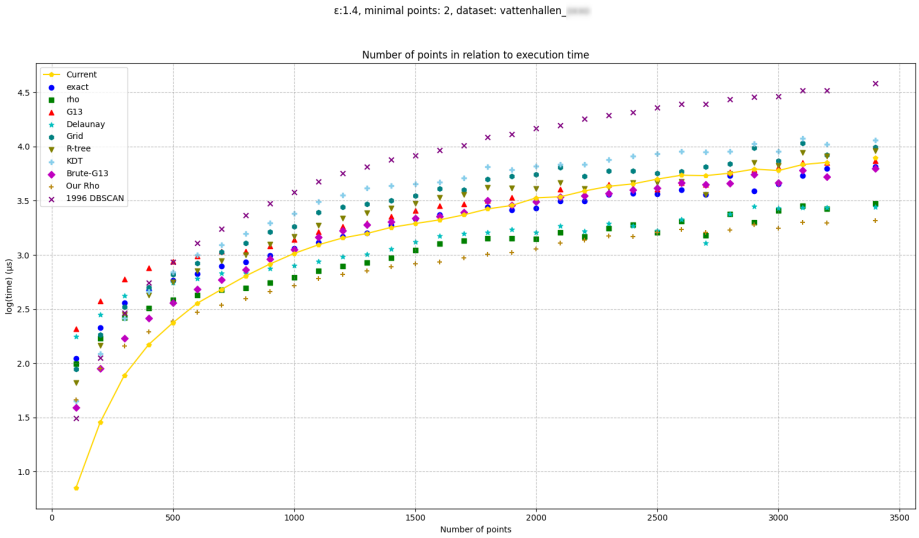


Figure 6.9: *Vattenhallen* dataset.

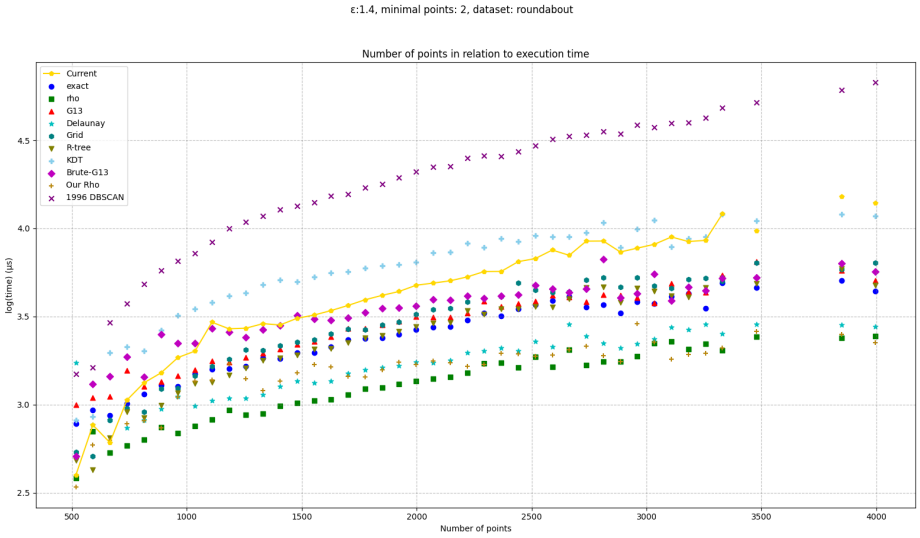


Figure 6.10: *Roundabout* dataset.

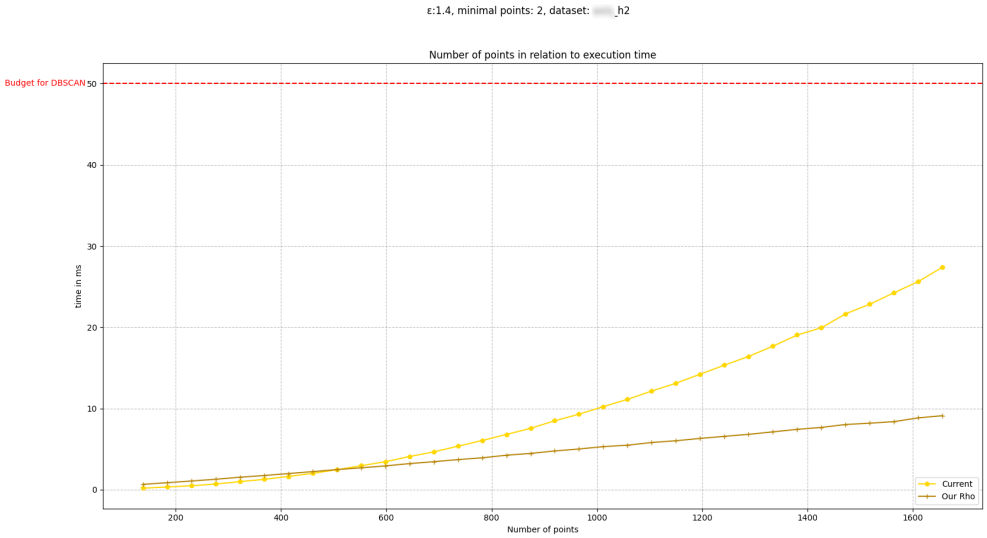


Figure 6.11: *H2* radar result.

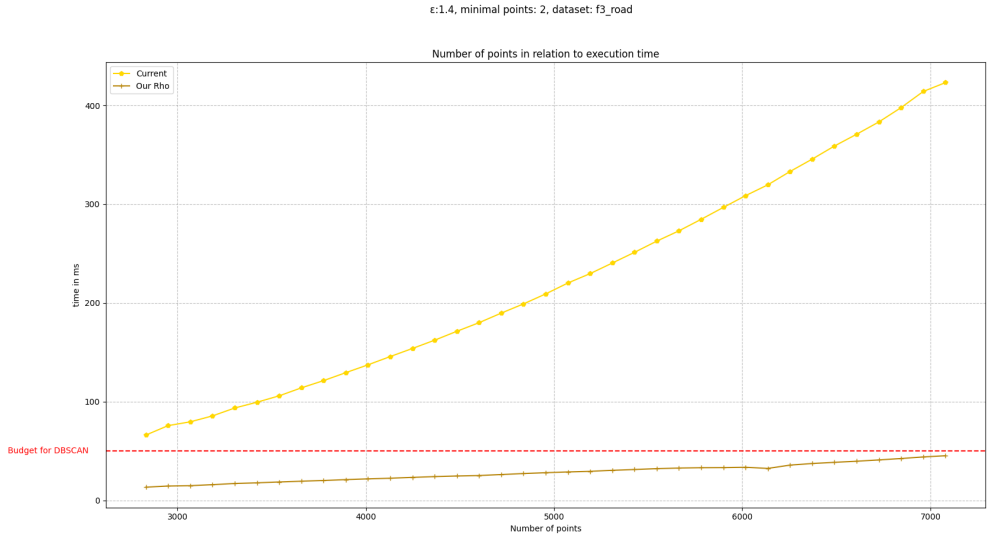


Figure 6.12: *F3* radar result.

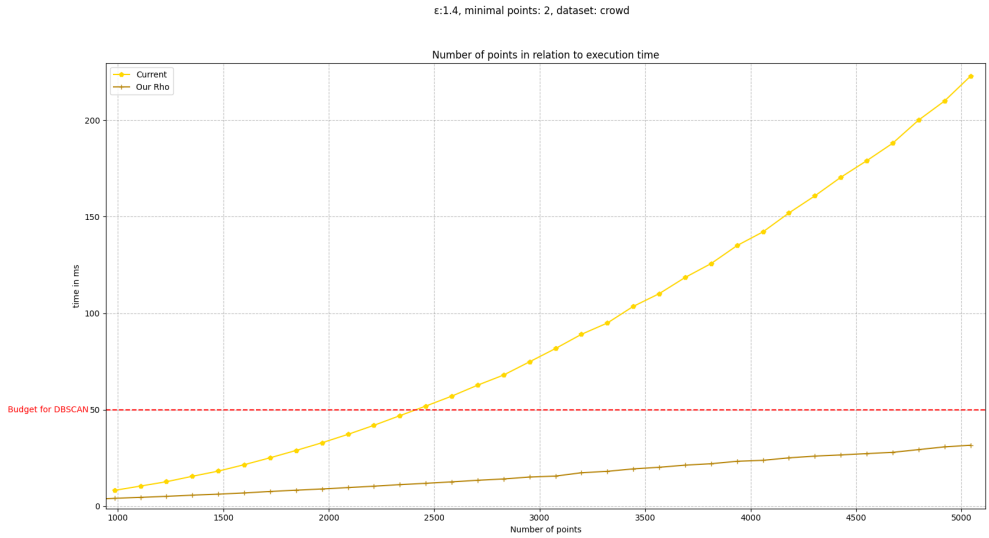


Figure 6.13: *Crowd* radar result.

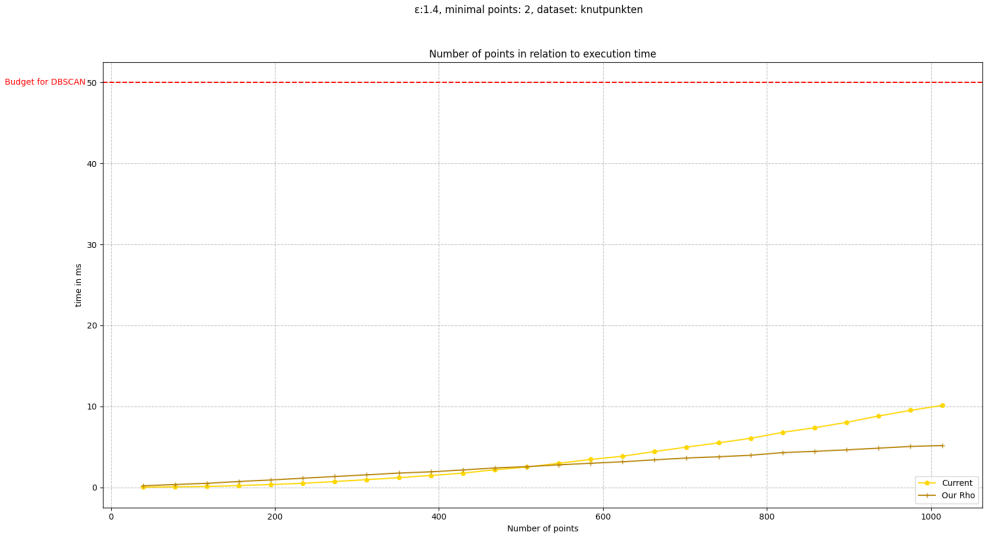


Figure 6.14: *Knutpunkten* radar result.

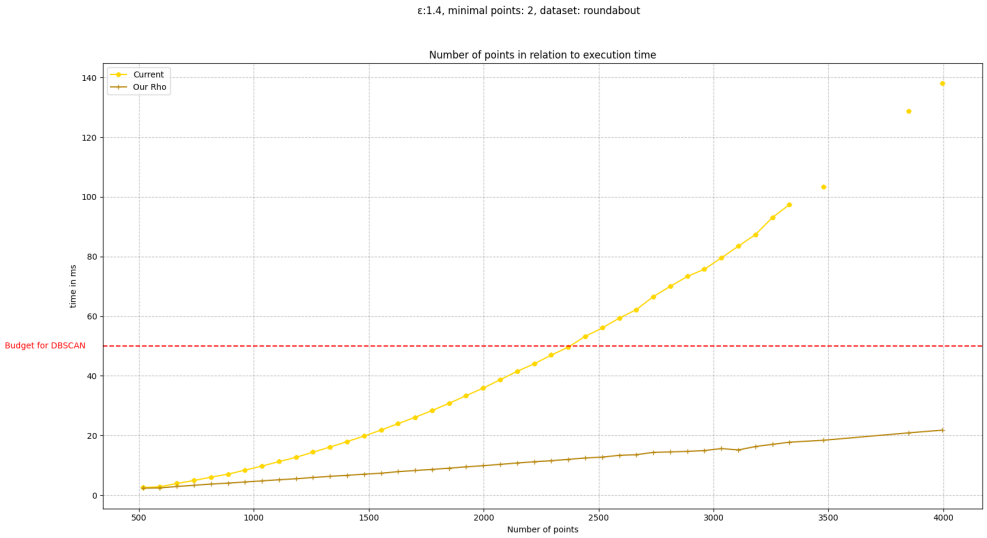


Figure 6.15: *Roundabout* radar result.

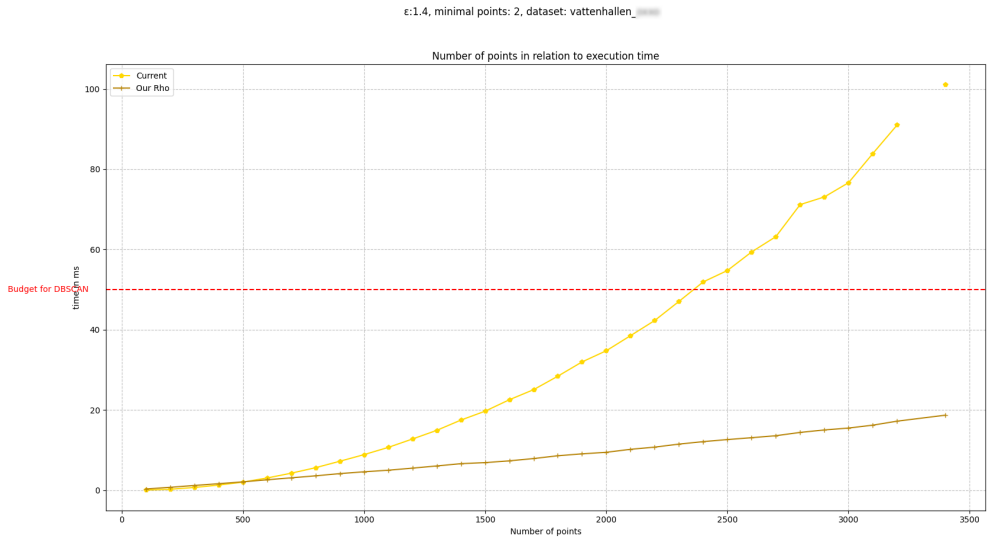


Figure 6.16: *Vattenhallen* radar result.

ρ-approximate DBSCAN					Original DBSCAN			
Total time	Max	Avg time	Triggers	(Milliseconds)	Total time	Max	Avg time	Triggers
23311.9	29.75	10.55	2210	Clustering	106823	82.35	46.1	2318
1737.8	2.85	1	1710	Association	2558.1	3.3	1.1	2317
757.8	8	0	102299	Tracking Prediction	1069.1	1.1	0	147167
5097.6	4.15	0.05	71500	Track Update	7314.5	6	0.05	102792
159279	44.85	6.25	25571	Classification	223184	32.7	5.9	37841

Table 6.1: Comparison between the radar probe results of ρ-approximate DBSCAN (left) and standard DBSCAN (right).

6.3 Multidimensional scaling

According to Figure 6.17, it can be seen the KDT-DBSCAN is not as fast as the other algorithms.

6.4 Probe measurements and flame graph on radar

The flame graph and probe report show the same instance of the *F3* dataset being run on the radar with two different firmware images. The first image uses the so-called "Current-DBSCAN". The second uses the ρ-approximate DBSCAN. No downsampling or other data reduction strategies were used to produce these measurements.

In Table 6.1, we see the ρ-approximate results to the left, the middle is the signal processing chain steps, and on the right are the results of the regular DBSCAN. Figure 6.18 shows the time spent in the call stack for each implementation.

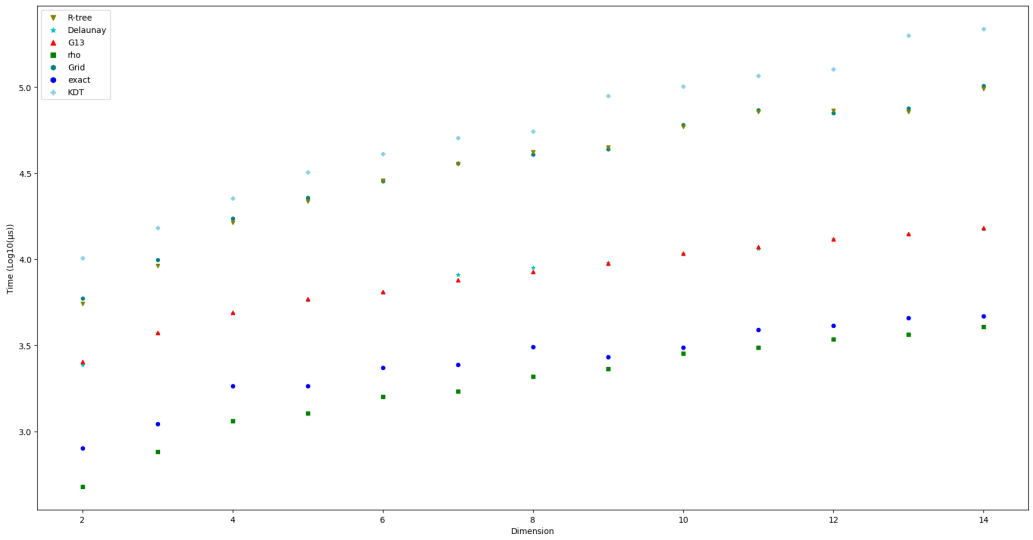


Figure 6.17: The average run time for algorithms for the increasing dimensions and number of points.

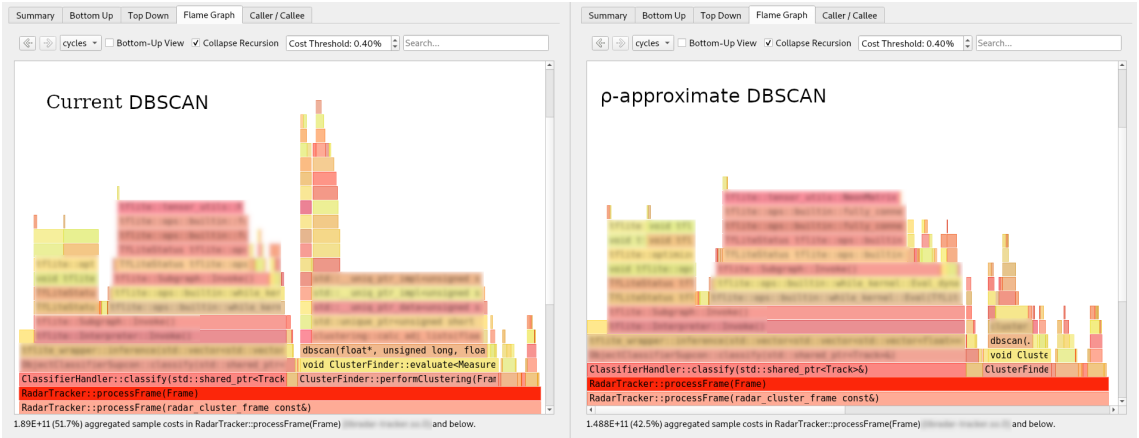


Figure 6.18: Hotspot comparison between the current implementation of DBSCAN (left) and ρ-approximate DBSCAN (right) on the radar.

Chapter 7
Discussion

This chapter covers the discussion of the results provided in the previous chapter.

7.1 Synthetic Data

7.1.1 Direct

According to the synthetic data in Figure 6.1, we can see that Current-DBSCAN outperforms most of the algorithms for datasets from *spiral* to *compound*. What is essential to observe in this graph is that the performance depends heavily on the data clustering properties, such as scattering and noise, not just the number of points. Brute-G13 is an example of this generally performing better than G13 Voronoi counterpart; however, in *birch*, it performs much worse because there are a lot more local points, leading to the $O(n^2)$ merge taking much longer than with Voronoi or Delaunay.

Since our ρ-approximate utilizes brute force calculations when deemed faster and lazily builds the subcells and neighbor lists, it performs much faster than even the original authors' version of ρ-approximate. In order to be sure, however, we would need to use the same compiler version.

The current optimizations of the 1996 DBSCAN have improved it significantly, The Current-DBSCAN implementation works well once the number of points increases too much, causing it to fall behind other lower-time complexity versions. Figure 6.1 shows that it is still the most viable DBSCAN version for datasets with fewer than 800 points on the PC.

7.1.2 The radar system

According to Figures 6.4a, 6.4b, 6.4c and 6.4d, it can be observed that our ρ-approximate DBSCAN already outperforms Current-DBSCAN at *aggregation* which contains 788 points

and after that the Current-DBSCAN processing time increases dramatically while our implementation passes the threshold at 15000 points. The last dataset *birch* finished within a second using our ρ-approximate, while it did not finish with Current-DBSCAN.

An observation is that the threshold number of points where ρ-approximate became faster is lower on the radar. This will be discussed further in 7.3.

7.2 Real-world data

7.2.1 Direct

Looking at the benchmark Figures 6.5, 6.6, 6.7, 6.8, 6.9 and 6.10, it is evident that even with different recordings in different context, the "Current" line intersects with our ρ-approximate implementation at around 500-550 points, then the other ρ-approximate and Delaunay at around 800 to 850 points. After 800-850 points, it can be seen in the *Vattenhallen* dataset that Current-DBSCAN gets outperformed by all ρ-approximate (both ours and Gan-Tao) and Delaunay. In addition, around 1800 points is the intersection point where the Brute-G13 and exact implementations outperform Current-DBSCAN, and gradually the rest get closer to Current-DBSCAN's execution time. Furthermore, according to the *F3* and *Roundabout* datasets, KDT surpassed Current-DBSCAN between 3000 and 4000 points, but that does not always seem to be the case as it did not surpass *Crowd* at these points. This can be because the points are focused in one area, so KDT-DBSCAN skips too few (if any) subtrees.

In Figure 6.5 and 6.10, our version of ρ-approximate did not perform as fast on average as the other version. There are many explanations as to why this is. That version might have optimizations that work well for this kind of data. The tests were only run eleven times and might have a bit of variance. Lastly, depending on other tasks run or interrupts on the computer, that might have impacted our ρ-approximate unfavorably in this particular test. This is discussed in Section 7.8. Nonetheless, the most likely scenario is that their ρ-approximate is written differently and more appropriate to specific scenarios that were found in *Roundabout* and *H2*.

Another observation is between Delaunay and G13. Even though both are using Voronoi for *RangeQuery*, Delaunay seems to be doing this much faster, so much in fact that it is often getting similar results to the $O(n)$ algorithms.

Brute-G13 results depended mainly on which recording we ran. It did very well in the *Knutpunkten* dataset in Figure 6.8, performing better than all other G13 variants. The results were, on the contrary, in *F3*, Figure 6.7 where it performed the worst along with the other $O(n^2)$ algorithms.

7.2.2 Performance depending on ϵ

As seen in Figure 6.2, for the majority of data points, ϵ does not appear to have a significant impact; however, we suggest two findings:

Brute-G13 performs much faster when ϵ is low enough. The performance for Brute-G13 gets evened out with the other algorithms at $\epsilon \approx 200$ and after. As the cell sizes depend on ϵ, it is theorized that the Brute-G13 performance depends significantly on the size of ϵ. If ϵ is too small, the chance of directly identifying cores by only counting the number of points of

one cell decreases; this would enforce counting the points in the 21 neighboring cells almost every time. This would essentially increase the need for core merging. When the ϵ value is at least big enough, all of its points will be counted as core points. As a result, the time complexity will be $O(n^2)$.

We do not know why our Brute-G13 performs so much better than the other G13 and ρ-approximate variants when ϵ is low, except that brute-forcing is favorable for that dataset.

7.2.3 Performance depending on *minPts*

For algorithms in Figure 6.3, most have a varied execution time except *Current-DBSCAN* and *1996 DBSCAN*, which both are more stable. For our implementation of G13, we can also see an apparent increase in performance the higher the variable ***minPts*** is. This is because fewer points are considered core, and more are considered border points. Since determining border points is an $O(n)$ operation, a higher proportion of time being spent in that section increases performance. We have no explanation why our ρ-approximate performs better than the other algorithms with low amount of ***minPts*** besides that we have less overhead.

7.2.4 The radar system

All the results from the radar showed the same trends. Current-DBSCAN is faster for a few points, but at around 500 points, ρ-approximate performs better. After that, the time for ρ-approximate increases linearly while Current-DBSCAN's time increases quadratically, even with the biggest dataset that could be found, *F3*, ρ-approximate still kept under budget.

Since the bottleneck occurs whenever more data is introduced, we can keep to the frame budget during small data loads, changing from the Current-DBSCAN to our implementation of ρ-approximate DBSCAN. We might lose some minimal time instead of using Current-DBSCAN for small data loads, but this lowers the complexity and maintenance of the code compared to having an algorithm-switching mechanism.

7.3 Difference between radar and PC

One reflection was that the algorithms performed differently between ARM and x86. Generally, x86 favored Current-DBSCAN, and ARM favored our ρ-approximate. We are still unsure why this is the case. One theory is that a bigger cache favors the quadratic algorithms for longer. Another theory is that x86 has out-of-order execution, while the ARM processor has dual in-order execution. Both theories mean more time gets spent computing than fetching data on x86, while ARM avoids doing needless computations that take more time on that architecture.

However, this slight difference could come from other factors, such as how we measured the results.

7.4 KDT

The KDT-DBSCAN performance steadily increases execution time compared to increased ϵ values. This increase in execution time is due to the dependency of ϵ. KDT-DBSCAN uses ϵ to resolve *minKey* and *maxKey*, and these two keys are used to decide if sub-trees should be skipped. When ϵ increases, the area between the two keys increases, which increases the chance of points landing within them, resulting in a lower probability of skipping subtrees. Ultimately, this would continuously increase until ϵ is large enough to reach all the points and the time complexity becomes $O(n^2)$.

7.5 Increasing number of dimensions

Scaling clustering algorithms into higher dimensions is likely more costly because it adds one or more parameter(s) to the computationally expensive (when run on every point) distance calculation.

Instead of calculating $\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2} \leq \epsilon$, it becomes more generalized;

$$\sqrt{\sum_{i=1}^{dim}(q_i - p_i)^2} \leq \epsilon$$

where p and q are the two points, dim is the number of dimensions and p_i and q_i represent the two points' coordinates in dimension i.

In the 1996 DBSCAN with constant dim = 2 dimensions, the algorithm runs in $O(2n^2)$ (normalized to $O(n^2)$). However, given an unknown number of dimensions (not constant), the distance calculation becomes more computationally expensive. The clustering is $O(n)$. At the same time, the nested *RangeQuery* is $O(n)$, and now the distance calculation within *RangeQuery* becomes $O(d)$ resulting in a total of $O(dn^2)$ time complexity.

Fortunately, as seen in Figure 6.17, the G13, ρ-approximate, and KDT implementations of DBSCAN are more applicable for scaling. Notably, the number of points increases in higher dimensions to keep a constant $k = 9$ clusters with similar clustering, but this also means that ϵ gradually increases.

KDT, Grid and R-Tree

It is clear that these three implementations have similar trends when the number of dimensions is increased, and the reason why is that they all have the same time complexities. This means the overhead is one of the few factors that make a difference. Another factor is what makes KDT-DBSCAN different. As shown in Figure 6.2 (and explained earlier), KDT-DBSCAN depends on the value of ϵ. This dependence can be seen again in Figure 6.17 because ϵ jumps by 10000 at nine dimensions (from 30000 to 40000), resulting in a jump for KDT-DBSCAN and then ϵ increases again at 14 dimensions (from 40000 to 50000) and KDT-DBSCAN's execution time goes up again.

The KDT-DBSCAN has overhead that can be lowered, which would close the gap to the other implementations. The other implementations were worked on for a long time and had low overhead. Our KDT-DBSCAN implementation was not worked on for that long and had room for improvement.

Figure	Dataset(s)	$\lvert p_1 \rvert$	$\lvert p_2 \rvert$	$\Delta\lvert p \rvert$	Δdim	ΔT(µs)	ΔT(%)
6.6	*Crowd*	2000	2600	600	0	2826	+52
6.10	*Roundabout*	2000	2600	600	0	3900	+27
6.9	*Vattenhallen*	2000	2600	600	0	2894	+45
6.17	$dim2, dim3$	2025	2701	676	1	5087	+50

Table 7.1: KDT execution time change.

G13, Delaunay, exact and ρ-approximate

These implementations have better time complexities in increased dimensions and less overhead. ρ-approximate outperforms the rest regarding increasing dimensions, too.

7.6 Number of points vs dimensions

One issue to investigate is to verify the correlation between execution time and the number of dimensions in Figure 6.17. As mentioned in Section 4.2.3, the number of points increases when the number of dimensions is increased[1]. An increasing number of points affects execution time, which results in an unreliable correlation for such a graph. Fortunately, it renders helpful as the number of points in $dim2$ and $dim3$ is known. These numbers and some previous benchmarks can be used to calculate $\Delta|p|$ and ΔT. $\Delta|p|$ is the change in number of points where $\Delta|p| = |p_2| - |p_1|$ and ΔT is the change in execution time where $\Delta T = T_2 - T_1$. Additionally, the benchmark $|p_i|$ values need to be estimated by taking an average of all points (at any frame) within ± 50 from said $|p_i|$. Table 7.1 shows these calculations.

Even if they use separate datasets (which would change their execution times), the percentile increase of execution time should not differ greatly. In this case, three examples have $45 - 52\%$, and the *Roundabout* example seems different. It is important to say that real-world data is being compared to synthetic datasets ($dim2$ and $dim3$), so the comparisons would not be accurate. However, if an increase in dimensions would affect KDT-DBSCAN dramatically, it would be evident in this observation. Further investigation is required to validate this observation.

7.7 Flame graph comparison with ρ-approximate DBSCAN and standard DBSCAN

In Figure 6.18, the relative time between classification and clustering is much lower in ρ-approximate DBSCAN. This means that clustering takes less time than before or that classification takes more time. The reason could also be both. If bigger clusters are processed much faster, other parts of the signal-processing chain can introduce bottlenecks.

The probe result in Table 6.1 confirms that clustering takes lower overall time, with a total speed improvement of 458%. The maximum time and average time decrease substantially when clustering, but the maximum time increases substantially for ρ-approximate in classification and tracking prediction. The number of triggers varies significantly in the two tests

[1]Increasing number of points is needed to keep the number of clusters the same and the clustering constant.

because the number of frames dropped is fewer on our version. When frames are dropped and enough time has passed, tracking will have to restart, leading to many more triggers and time being spent on the other steps besides clustering. This means that time is saved not only in the clustering part of the chain but propagates to other parts that do not need to do as much computationally heavy work because the clusters are kept. Another explanation is that there is a change in the functionality of DBSCAN and that it generates other clusters compared to the original; however, this is not the case. When the probes are run on a PC, where frames are not dropped, it shows the same amount of triggers, max, and averages between the two implementations. In other words, only the clustering time is changed when run on a PC.

7.8 Shortcomings when measuring results

The tests executed on the PC were run eleven times, creating a higher variance in the statistics. We decided not to increase the number of times executed because of time constraints. These tests were often run in parallel with other tests, and during these tests, the computer was used for other purposes, such as programming, compiling web browsing, or other resource-demanding activities.

On the other hand, the radar tests were run fifty-one times without other processes using resources. Consequently, the variance should be much lower in these tests. However, no further analysis or verification was made.

The flame graph and probe reports were done by simulating the *F3* recording on the radar and attaching perf to the process. The radar was flashed with a firmware image containing a debug version, making probe reports available. The flags for compiling the image to the radar use "-O2" instead of "-O3", so there is a possibility that the performance may vary between these optimization levels. Attaching perf is not an ideal solution since we can not guarantee that both versions run the same amount of frames. We stop the recording after five minutes. Even if the time is identical, triggers could still vary, as there are other considerations when running the signal processing chain on the radar, which makes it hard to carry out entirely accurate tests.

There also seems to be a discrepancy regarding our conclusion about the regular DBSCAN used compared to the internal documents discussed in 2.6. The conclusion drawn from that document was that a maximum of 400 points are allowed before throttling begins, but in all the cases tested, the regular DBSCAN can achieve over 2000 points before it comes close to the 50 ms deadline. This was also the case with "-O2" optimizations. There are multiple reasons why these findings vary, ranging from some assumptions about the time scaling between a PC and the radar to the fact that the tests could have been carried out without any optimizations when compiling. After these measurements, there have also been a few additional changes to the algorithms making it harder to compare. Overall, the internal documents' conclusions are a pointer of what to expect, but they are not necessarily crucial to the independent findings from our tests.

Chapter 8
Conclusion

8.1 ρ-approximate DBSCAN vs regular DBSCAN on radar

This paper has found that ρ-approximate DBSCAN performs better on input over five hundred points and that the current implementation of DBSCAN performs slightly better under five hundred points. Therefore, we suggest changing the current implementation on the radar to our algorithm.

8.2 Research questions

RQ1. *Are there any possibilities to improve the time complexity of the current algorithms?*

Yes, we found an algorithm that we could significantly improve the time complexity.

RQ2. *What are the advantages and disadvantages of the various implementations of the algorithms?*

Current-DBSCAN provides low overhead but high time complexity which works very well for a relatively low number of points. As a result, it was observed that the investigated implementations eventually surpassed Current-DBSCAN in terms of execution time. Brute-G13 DBSCAN, compared to Current-DBSCAN, has overhead, leading to higher execution time for a lower number of points, and the performance varies depending on the cluster formation. Our ρ-approximate DBSCAN also has some overhead, but it surpasses the Brute-G13 DBSCAN in execution speed. Finally, the current KDT-DBSCAN has a high overhead, but it can be observed that it has better time complexity than Current-DBSCAN as it eventually surpasses it. One advantage of

KDT-DBSCAN is that the implementation can run on arbitrary dimensions, requiring further optimization.

Chapter 9
Further Work

Over recent years, there has been tremendous work on improving DBSCAN. While G13-DBSCAN and ρ-approximate DBSCAN already improve upon the original DBSCAN, other possibilities exist. This chapter covers some of those possibilities.

9.1 ρ-approximate DBSCAN

This section discusses future improvements to our ρ-approximate DBSCAN.

9.1.1 Create own hash table

The parts of our program taking the most time are the inserts and the lookup for our current hash table. The hash table currently in use is Boost's unordered_map which we have observed is slightly faster than the standard library implementation of unordered_map (in C++). Since all the cells could be made to have a constant size, and each cell has its bucket, it is possible to create a vector-based grid making lookup and insert as fast as can be.

9.1.2 Try different ρ parameters for our ρ-approximate implementation

Outlining the performance impact of the variable ρ is vital for future use in the radar. We know that a higher ρ increases the speed of the *RangeQuery*, while a lower ρ decreases the speed but increases the accuracy.

Being able to fine-tune this parameter depending on the input points to stay inside the frame deadline budget is essential in regulating so as much time as possible is allotted to each frame without trespassing onto another frame's time.

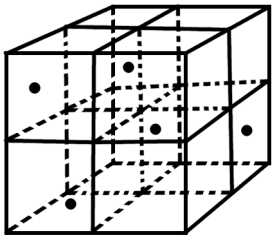


Figure 9.1: Example of a 3d grid.

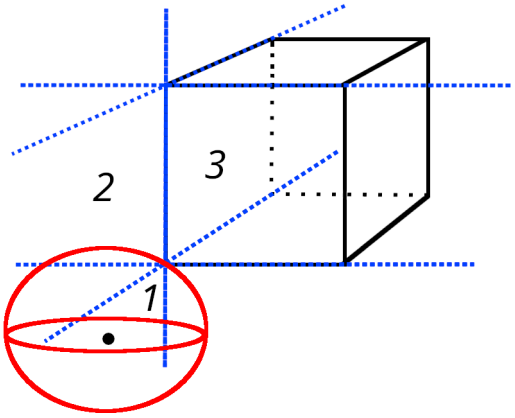


Figure 9.2: A point outside a cube which radius creates a sphere.

9.1.3 3D

In order to extend our current implementation and optimize it for 3D, some changes are needed. Every cell becomes a cube where the space diagonal now becomes ϵ size, meaning each side is $\frac{\epsilon}{\sqrt{3}}$. Additionally, the hash table used need to accommodate a grid $\mathcal{G}(i, j, k)$ and produce a hash function that works for a grid of cubes. This new grid is demonstrated in Figure 9.1, where we can see five points inside a grid of eight cubes.

Lastly, it will be different when calculating "disjoint", "intersect", and "fully covered" functions. In Figure 9.2, we can see the same three possibilities presented in the 2D case. A point could be close to a corner of the cube, it could also be close to the plane created by the side of the cube, and lastly, it could be inside a cube. Here the only change is the second case. In Figure 9.3, we see little difference between getting the shortest distance from a plane and a line. It is still the case that the orthogonal vector to the plane in relation to the point creates the shortest path. To create the "disjoint", "intersect", and "fully covered" functions, the same logic is applied as in the 2D case.

9.2 More on KDT-DBSCAN

Our implementation for KDT-DBSCAN is fairly optimized, but there is still room for improvement. Four kinds of optimizations may be done so that KDT-DBSCAN becomes more optimized.

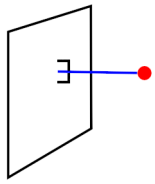


Figure 9.3: A red point close to a black plane, with a blue vector that is orthogonal to the plane.

9.2.1 *BuildTree*

Firstly, building the tree entails a high amount of overhead. The bottleneck in *BuildTree* calculates the median by sorting lists on every level. This can be optimized by storing the sorting states for each dimension so that the sorting is done only k amount of times where k is the number of dimensions. This is complex because it will make the splitting process more tedious as splitting will be performed on all states instead of just one list. There is also a proposed optimization where MapReduce is used for producing a balanced tree [10].

The latter improvement that can be done on KDT-DBSCAN is the following. Our implementation for KDT-DBSCAN was focused on improving *RangeQuery* while the rest of the DBSCAN implementation stayed identical to the 1996 DBSCAN. This may be the issue in the implementation. Adapting KDT-DBSCAN to Current-DBSCAN may be the next step to improving the execution time of the clustering algorithm. Additionally, it may be possible to apply KDT-DBSCAN in a more G13 way and divide the KDT-DBSCAN clustering process into phases.

Quickselect

One possibility is to improve the search for median points by using *Quickselect* [35]. *Quickselect* is an algorithm used to select the median and split the list in the middle, where the values on the left are guaranteed to be less than the median and vice versa. *Quickselect* works better than *Quicksort* in this situation as it has $O(n)$ instead of *Quicksort*'s $O(n \log(n))$, and it is not necessary to sort the list on each level completely.

Parallelization

There is a possibility to take advantage of parallelization in building the tree for KDT-DBSCAN. Suppose the mentioned optimization to store the sorting state is used. In that case, it becomes possible to run the process of splitting the different sorting lists in parallel, as they are not all dependent on each other. In this case, the performance will be improved for dimensions higher than two, but it may remain the same for two dimensions. This remains to be tested.

9.2.2 *RangeQuery*

There are few ideas to improve KDT-DBSCAN's range query, but there is one inspired by G13 where two other ***min/max keys*** are introduced that guarantee the point is within ϵ. In

this case, these two ***keys*** are only calculated when the point is within the original ***keys***, but these ***keys*** will have the coordinates added or subtracted[1]with $\frac{\epsilon}{\sqrt{2}}$ instead (same as G13 cells). This will guarantee that the point is within ϵ without distance calculation. This is expected to improve the algorithm for high-density clusters and reduce its dependency on ϵ as the number of distance calculations would be reduced.

9.2.3 More tests

The benchmarks that were performed on KDT-DBSCAN were inconclusive as more tests are required to be done. One observation that needs to be done is comparing KDT-DBSCAN in 2 dimensions with it being used in 3 dimensions, and the only variating variable would be the number of points.

9.3 GriT-DBSCAN

GriT-DBSCAN is another optimization for DBSCAN that is a grid-based exact DBSCAN in Euclidean space. It builds on the findings from Gan and Tao and was released this year. The algorithm is divided into two main parts. The first part involves organizing Grid Tree; secondly, a technique is used to prune unnecessary distance calculations [22].

Some of the first steps in GriT-DBSCAN involve building the Grid Tree, which only contains non-empty grids. Afterward, Grid Query can be performed, equivalent to *Range-Query* but is performed on grid level and optimized by determining core points by stopping query once ***minPts*** is reached. Other stages of GriT-DBSCAN involve merging, which is optimized with a method called Fast-Merging, and another optimization technique called pruning which eliminates unnecessary distance calculations.

In total, GriT-DBSCAN's time complexity is complicated as it involves multiple stages and optimizations. However, according to the Article [22], it has a time complexity of $O((2\lceil\sqrt{d}\rceil)^{d+2}\kappa n + d\eta)$. The time complexity is given where n is the number of points, d is the number of dimensions, κ is the maximum number of iterations in the merging step, and η is the maximum interval number in the feature space. Given these variables to be constant, the algorithm's time complexity becomes $O(n)$. According to their results, this algorithm performs much better than ρ-approximate DBSCAN.

[1]Depending on if it is ***max*** or ***min***.

References

[1] Gene M Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 483–485, 1967.

[2] Guilherme Andrade, Gabriel Ramos, Daniel Madeira, Rafael Sachetto, Renato Ferreira, and Leonardo Rocha. G-dbscan: A gpu accelerated algorithm for density-based clustering. *Procedia Computer Science*, 18:369–378, 2013.

[3] Mihael Ankerst, Markus M Breunig, Hans-Peter Kriegel, and Jörg Sander. Optics: Ordering points to identify the clustering structure. *ACM Sigmod record*, 28(2):49–60, 1999.

[4] Arthur O Bauer. Christian hülsmeyer and about the early days of radar inventions. *Self-published https://aobauer. home. xs4all. nl/Huelspart1def. pdf*, 2005.

[5] Ahmad Biniaz, Prosenjit Bose, David Eppstein, Anil Maheshwari, Pat Morin, and Michiel Smid. Spanning trees in multipartite geometric graphs. *Algorithmica*, 80:3177–3191, 2018.

[6] Christian Böhm, Karin Kailing, Peer Kröger, and Arthur Zimek. Computing clusters of correlation connected objects. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 455–466, 2004.

[7] Prosenjit Bose, Anil Maheshwari, Pat Morin, Jason Morrison, Michiel Smid, and Jan Vahrenhold. Space-efficient geometric divide-and-conquer algorithms. *Computational Geometry*, 37(3):209–227, 2007.

[8] E Oran Brigham. *The Fast Fourier Transform and its applications*. Prentice-Hall, 1988.

[9] Louis Brown. A radar history of world war ii. *J. Am. Hist. Res*, 1999.

[10] Russell A Brown. Building a balanced kd tree with mapreduce. *arXiv preprint arXiv:1512.06389*, 2015.

[11] Guénaël Cabanes and Younès Bennani. *Learning the Number of Clusters in Self Organizing Map*. 04 2010.

[12] Hong Chang and Dit-Yan Yeung. Robust path-based spectral clustering. *Pattern Recognition*, 41(1):191–203, 2008.

[13] Vineet Chaoji, Mohammad Al Hasan, Saeed Salem, and Mohammed J Zaki. Sparcl: Efficient and effective shape-based clustering. In *2008 Eighth IEEE International Conference on Data Mining*, pages 93–102. IEEE, 2008.

[14] James W Cooley and John W Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of computation*, 19(90):297–301, 1965.

[15] Pierre Duhamel and Martin Vetterli. Fast fourier transforms: a tutorial review and a state of the art. *Signal processing*, 19(4):259–299, 1990.

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

[17] Junhao Gan and Yufei Tao. Dbscan revisited: Mis-claim, un-fixability, and approximation. In *Proceedings of the 2015 ACM SIGMOD international conference on management of data*, pages 519–530, 2015.

[18] Junhao Gan and Yufei Tao. On the hardness and approximation of euclidean dbscan. *ACM Transactions on Database Systems (TODS)*, 42(3):1–45, 2017.

[19] Aristides Gionis, Heikki Mannila, and Panayiotis Tsaparas. Clustering aggregation. *ACM Trans. Knowl. Discov. Data*, 1(1):4–es, mar 2007.

[20] Ade Gunawan and M de Berg. A faster algorithm for dbscan. *Master's thesis*, 2013.

[21] John L Gustafson. Reevaluating amdahl's law. *Communications of the ACM*, 31(5):532–533, 1988.

[22] Xiaogang Huang, Tiefeng Ma, Conan Liu, and Shuangzhe Liu. Grit-dbscan: A spatial clustering algorithm for very large databases. *Pattern Recognition*, page 109658, 2023.

[23] I. Kärkkäinen and P. Fränti. Dynamic local search algorithm for the clustering problem. Technical Report A-2002-6, Department of Computer Science, University of Joensuu, Joensuu, Finland, 2002.

[24] I. Kärkkäinen and P. Fränti. Gradual model generator for single-pass clustering. *Pattern Recognition*, 40(3):784–795, 2007.

[25] George Karypis, Eui-Hong Han, and Vipin Kumar. Chameleon: Hierarchical clustering using dynamic modeling. *computer*, 32(8):68–75, 1999.

[26] Kdab. Kdab/hotspot: The linux perf gui for performance analysis., Aug 2017.

[27] David Kirkpatrick and Stefan Reisch. Upper bounds for sorting integers on random access machines. *Theoretical Computer Science*, 28(3):263–276, 1983.

[28] Matthias Klusch, Stefano Lodi, and Gianluca Moro. Distributed clustering based on sampling local density estimates. In *IJCAI*, volume 3, pages 485–490. Citeseer, 2003.

[29] Peter Lammich. Efficient verified implementation of introsort and pdqsort. In Nicolas Peltier and Viorica Sofronie-Stokkermans, editors, *Automated Reasoning*, pages 307–323, Cham, 2020. Springer International Publishing.

[30] Eric Hsueh-Chan Lu, Vincent S Tseng, and S Yu Philip. Mining cluster-based temporal mobile sequential patterns in location-based service environments. *IEEE transactions on knowledge and data engineering*, 23(6):914–927, 2010.

[31] What Is Data Mining. Data mining: Concepts and techniques. *Morgan Kaufinann*, 10:559–569, 2006.

[32] Rong Nie, Zixuan Yao, Dong Wei, Kaizhang Hou, and Jishuai Lin. Research on kdt-dbscan-based personal semantic location acquisition. In *2020 Chinese Control And Decision Conference (CCDC)*, pages 2577–2581. IEEE, 2020.

[33] Sankar K Pal and Pabitra Mitra. *Pattern recognition algorithms for data mining*. CRC press, 2004.

[34] Sheikh F Qureshi, Hector A Gonzalez, Chen Liu, Bernhard Vogginger, Marco Stolba, Stefan Scholze, Sebastian Höppner, and Christian Mayr. Efficient dbscan implementation in a multi-core dsp for fmcw radars. In *2022 IEEE Radar Conference (RadarConf22)*, pages 1–6. IEEE, 2022.

[35] André Rauh and Gonzalo R. Arce. A fast weighted median algorithm based on quickselect. In *2010 IEEE International Conference on Image Processing*, pages 105–108, 2010.

[36] Mark A Richards. *Fundamentals of Radar Signal Processing*. McGraw-Hill, 2005.

[37] Mark A Richards. *Fundamentals of radar signal processing*. McGraw-Hill Education, 2014.

[38] Boris Schäling. *The boost C++ libraries*. Boris Schäling, 2011.

[39] Gholamhosein Sheikholeslami, Surojit Chatterjee, and Aidong Zhang. Wavecluster: a wavelet-based clustering approach for spatial data in very large databases. *The VLDB Journal*, 8:289–304, 2000.

[40] Merrill I Skolnik. *Introduction to Radar Systems*. McGraw-Hill, 2001.

[41] Josef Steinbaeck, Christian Steger, Gerald Holweg, and Norbert Druml. Design of a low-level radar and time-of-flight sensor fusion framework. 08 2018.

[42] Pang-Ning Tan, Michael Steinbach, and Vipin Kumar. Data mining introduction. *People's Posts and Telecommunications Publishing House, Beijing*, 2006.

[43] Alfred Ultsch. U*matrix: a tool to visualize clusters in high dimensional data. 01 2003.

[44] Vincent M Weaver. Linux perf_event features and overhead. In *The 2nd international workshop on performance analysis of workload optimized systems, FastPath*, volume 13, page 5, 2013.

[45] Florian Wende. C++ data layout abstractions through proxy types. In *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 758–767, 2019.

[46] Sungwon Yoo, Shahzad Ahmed, Kang Sun, Duhyun Hwang, Jungjun Lee, Jungduck Son, and Sung Ho Cho. Radar recorded child vital sign public dataset and deep learning-based age group classification framework for vehicular application. *Sensors*, 21:2412, 03 2021.

[47] C.T. Zahn. Graph-theoretical methods for detecting and describing gestalt clusters. *IEEE Transactions on Computers*, C-20(1):68–86, 1971.

[48] T. Zhang, R. Ramakrishnan, and M. Livny. Birch: A new data clustering algorithm and its applications. *Data Mining and Knowledge Discovery*, 1(2):141–182, 1997.

[49] Changqing Zhou, Dan Frankowski, Pamela Ludford, Shashi Shekhar, and Loren Terveen. Discovering personally meaningful places: An interactive clustering approach. *ACM Transactions on Information Systems (TOIS)*, 25(3):12–es, 2007.

Appendices

Appendix A
Contribution Statement

	Segment	Joakim	Joseph
Writting	Introduction	◔	◕
	Bottlenecks and optimizations	◐	◑
	Clustering algorithms	◐	◑
	Methodology	◔	◕
	Implementation	◐	◑
	Results	◕	◔
	Discussion	◐	◑
	Conclusion	◐	◑
	Further work	◐	◑
	Appendix	◐	◑
Developing	G13	◕	◔
	KD-Tree	○	●
	1996 DBSCAN	○	●
	ρ-approximate DBSCAN	●	○
Other	Measurements	◕	◔
	Figure Creation	◐	◑

Circles showing work contribution in segments.
The proportion of shading reflects the amount of contribution.

Appendix B
Results Details

B.1 Real-world recordings

The recording locations can be viewed in Table B.1

B.2 Raw benchmarks

When first approaching how to visualize the results, our first plots were constructed with time on the y-axis and each frame on the x-axis. These plots could be more readable and needed to be visualized differently. However, one benefit is how they can visualize the load during a recording. In Table B.2, one can see the raw benchmarks for each recording made. For example Figure B.8, one can see many points being recorded in the 2100 first frames; after that, the number of points almost goes to zero. This can indicate that all the walking took place before frame 2100, and after that frame, everyone was inside the house and not getting picked up by the radar.

Location name	Description	Figure
H2	Covering parking lot and some bushes	B.1
Crowd	Crowd of employees at hosting company walking	B.2
F3	A lot of bushes and trees as well as a road	B.3
Knutpunkten	Overseeing a traffic light	B.4
Roundabout	Busy roundabout with cars, buses and trucks	B.5
Vattenhallen	Road near E-huset at LTH	B.6

Table B.1: Description of the different locations.

Figure B.1: Picture of parking lot with bushes.

Figure B.2: People walking in crowd.

Figure B.3: Road with swaying bushes and trees.

Figure B.4: Crowded traffic light with park near by.

Location name	Figure
H2	B.7
Crowd	B.8
F3	B.9
Knutpunkten	B.10
Roundabout	B.11
Vattenhallen	B.12

Table B.2: Table referencing every raw benchmark.

Figure B.5: Busy roundabout.

Figure B.6: *Vattenhallen* near LTH in Lund.

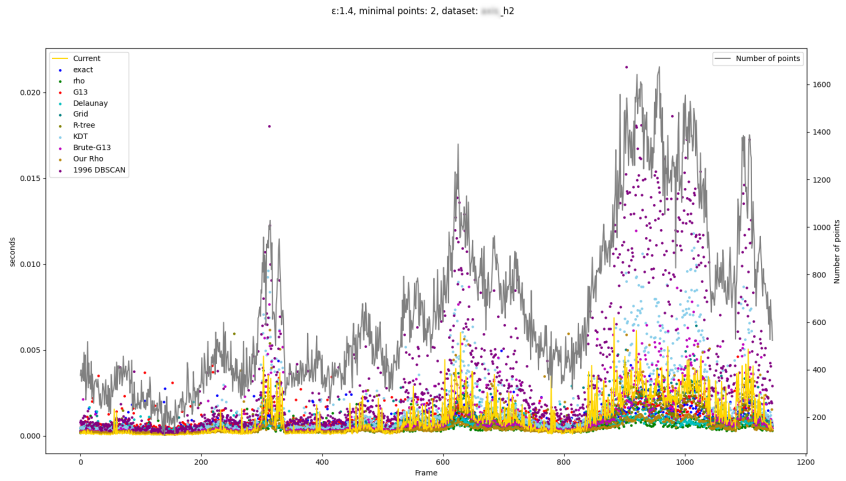


Figure B.7: *H2* raw Benchmark.

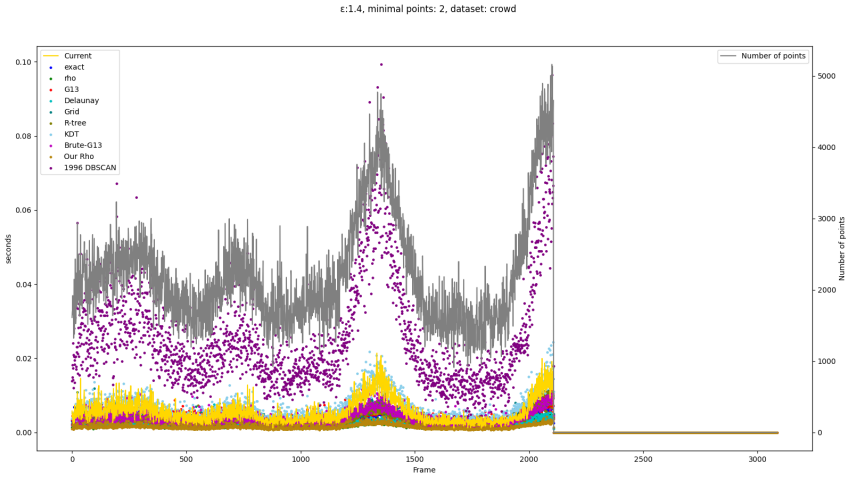


Figure B.8: *Crowd* raw Benchmark.

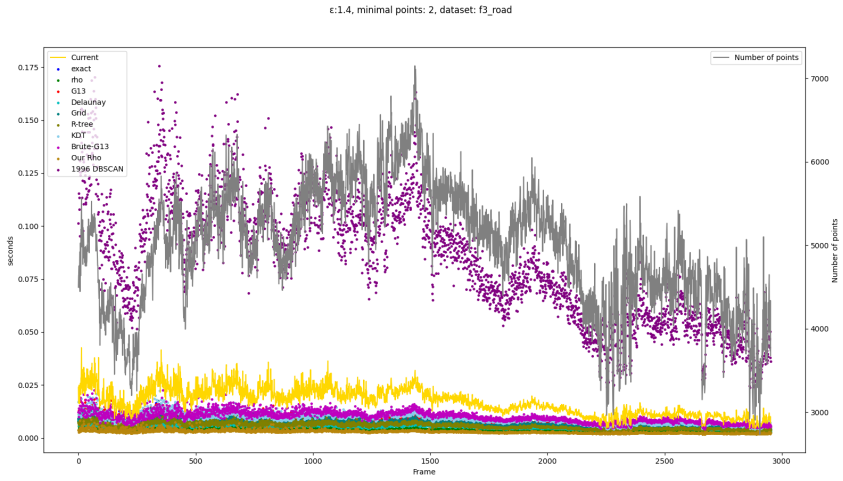


Figure B.9: *F3* raw Benchmark.

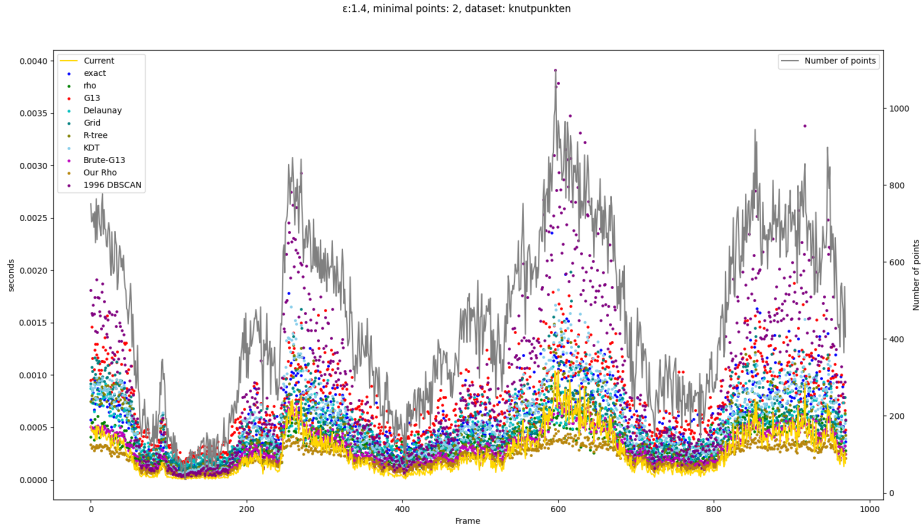


Figure B.10: *Knutpunkten* raw Benchmark.

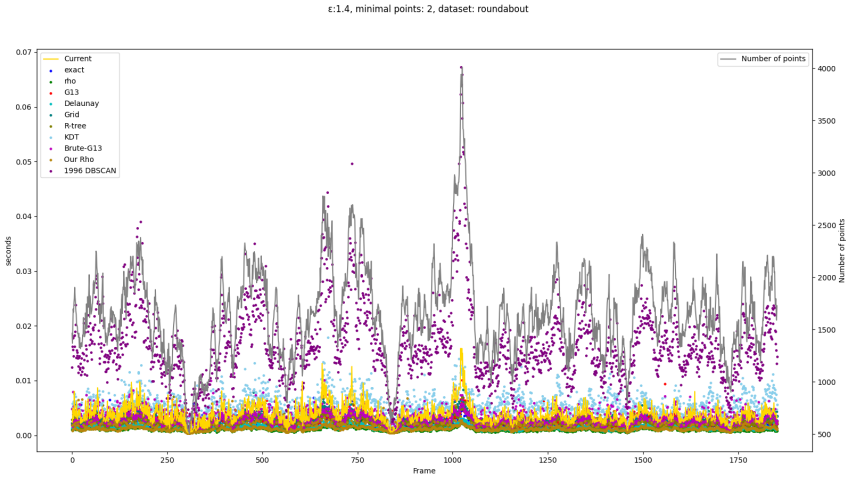


Figure B.11: *Roundabout* raw benchmark.

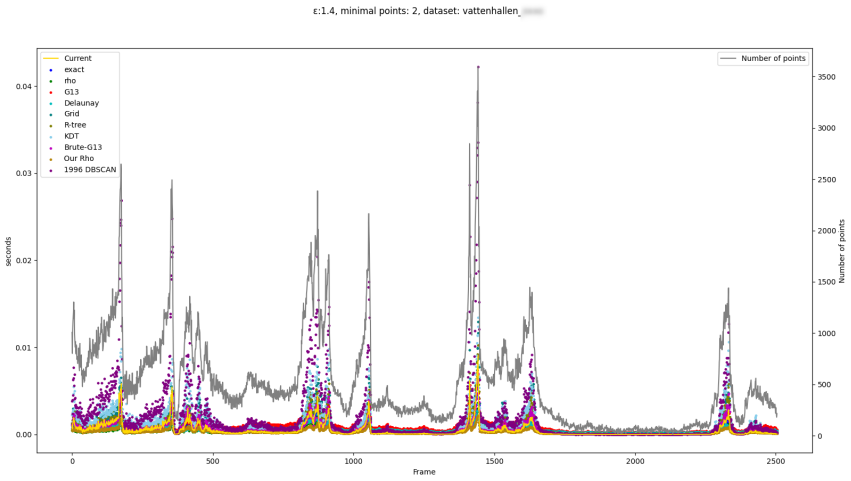


Figure B.12: *Vattenhallen* raw benchmark.

Appendix C
DBSCAN: Other Variations

C.1 Leaf-DBSCAN

This is one of the first variations of DBSCAN that were researched. This implementation uses a grid-based system for DBSCAN and reduces memory usage by reducing the number of heap allocations [34]. This was not further investigated because memory optimization is not the main focus of the thesis since the radar units do not struggle in that aspect.

C.2 G-DBSCAN

One of the DBSCAN implementations that were looked at is G-DBSCAN. The idea behind this implementation is to run the clustering on the GPU [2]. However, this was already investigated by the hosting company, so we did not prioritize investigating this further.

C.3 Multi-Core

Parallelization of programs is one of the classic ways to attempt to improve the performance of a program. Parallelization revolves around using multiple processors or cores to execute the same program in parallel to reduce execution time. Respectively, there are a few factors that need to be taken into account before deciding if a multi-core approach is worth attempting.

First, it may seem obvious, but it is necessary to know if the hardware contains multiple cores (or processors) available for parallelization. If the processing unit contains only one core, the program will likely be slower due to unnecessary context switches.

Secondly, any dependency between the cores for shared resources must be considered when implementing a multi-core approach. If the code contains too many dependencies, it

must be run sequentially. Consequently, if most of the code can not be parallelized, then the performance gain will be limited by Amdahl's law $S_{latency}(s) = \frac{1}{(1-p)+\frac{p}{s}}$ [1, 21].

The question then is if the DBSCAN in the given radar can be parallelized, and the answer is probably not, according to some of our tests with locks and *OpenMP*. The CPU has multiple cores, which means it is possible to run threads in parallel.

We have added a simple lock-based multi-core functionality to an improved implementation of DBSCAN based on our "1996 DBSCAN" and realized that there is an excessive amount of overhead, making the multi-core approach generally slower than the sequential DBSCAN. However, the part of DBSCAN that was parallelized was only the calculation of neighbors, which is the most costly part. In addition, that part is the most independent. It is also possible to parallelize the clustering algorithm itself. On the one hand, parallelizing the cluster identification would improve execution time. On the other hand, the time complexity of this step is not as significant as the neighbor calculation since that part of DBSCAN is only $O(n)$, so the performance would not be improved enough. It remains to be seen if using a lock-free data structure to calculate neighbors would be faster than the sequential algorithm.

INSTITUTIONEN FÖR DATAVETENSKAP | LUNDS TEKNISKA HÖGSKOLA | PRESENTERAD 2023-08-24

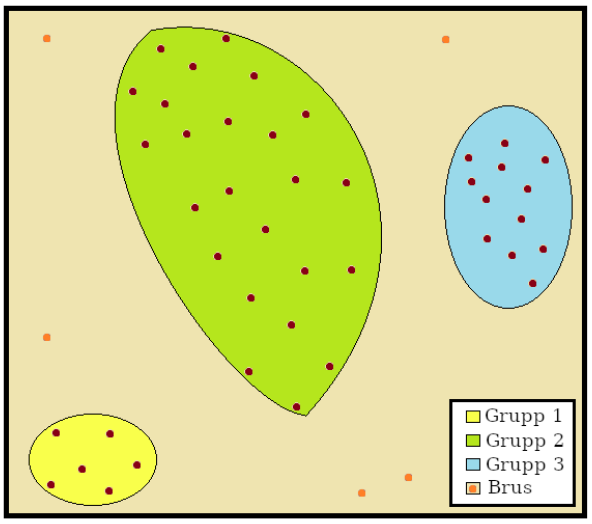
EXAMENSARBETE Investigating Optimization Techniques for Algorithms in Radars
An effort to improve the execution time of the signal processing chain
STUDENTER Joakim Mörling, Joseph Atalla
HANDLEDARE Jonas Skeppstedt (LTH), Aras Papadelis, Henrik Lehtonen, Santhosh Nadig (Axis Communications AB)
EXAMINATOR Per Andersson (LTH)

Ett snabbare radarsystem

POPULÄRVETENSKAPLIG SAMMANFATTNING **Joakim Mörling, Joseph Atalla**

Ett radarsystem behöver kunna hantera stora mängder data varje sekund. Vi har utforskat olika sätt att göra radarsystemet bättre, vilket lett till en ny algoritm. Algrotimen ger en växande tidsvinst när man ökar mängden data.

Ett radarsystem kan användas till att spåra bilar och människor. Den signalen som radarn skickar ut och tar emot kan omvandlas till koordinater där det finns rörelse. Det är dessa koordinater som sedan blir till mätpunkter, som grupperas för att avgöra vad det är som rör sig.



Vid stora mängder punkter är det svårt att hantera alla punkter. Ett sätt att lösa detta problem är att ta bort punkter, istället för att hantera dem. Detta fungerar men har nackdelen att man mister informationen som dessa punkter kan ge. Lycklingsvis finns det en lösning där inga punkter behöver kastas. Denna lösning är att se över algoritmen, för att göra grupperingen snabbare.

Vårt examensarbete har tittat på den senaste forskningen om dessa typer av algoritmer. Vi har även undersökt att använda algoritmen i tre dimensioner istället för de två dimensioner som algoritmen använder just nu.

Olika algoritmer testades och jämfördes på en dator tills en utmanare uppenbarade sig. Den förbättrades efteråt för radarnsystemets specifikationer.

Resultaten visar en stor tidsvinst på radarsystemet när det är mer än 500 mätpunkter. Efter 500 mätpunkter så blir vår algoritm snabbare och snabbare jämfört med en referensalgoritm. Det är först när det är över 10 000 punkter som vår algoritm inte kan hantera punkterna i tid. Slutsatsen blir således att referensalgoritmen bör ersättas med vår algoritm då den är snabbare vid många punkter och runt samma hastighet vid få punkter.