A Single-Pass Grid-Based Algorithm for Clustering Big Data on Spatial Databases

by

Evangelos Taratoris

B.S., Massachusetts Institute of Technology (2014)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Masters of Engineering in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2017

Signature redacted

Author ....
Department of Electrical Engineering and Computer Science
March 8, 2017

Signature redacted

Certified by.....
Samuel R. Madden
Professor
Thesis Supervisor

Signature redacted

Accepted by ....
Christopher J. Terman
Chairman, Masters of Engineering Thesis Committee

A Single-Pass Grid-Based Algorithm for Clustering Big Data on Spatial Databases

by

Evangelos Taratoris

Submitted to the Department of Electrical Engineering and Computer Science
on March 8, 2017, in partial fulfillment of the
requirements for the degree of
Masters of Engineering in Computer Science and Engineering

Abstract

The problem of clustering multi-dimensional data has been well researched in the scientific community. It is a problem with wide scope and applications. With the rapid growth of very large databases, traditional clustering algorithms become inefficient due to insufficient memory capacity. Grid-based algorithms try to solve this problem by dividing the space into cells and then performing clustering on the cells. However these algorithms also become inefficient when even the grid becomes too large to be saved in memory.
This thesis presents a new algorithm, *SingleClus*, that is performing clustering on a 2-dimensional dataset with a *single pass of the dataset.* Moreover, it optimizes the amount of disk I/O operations while making modest use of main memory. Therefore it is theoretically optimal in terms of performance. It modifies and improves on the Hoshen-Kopelman clustering algorithm while dealing with the algorithm's fundamental challenges when operating in a Big Data setting.

Thesis Supervisor: Samuel R. Madden
Title: Professor

Acknowledgments

First, I would like to thank my MEng thesis advisor, Sam Madden, for his continuous guidance and support during my time as an MEng student. I was very fortunate to have a great advisor that gave me enough freedom to explore various topics individually before I decided on my Thesis topic. Moreover, he provided immeasurable support during the last year, while I was in the process of writing my Thesis. The fact that he managed to be so helpful, while at the same time having many other responsibilities, is something for which I will be forever grateful.

I would also like to thank my great mentor, Stavros Papadopoulos. He provided great insights and support throughout my time as an MEng student. Moreover, he was very patient while I was in the process of finding a good Thesis project. He always provided constructive criticism and useful technical knowledge when he felt I was heading to the wrong direction.

I would like to thank Anantha Chandrakasan who has been my academic advisor for 5 years and who gave me my first research opportunity as an undergraduate student at MIT. He was always very helpful and supportive of my choices and I appreciate it greatly.

I would also like to thank Anne Hunter, Linda Sullivan, Vera Sayzew and Jason McKnight for their help during my time as an MEng student.

I would like to thank my close friends that gave me balance during my time at MIT.

Finally I would like to thank my family, and especially my parents Ioannis and Evlampia, for their incredible amount of faith in my abilities and their constant moral support when I needed it most.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Clustering multi-dimensional data is a basic problem in the field of knowledge discovery. The most well-known algorithms in the field are DBscan and K-means ([1],[2]). Many other algorithms have been proposed for discovering clusters.
With the increasing prevalence of Big Data and very large databases, many of these algorithms become inefficient due to insufficient memory. Moreover, any algorithm with non-linear running time becomes prohibitively slow. Hence, there is a need for a clustering algorithm that can run efficiently when clustering vast amounts of data, even when the memory is a bottleneck.

1.1 Motivation

Multi-dimensional databases appear in many fields. Examples include geospatial data, oceanographic data, genomics data and temporal data. ([12]). Often, when observing the distribution of these data in space, regions of higher relative density emerge. These regions are called clusters. Points that belong to the same cluster usually share some common characteristics. Therefore it is of interest to efficiently discover these clusters.

Current clustering algorithms perform clustering in main memory. As the size of databases increases rapidly, memory becomes a bottleneck. Hence, these algorithms don't just become inefficient but largely useless.

A new category of algorithms, called grid-based algorithms ([8]), have been proposed to solve this problem. These algorithms divide the multi-dimensional space into regular grids. Then they assign each data point to a cell in the grid and they perform clustering on the cells instead of the points. This approach is problematic because when dealing with vast amounts of data located into an equally vast grid, it is impossible to store all the grid cells in memory.

When dealing with data in the order of Terrabytes or Petabytes, any algorithm with superlinear running time becomes prohibitively slow. Hence there is a necessity for an algorithm that can perform clustering with running time that is linear to the total number of data points. More specifically we want to read each data point exactly once and we want to perform the theoretical minimum number of disk I/O operations since they incur a heavy performance cost.

In this thesis we present SingleClus, a grid-based clustering algorithm for 2-dimensional arrays that processes each data point exactly once. It runs on linear time in CPU and performs a linear number of I/O operations. Therefore, it is theoretically optimal in terms of run time. Moreover, our algorithm requires only a small percentage of the entire database to be resident in main memory at all times. Therefore it does not suffer from the memory issues that most other algorithms do. Our algorithm improves on the Hoshen-Kopelman clustering algorithm ([9]) and solves some of this algorithm's fundamental deficiencies in the Big Data setting.

1.2 Thesis Contribution and Organization

This thesis contributes a design and analysis of SingleClus, a grid-based clustering algorithm for Big Data on 2-dimensional spatial databases. We present an implementation and analysis of the algorithm and a description of the basic clustering query.

SingleClus differs from previous algorithms in that it is only reading each data point exactly once. In addition, SingleClus works under very modest memory contraints and it is provably the most efficient algorithm for these constraints.

The rest of the thesis is organized as follows. Chapter 2 discusses the background and related work and explains the main deficiencies of existing clustering algorithms. It also introduces the Hoshen-Kopelman algorithm and discusses its challenges when operating in the Big Data setting. Chapter 3 discusses the algorithm design and provides a run-time and space complexity analysis. Chapter 4 provides a qualitative evaluation of SingleClus based on is performance on two synthetic datasets. Chapter 5 concludes and discusses future work.

Chapter 2

Background

This chapter presents background on clustering algorithms, discusses previous literature, and explains the weaknesses of existing algorithms when dealing with very large multi-dimensional datasets. It also mentions briefly how SingleClus will solve these problems.

2.1 Clustering Overview

Clustering is the problem of grouping together elements that are more similar in some way than elements in other clusters. What constitutes "similarity" depends on the type of the dataset and the goal of the clustering. In the multi-dimensional array model, our data are points in multi-dimensional space. Hence similarity is based on proximity. Intuitively speaking, if many points are close to each other then they form a cluster.

There is no universal definition of what constitutes a cluster. As will be discussed below, what constitutes a cluster is highly subjective. Different algorithms produce different clusters on the same dataset based on the models they follow. Moreover, most algorithms take some input parameters by the user. Therefore even the same algorithm will produce different clusters based on the parameters provided by the user. Hence, clustering quality can only be evaluated subjectively.

There are various algorithms for clustering but most of them fall into one of the following categories:

- Centroid-based algorithms
- Connectivity-based algorithms
- Density-based algorithms
- Grid-based algorithms

We proceed to discuss each of these categories below.

2.1.1 Centroid-Based Clustering Algorithms

Centroid-based clustering algorithms are performing clustering by representing each cluster by a central vector. The central vector of the cluster may or may not be a member of the data set. Usually the number of clusters to be found is a fixed integer k. The k-means problem is to find the k cluster centers and assign the objects to the nearest cluster center, such that the squared distances from the cluster are minimized ([2]).

The k-means problem is known to be NP-hard. However, approximate methods have been devised to solve the problem. The most popular such method is Lloyd's algorithm ([6]). Lloyd's algorithm does not find the global optimum to the k-means problem. Instead it returns a local optimum and is commonly run multiple times with different random initializations. One of the benefits of Lloyd's algorithm is that it runs on O(n) time.

Even though fast approximate algorithms such as Lloyd's exist, centroid-based algorithms suffer some serious setbacks. First, the parameter k that the user provides in advance may not coincide with the actual number of clusters in the dataset. In this case, centroid algorithms return very counterintuitive cluster results.

In addition, the algorithms return good quality results only when the clusters are of approximately similar size and spherical in shape. As can be seen in Figure 2-1 the algorithm may produce problematic borders between clusters even if the number

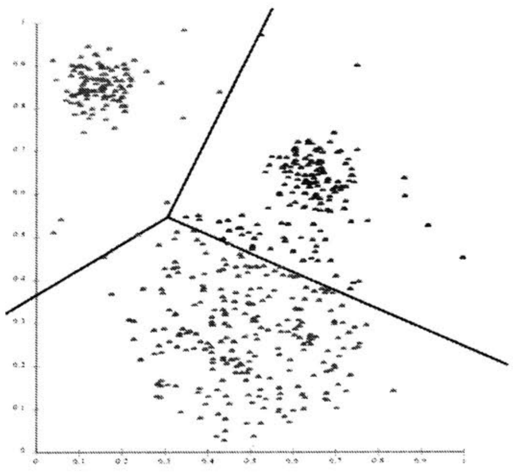

Figure 2-1: Clusters produced by Lloyd's k-means algorithm for k=3.

k of the clusters has been guessed correctly. Intuitively this makes sense because centroid-based algorithms try to optimize the centers of the clusters and not the borders. Variations of k-means often include such optimizations as choosing the best of multiple runs, but also restricting the centroids to members of the data set (k-medoids), choosing medians (k-medians clustering), choosing the initial centers less randomly (K-means++) or allowing a fuzzy cluster assignment (Fuzzy c-means). However in the end, all theses algorithms suffer from the same qualitative issues.

The linear running time of Lloyd's algorithm make it a very tempting choice to use with large datasets. However, the obvious quality setbacks when dealing with an unknown number of randomly shaped clusters eventually make this an inappropriate choice for cluster discovery.

2.1.2 Connectivity-Based Clustering Algorithms

Connectivity-based clustering, also known as hierarchical clustering, is based on the idea that objects that are closer to each other are more related than objects further away from each other. Moreover, in hierarchical algorithms, two data points can be seen as being part of the same cluster or not, depending on the distance resolution we choose to operate on. Connectivity-based clustering algorithms do not create a single partition of the dataset. Instead, they create a hierarchy of clusters that merge when the distance resolution decreases and split when the distance resolution increases. Hierarchical algorithms use a dendrogram data structure to represent the different clusters at different resolutions ([7]).

There are two approaches to implement hierarchical clustering algorithms:

- Agglomerative clustering
- Divisive clustering

Agglomerative clustering is a "bottom up" approach. Each data point begins as a cluster and eventually merges with other clusters when the distance resolution decreases. Divisive clustering is a "top down" approach. All the points start in the same cluster and splits are performed while we increase the distance resolution.

Unfortunately, both approaches are prohibitively slow when operating on large datasets. Agglomerative algorithms' clustering complexity is $O(n^2 \cdot logn)$ and can be improved to $O(n^2)$ for special cases. Divisive algorithms' complexity is $O(2^n)$. In addition, these algorithms assume that the entire dendrogram can be stored in main memory. When we are dealing with very large datasets this is an impossibility. Hence hierarchical algorithms are not a realistic option for very large datasets.

2.1.3 Density-based Algorithms

The most popular clustering algorithms fall into the density-based category. Density-based clustering algorithms define a cluster to be a region where each point has at least a given number of points moderately close to it. The best-known algorithm in this category is DBscan ([1]). DBscan operates by performing a range query for each point it encounters that has not yet been marked as noise and has not been assigned into a cluster. If the neigborhood of radius Eps contains more than MinPts points, then the point is considered a core point and it is put into a cluster. Then, the neigborhood points are put into a queue and we iteratively perform the same procedure to all the points in the queue until there are no more points to be inserted.

DBscan provably satisfies some performance guarantees. It is mentioned in [1] that the algorithm runs in $O(n \cdot logn)$ time, however this is false. In reality, we are performing n range searches which actually brings the running time to $O(n^2)$ because each range search must access entire dataset in the worst case. Recently, Yufei et al. ([4]) dealt with the problem of finding an $O(n \cdot logn)$ algorithm for DBscan. They proved that the exact problem is computationaly intractable when we are operating on 3 or more dimensions. They proceeded to present an approximate algorithm who runs in E[$O(n)$] time. This algorithm produces results that fall between clustering with Eps'$=(1-\epsilon)$Eps and Eps"$=(1-\epsilon)$Eps for all possible values of ϵ.

This result is important as it reinforces the notion that clustering quality is largely subjective. Hence, an approximate algorithm that produces a clustering within ϵ of a target and runs in expected linear time can be sufficient for many applications.

However, we must mention the problems of the above algorithm. To perform the clustering it creates a supporting tree data structure which is implied to reside in main memory. However the size of it is equal to the entire dataset. Hence, this algorithm is impossible to be utilized when dealing with very large datasets. If, instead, only some summary structures are saved in memory the problem of insufficient memory could be solved.

2.1.4 Grid-based Algorithms

The last category of clustering algorithms that we analyze is grid-based clustering algorithms. Grid-based clustering algorithms deal with the problem of vast datasets by dividing the domain space into a grid. Subsequently, they place each point into the grid cell it belongs while updating the point count of each grid cell. Finally, they proceed to perform clustering on the grid cells instead of the points themselves ([8]).

It is easy to see that this approach can help with very big datasets. Instead of having to cluster billions of points, we are now dealing with grid cells whose number can be orders of magnitude less than that of the points. Therefore, there is the possibility to actually perform clustering on main memory.

In the existing bibliography, it is usually assumed that the grid cells can all be saved in main memory. Even if we were to reduce the entire grid cell into a bitmap (where a cell with a 0 indicates a cell that is empty or that does not have a significant number of points in it and a cell with a 1 has enough points inside) then it is still possible that the grid too big to fit in main memory. Moreover, saving only a bitmap is insufficient to save important information about the cell. Indeed, at the very least we need to store some information regarding the cluster membership of each cell. Another problem is that if the bitmap is indeed able to fit in main memory, sometimes this only happens at the cost of resolution. This means that if we choose a large enough side for our grid cells then it will be definitely possible to fit the grid in main memory. However, this is usually a bad idea since the quality of the clusters produced is significantly deteriorated.

2.1.5 The Hoshen-Kopelman Clustering Algorithm

In their paper ([9]) Hoshen and Kopelman propose a multi-label clustering algorithm on a 2-dimensional grid of cells. The algorithm was intended to have application in the area of percolation theory. We will discuss the algorithm, explain what are the challenges and shortcomings in our setting and finally explain why our adaptation is a good model for clustering Big Data.

Algorithm Input

The input to the Hoshen-Kopelman algorithm (henceforth HK) is a grid in 2-dimensions. Each grid cell either contains a 1 or a 0. Only cells containing a 1 are members of clusters.

HK Cluster Definition

Definition: A *grid cluster* C is a **maximal** set of cells such that for every 2 cells $a, b \in C$, there exists a sequence $a = c_1, c_2, ..., c_l = b$ such that:

- $c_i \in C$ for all $i = 1, 2, .., l$
- c_i and c_{i+1} are neighbours for $i = 1, 2, .., l$

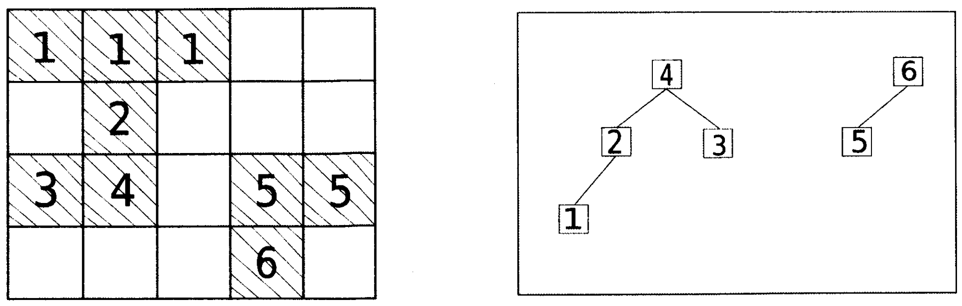

Figure 2-2: Output of Hoshen-Kopelman Algorithm. On the left is the grid with each full cell containing a cluster ID. On the right are the trees representing connections between cluster IDs.

Two grid cells are **neighbours** if they share a common side. To make this concept more precise, if we are given two cells c and d with coordinates $c = (c_1, c_2)$ and $d = (d_1, d_2)$ then c and d are neighbours if and only if

$$\sum_{i=1}^{2} |c_i - d_i| = 1$$

Algorithm Output

The output to the algorithm is another grid, of the same size as the original grid, such that each grid cell contains a cluster ID.(In most applications the old grid just transforms into the new grid). Two cells can contain the same or different cluster IDs. The cluster IDs are organized in a data structure that is modeled as a set of disjoint trees. Each tree contains some IDs and every ID belongs in one tree exactly. If a grid cell does not belong in any cluster then it contains a cluster ID equal to 0. The rest of the cells contain positive integer cluster IDs. Two grid cells that belong to the same cluster, according to the definition given above, contain cluster IDs that belong to the same tree. Hence, the tree data stucture and the cluster ID of each grid cell are together needed to determine whether two grid cells belong to the same cluster.

For example, in Figure 2-2 we see the result of HK clustering on a 2-dimensional cluster. The tree structure corresponding to this clustering is shown in the right. The roots of the trees are IDs 4 and 6. The roots of each tree are coinciding are their own parents.

HK Algorithm Description

We will briefly discuss how the HK algorithm works. We will describe how the algorithm takes the input discussed above and performs the clustering stage.

The algorithm operates by performing a raster scan of the grid. It traverses the grid in row major order. If a grid cell c contains a 0 then we just proceed to the next grid cell according to the row major order. If a grid cell c contains a 1 then we invoke a neighbour finding function which returns the coordinates of all the cell's neighbours that are before the current cell in the global order.

Once we have found all the previous neighbours of cell c we check their assigned IDs by accessing their locations in the grid (Since they are *previous* neighbours, they have already been visited and hence have been assigned an ID). We collect all the assigned cluster IDs of the neighbours. To do so we only need to access grid cells in the current level and in the previous level of the grid.

Definition: A *grid level* C is the set of all cells that have the same first coordinate.

If all the cluster IDs of the neighbours are equal to 0 then it means that they are not part of any cluster. In this case, we increase a global counter variable GlobalID by 1 and we assign the new value of GlobalID to grid cell c.

If one of the 2 previous neighbours contains a non-zero cluster ID then there are 2 cases:

- Both neighbours contain the same cluster ID. In this case we just assign this ID to our current grid cell c.
- There are 2 different cluster IDs in the neighbours' grid cells. In this case we have to merge the IDs to indicate that they all represent the same cluster. This merging can be done in various ways but the main idea is common in all methods. First we just choose one representative cluster ID x among those

found (usually the one with smallest or largest numerical value). Then we set the root of the tree in which the other ID is contained to point to the root of the tree that x is contained. This is usually implemented through the UNION-FIND ([5]) algorithm or through variations thereof. Then we assign the value of x to the grid cell c.

In the end of this raster scan we are left with the grid and the tree structure. There is the possibility that there are trees of size 1 (i.e. single nodes) in our tree structure if there were no merge operations for that node. Each cell in the grid now either contains a 0 (if it had a 0 to begin with) or a poistive integer cluster ID. Two grid cells belong to the same cluster if and only if their cluster IDs belong to the same tree in the tree data structure.

Challenges of Hoshen-Kopelman

There are various challenges when adapting the Hoshen-Kopelman algorithm in a general Big Data setting. We discuss them here. We also give some high level description about how we solve those issues with SingleClus. All of these will be discussed in detail in the next chapter.

Clustering Data Points: The first challenge is that HK is used to cluster grid cells containing a 0 or 1 value. However, in a general setting we are interested in clustering 2-dimensional *points*. There is a precise definition (which we provided) of what constitutes a neighbour for a grid cell. The same is not true for random points. The solution that we propose is the same that is used by many other grid-based algorithms. Specifically, for each point with coordinates $p = (p_1, p_2)$ we can determine the coordinates of the grid cell it is contained in. To achieve this we need some domain knowledge. Namely, we assume that we are familiar with the range of possible values $[Min_i, Max_i]$ along each dimension i. Moreover we assume that we know the length $cellSide_i$ of the side of each grid cell along dimension i (we assume a uniform grid). Therefore finding the grid cell is equivalent to performing a divisions of each of the 2 coordinates p_i by the respective $cellSide_i$.

Moreover, to emulate the binary nature of the grid (i.e. that each grid cell contains either a 0 or a 1) we define a threshold T_h which is a positive integer. If at least T_h points fall within a given grid cell then it is considered *full*, else it is considered *empty*. T_h and *CellSide* can be viewed as analogous to DBScanma's *MinPts* and *Eps* user-defined constants ([1]). With these modifications in mind, a data point cluster consists of all the points that are contained in grid cells that are themselves part of a *grid* cluster in the paradigm of Hoshen-Kopelman.

Memory Bottleneck: Another challenge when using the Hoshen-Kopelman algorithm with Big Data is the potential of a memory bottleneck. By assigning each data point to a grid cell we are reducing the size of our dataset, however the grid itself may still be too large to fit in memory. This problematic situaton is briefly acknowledged in [9]. The authors mention that only two levels of the grid need to reside in main memory at all times and they are correct. However they fail to acknowledge that the tree data structure (or at least parts of it) needs to be in main memory during clustering and this structure can also get too large for the memory. Hence we will have to deal with this issue in our algorithm as well.

The final two challenges of clustering Big Data with HK are the most important and herein also lies the biggest contribution of this thesis.

Query Performance and Disk I/O: There is one query that all clustering algorithms have to be able to answer successfully. This is the *cluster membership query*. In simple terms, given a grid cell c_1, we want to know if another cell c_2 belongs to the same cluster as c_1. The grid can't be saved in main memory in its entirety as we have mentioned. Therefore, any cluster membership query will involve at least 2 disk reads to bring in memory the information pertaining to the cells that we want to query. One of the main drawbacks of the simple form of the Hoshen-Kopelman algorithm is that each tree in the tree structure contains potentially many nodes. So, the only way to determine whether two IDs belong to the same tree (and by extension to the same cluster) is to traverse the tree until we reach the root and then check if the root IDs of the two IDs are coinciding. Unfortunately this can take a long time. Specifically, in the worst case, a tree with n nodes can have height equal to n (in the case where it is isomorphic to a linked list). The average time it takes to reach the root then is $O(n/2) = O(n)$ which is prohibitively slow when we want to perform many such queries.

To make matters even worse, the tree may not even fit in main memory in its entirety so we have to perform multiple I/O operations just to answer a simple query. The obvious solution to this problem is that we have to perform a second pass of the grid and "flatten out" the trees. This means that we want all the cluster IDs contained in a tree to point to the root of the tree. This was suggested in [9], however no specific implementation was proposed. A challenge that has to be dealt with is the fact that only parts of the tree structure can be stored in memory at any given time and therefore we have to perform various I/O operations to read parts of the tree structure in memory.

We mentioned briefly in the introduction that one of the main challenges of clustering (and processing in general) Big Data is the fact that memory is a limited resource. As we mentioned in the previous paragraph, this is the case even if we reduce the problem to that of clustering on a grid. When processing vast amounts of data we have to read each data point in main memory *at least once.* Moreover we have to write back in disk information about each grid cell. Hence, we need to eliminate all I/O operations that go beyond the absolutely essential ones. In the research bibliography, there hasn't been a succesfull attempt to provably minimize the number of I/O operations.

A proposed solution to this problem was suggested in ([11], [10]) where the tree structure is discarded and instead each cluster has a representative *grid cell* (and not ID). The papers propose a two pass algorithm and after the end of the second pass of the grid all the grid cells in a cluster contain the same cluster ID. There are some problems with the proposed approach:

- During the first grid pass, every time that a merging of clusters occurs, we have to access the representative cells of the relevant clusters. However, there is no guarantee that these cells will be already in memory, especially if they are in much earlier grid levels. Hence, every time a merge operation occurs, multiple

I/Os must happen.

- Another problem is that the representative of a cluster may change during merge operations, however, grid cells labeled with the prior representative's ID may still participate in merging operations. Hence, for every merge operation that involves these grid cells, we have to do what amounts to a tree traversal in order to find the very latest representative cell of a given cluster. If the merge operations become sufficiently complicated, this process can become really time-consuming (many I/O operations **and** in-memory cost)
- Additionally, in the second pass of the grid, when a grid cell is encountered, in order to determine its final cluster ID we access the representative cell of the relevant cluster once more. Hence, again we have to perform I/O operations to bring the relevant data in memory.

The number of these additional I/O operations depends on the structure of the grid and, specifically, on the number of tree merge operations during the first grid scan. However the exact number of I/O operations that are absolutely necessary is just those required to read the grid into memory twice as we will show.

In our algorithm we propose a method that accesses each *data point* exactly once. In addition, and most importantly, we read each grid cell into memory exactly once and we write it exactly twice back into disk. Hence, we perform the *theoretically optimal number of I/O operations.* Specifically we will prove the correctness of our algorithm when operating with modest (compared to the size of the entire array) memory requirements.

2.1.6 The Contributions of Our Algorithm

We suggest that our algorithm SingleClus is the only clustering algorithm for very large 2-dimensional datasets that satisfies the following requirements:

- **i)** It clusters data points and not just cells.
- **ii)** It accesses each data point exactly once.

- **iii)** It creates auxiliary structures (grid and tree structure) that are written twice in disk and only read once back in memory.

- **iv)** It requires only a minor memory budget compared to the total size of the array.

Specifically (ii) and (iii) are providing theoretical optimality in terms of performance. These contributions make SingleClus an attractive option for clustering very large multi-dimensional arrays.

Chapter 3

Algorithm Design

This chapter presents the design, implementation and analysis of SingleClus. We begin by presenting our operational model. We discuss the memory and disk structures we will be using and we present theoretical results that prove that SingleClus is correct and it is performing the optimal number of I/O operations.

3.1 Disk Model

The data model that we will be using is the 2-dimensional array model. The input to our algorithm is an array in CSV format. Each line represents a 2-dimensional data point. Each coordinate is assumed to be an integer, however we can easily generalize to other types. For each point $x = (x_1, x_2)$, we know that x_i is always in a range $[Min_i, Max_i]$. We assume we know this range in advance. We save these ranges in four constants $Min_1, Max_1, Min_2, Max_2$. The combination of the 2 ranges determines the 2-dimensional space boundaries.

The data points are assumed to be sorted in **row major order**. This means that points are first sorted by the first coordinate, and then by the second coordinate. This assumption *does not affect clustering efficiency* because clusters can be of any shape and can extend towards any dimension.

Finally, we assume that the input dataset is in a file InputArray. We assume that the number of data points is too large for all the points to be loaded in main memory simultaneously. Hence, multiple I/O operations will be necessary to access the entire dataset. Therefore, algorithms that assume that the data can reside in memory can't be utilized.

In our discussions we will follow the disk access model ([3]). This means that we acknowledge that I/O operations will take the majority of time compared to memory operations. Therefore, we will be attempting to minimize the number of disk I/O operations we will be performing. Additionally, our algorithm achieves linear in-memory performance. Even though not all disk I/O operations take the same time (depending on where each disk block requested is located in the disk) we accept this model as a good first order approximation.

3.2 Constants

We utilize the following constants.

T_h**:** This is a user-defined positive integer constant that indicates the number of points that have to be contained within a grid cell in order for it to be considered *full*. Every *full* cell is part of a grid cluster. If a grid cell does not contain at least T_h points then it is considered *empty*.

$(CellSide_1, CellSide_2)$**:** These are 2 user defined positive constants that indicate the extent of a grid cell along each dimension. If we want the grid cells to have the same extent along each dimension then we just set these 2 constants equal to each other. For the rest of the discussion, we just assume that both the extents are equal to $CellSide$.

(M_1, M_2)**:** M_i is the number of grid cells that exist along dimension i. It can be calculated by

$$M_i = \left\lfloor \frac{Max_i - Min_i}{CellSide_i} \right\rfloor + 1$$

.

$MaxPoints$**:** This is a constant that depends on the size of the dataset, as well as to the memory capacity of the system that will be running SingleClus. It is defined to be the largest possible integer such that $(4 \times M_2 + M_1 + 2 \times MaxPoints)$ integers can fit into main memory at any given time (excluding of course parts of the memory that are occupied by program execution, kernel code etc). It represents how many points from InputArray we are reading into memory at once during the clustering phase. Therefore, the larger $MaxPoints$ is, the less I/O operations we will have to perform to access each data point.

The significance of the $4 \times M_2 + M_1$ term will become obvious in the next section. Even though we assumed that the grid itself may be too large to fit in main memory (and therefore $M_1 \times M_2$ integers do not fit in memory), we assume that $4 \times M_2 + M_1$ integers can always fit in main memory. This is a size equal to $\frac{5}{M_1}$ of the size of the entire grid if we assume M_1, M_2 are comparable in size. M_1 can be in the tens of thousands so we only have to keep a very small portion of the grid in main memory at all times.

3.3 Data Structures

3.3.1 Grid

Grid is just a file that resides in disk. It starts empty and progressively (during the first clustering phase) it gets filled with entries containing cluster IDs representing the contents of the grid cells. We don' save the coordinates of each grid cell along the cell's cluster ID. We can infer the grid cell's coordinates simply by noting the offset of each element in Grid and then using the getCoordsFromID function (described in the *Algorithm Description* Section).

3.3.2 TreeStruct

It was explained in the previous chapter that each cluster is represented by a set of IDs. These IDs are organized as a set of disjoint trees so that two grid cells are parts of the same cluster if and only if their IDs belong to the same tree. The way we represent this trees is simple. We save them in a file called TreeStruct. TreeStruct contains a single integer at each line. If integer j is contained in the i-th line, this indicates that the parent of ID i is ID j. If an ID is the root of its respective tree, then it is pointing to itself. Hence when we have a situation where the integer contained in the i-th line is i itself, it indicates that i is the root of its tree.

When merging two trees whose roots have IDs i and j, we just have to change ID j to point to i instead of pointing to itself (or vice versa). This operation happens in memory in the CurrentTreeList and PreviousTreeList arrays (we will describe those in the *Memory Buffers* subsection). An important thing to notice is that the size of TreeStruct is at most equal to the size of the Grid. The reason for this is that each cluster ID has to belong to at least one grid cell. An important result of this observation is that *traversing the entire TreeStruct takes less than or equal amount of time to traversing the entire Grid* (and less than or equal number of I/O operations).

3.4 Choosing T_h and CellSide

Unfortunately, there is no good way to determine *a priori* the T_h and *CellSide* parameters efficiently so that we produce good quality clusters. We will have to do so heuristically (i.e. intuitively) and check the quality of our cluster *a posteriori.* This could mean that we would have to go through a lot of bad guesses before we find the right combination.

When we are operating with Big Data we might not have the luxury to perform multiple clusterings of the entire dataset before we find the right choice of T_h and *CellSide.* This is a problem that is discussed briefly in Chapter 4. One possible attempt to find good T_h and *CellSide* values, is to read a small portion of the dataset into memory and perform clustering on them with various choices of T_h and *CellSide.*

3.5 Memory Buffers/Arrays

PointsBuffer This array just holds $MaxPoints$ data points from InputArray during the first clustering phase. Every time we finish processing all the points in PointsBuffer we load a new batch of points from the disk until we have read all the points from InputArray.

CurrentLevel This is a list of length M_2 and it represents the grid cells of the current level (by "current level" we mean the grid level that we are operating on at a given instant). For each grid level, while we are reading points from PointsBuffer, CurrentLevel gets populated with integers indicating how many points are contained within each grid cell in this level. If at least T_h points have fallen within a grid cell then this grid cell is *full.* After we have read all data points at a given grid level, we start performing clustering on the grid cells of that level. After this step, each entry in CurrentLevel will get updated once again. After the updates, each entry will contain either a 0 or a cluster ID.

PreviousLevel This is a list of length M_2 containing the cluster IDs of the grid cells at the *previous level* of the one we are currently operating on. This array is necessary for the clustering step because it allows us to determine cluster connections between the current and previous levels.

CurrentTreeList This is a list of length **at most** M_2 that contains all the parent relationships of the cluster IDs that were created in the current grid level. It is used in the first clustering phase to perform merging operations and in the second phase to perform the tree flattening operations. CurrentTreeList is accompanied by a variable *CurrentTreeIndex.* This variable holds the ID which is represented in the first location of CurrentTreeList. This is just another way of stating that CurrentTreeList[i] is the parent of ID *CurrentTreeIndex*$+i$. For example, if CurrentTreeList[2]$=22$ and *CurrentTreeIndex*$=15$ then the parent of ID $17=$*CurrentTreeIndex*$+2$ is ID 22.

PreviousTreeList This is a list of length **at most** M_2 that contains all the parent relationships of the cluster IDs that were created in the *previous* grid level. It is used in the first clustering phase to perform merging operations and in the second phase to perform the tree flattening. PreviousTreeList is also accompanied by a variable *Pre-*

viousTreeIndex. This variable holds the ID which is represented in the first location of PreviousTreeList. This is just another way of stating that PreviousTreeList[i] is the parent of ID *PreviousTreeIndex*+*i*.

OffsetIDList This is a list of length exactly M_1. It is always entirely in main memory. OffsetIDList[i] contains the ID of the last cluster ID that was created while clustering the i-th grid level during the first clustering phase. It will be useful in the tree flattening phase of clustering. From the way our algorithm works, OffsetIDList[i] is equal to OffsetIDList[$i-1$] plus the length of CurrentTreeList at the end of clustering the i-th grid level.

It is true that at any instant *PreviousTreeIndex*+length(PreviousTreeList)=*CurrentTreeIndex*.

3.6 Algorithm Description

Here we will discuss how our algorithm works. We will be referencing the structures mentioned above. Before we start, we introduce some useful definitions and vocabulary to make the discussion more succinct.

3.6.1 Vocabulary

We introduce some terminology to facilitate the description of the algorithm.

- A grid level L_i is the set of all grid cells who have their first coordinate equal to i. Hence, knowing M_1, the first level is L_0 and the last level is L_{M_1-1}.
- We will refer to grid cells by their two coordinates. Hence, c_{ij} is the grid cell that has coordinates (i, j). Therefore c_ij is located in the j-th position at level L_i.
- We will be usign capital letters to represent cluster IDs (except the letters T and L that represent trees and levels respectively). Hence, A,A',B,X,X_i are all valid cluster ID symbols.

- At any specific time instant during clustering, we will denote the cluster ID that is contained in a grid cell c_{ij} at that time instant by writing $\mathsf{id}(c_{ij})$. The mentioned grid cell can either be in the level we are currently clustering or at any other level. For example, if cluster ID 4 is contained in grid cell $c_{1,3}$ then $\mathsf{id}(c_{1,3})$=4.

- At any specific time instant during clustering, we will denote by $\mathrm{T}(X)$ the tree that contains cluster ID X at that specific time instant. For example, if T_1 is the tree that contains cluster ID 4, then $T_1 = \mathrm{T}(4)$.

- At any specific time instant during clustering, we will use $\mathsf{root}(T_i)$ to denote the root ID of a tree T_i at that instant. For example, if the root of the tree that ID 4 is contained in is 2 then $\mathsf{root}(\mathrm{T}(4))$=2.

- At any specific time instant during clustering, we will use $\mathsf{parent}(X)$ to refer to the parent of a cluster ID X in its tree at that specific time instant. If an ID is the root of its own tree then $\mathsf{parent}(X)=X$ (and vice versa).

- We will use $\mathrm{ID}(L_i)$ to denote the set of all cluster IDs contained in grid cells at level L_i at any specific time instant. This set does not necessarily coincide with the set of all the cluster IDs that were created during clustering L_i. The reason is that there is the possibility that, at some point, grid cell c_{ij} contained ID X (where X was created during the clustering of L_i) but before the clustering of the level was complete, the cell's cluster ID was changed. Therefore, at that instant, X may no longer belong in $\mathsf{ID}(L_i)$ even though it was created during the clustering of L_i

- We will denote by Grid[0:i] the sub-grid containing all levels up to, and including, L_i. A cluster in a sub-grid Grid[0:i] is defined in exactly the same way as in the entire Grid, however we only take into account cells up to the i-th level. Two clusters that are separate in Grid[0:i] may end up merging further down in the grid.

3.6.2 Functions and Variables

GlobalID: This is a variable that indicates how many different cluster IDs have been created thus far. It coincides with the largest cluster ID assigned to a grid cell at any time.

CurrentLevelCount:

This is a variable that indicates how many different cluster IDs have been created thus far *in the current level*(i.e. the one we are currently clustering).

findPointCell

The code for findPointCell is as follows:

```
def findPointCell(CoordList):
  CellCoords=[]
  for i in range(2):
    CellCoords.append((CoordList[i]-Min[i])/CellSide)
  return CellCoords
```

This functions takes as input the coordinates of a data point p. It returns the coordinates of the grid cell c that this p is contained in. It runs in $O(2) = O(1)$ time. A point $p = (p_1, p_2)$ is contained within cell $c = (c_1, c_2)$ if and only if

$$c_i \leq \left\lfloor \frac{p_i - Min_i}{cellSide_i} \right\rfloor < c_i + 1$$

for $i = 1, 2$.

convertCoordsId

The code for convertCoordsId is as follows:

```
def convertCoordsId(CoordsList):
  return CoordList[1]
```

Returns the index in CurrentLevel of the grid cell with coordinates *CoordsList*. For a grid cell $c = (c_1, c_2)$ its position in CurrentLevel is just c_2.

findLevel

The code for this function is as follows:

```
def findLevel(CoordList):
  return (CoordList[0]-Min[0])/CellSide
```

This function just returns the grid level that a *data point p* with coordinates *CoordList* is contained in.

findRoot

The code for this function is as follows:

```
def findRoot(id):
  if id>=CurrentTreeIndex and id<=GlobalID:
    newid=CurrentTreeList[id-CurrentTreeIndex]
    if newid==id:
      return newid
    else:
      return findRoot(newid)
  elif id<CurrentTreeIndex and id>=PreviousTreeIndex:
    newid=PreviousTreeList[id-PreviousTreeIndex]
    if newid==id:
      return newid
    else:
      return findRoot(newid)
```

This function takes as input a cluster ID X and returns the ID of root(T(X)). As we will prove later in the discussion, we only to need search this root in the two buffers CurrentTreeList and PreviousTreeList.

setParent

The code for this function is as follows:

```
def setParent(X,Y):
  if CurrentTreeIndex>=PreviousTreeIndex: \\used during phase 1
    if X>=CurrentTreeIndex and X<=GlobalID:
      CurrentTreeList[X-CurrentTreeIndex]=Y
    elif X<CurrentTreeIndex and X>=PreviousTreeIndex:
      PreviousTreeList[X-PreviousTreeIndex]=Y
  else: \\used during phase 2
    if X>=PreviousTreeIndex:
      PreviousTreeList[X-PreviousTreeIndex]=Y
    elif X<PreviousTreeIndex and X>=CurrentTreeIndex:
      CurrentTreeList[X-CurrentTreeIndex]=Y
```

This function takes as input two cluster IDs X and Y and sets parent(X)=Y.

3.6.3 Clustering Phases

The algorithm operates in 2 phases. In the first phase we scan all the data points and we progressively build Grid and TreeStruct. In the second phase we access and modify Grid and TreeStruct in such a way that all grid cells in a cluster share the same ID in the grid. We present those two phases separately.

First Phase (Raster Scan)

Initially the algorithm reads a batch of *MaxPoints* points from InputArray into PointsBuffer. We start accessing the points in PointsBuffer. For each point p, we find the grid cell c that it belongs in by using convertCoordsID. Then we increase c's count in CurrentLevel by 1. Once we have encountered a point that does not belong to the same grid level as the previous point, it means that we have read all the points on that grid level. The logic behind this is that we access points in row major order. Therefore the first coordinate of a point (the one that determines the grid level) can never become smaller as we access more points. If we reach the end of PointsBuffer without ever changing grid level, then we read another batch of *MaxPoints* points from InputArray into PointsBuffer.

When we first encounter a point that does not belong in the same level L_i as the previous point, we start the clustering step for L_i.

The clustering of each level L_i during the Raster Scan phase, is happening in **3 stages**.

Stage 1:

We start reading the elements of CurrentLevel in order. If a grid cell c_{ij} has value smaller than T_h (i.e. if CurrentLevel[j]$< T_h$) this means it does not belong to any cluster. We set CurrentLevel[j]=0 (i.e. id(c_{ij})=0) and proceed to $c_{i,(j+1)}$.

Alternatively, if c_{ij} has value greater than or equal to T_h in CurrentLevel, then it means that it belongs to a cluster. We find the values X=id($c_{(i-1),j}$) and Y=id($c_{i,(j-1)}$) from PreviousLevel[j] and CurrentLevel[$j-1$] respectively. Since we are traversing the grid cells in row major order, both of these grid cells have already been processed and have been assigned a cluster ID. There are the following 4 cases regarding X and Y:

- **X=Y=0 :** In this case, neither of the two previous neighbours belongs to a cluster. We let *GlobalID*=*GlobalID*+1 and set CurrentLevel[j]=*GlobalID*. Moreover we set parent(*GlobalID*)=*GlobalID* in CurrentTreeList.

- **X=0,Y≠0 :** In this case, we just set id(c_{ij})=Y in CurrentLevel and proceed to the next cell in L_i.

- **Y=0,X≠0 :** In this case, we find R=root(T(X)) and set id(c_{ij})=R in CurrentLevel.

- **X≠0,Y≠0 :** In this case we find R_1=root(T(X)) and R_2=root(T(Y)). If $R_1 < R_2$ then set id(c_{ij})=R_1 and parent(R_2)=R_1. If $R_2 < R_1$ then set id(c_{ij})=R_2 and parent(R_1)=R_2. If $R_1 = R_2$ set id(c_{ij})=R_1.

The last case is the only case where two separate clusters can merge. The above task is repeated for all $c_{ij} \in L_i$.

Stage 2:

Once we have processed the last grid cell in L_i (through accessing and modifying CurrentLevel) we start processing all the grid cells in L_i again but in *reverse row major order*. Specifically, starting from c_{i,M_2-1} and proceeding all the way down to c_{i0}, for each cell c_{ij} we find X=id(c_{ij}) in CurrentLevel. Then we set id(c_{ij})=root(T(X)) and set parent(X)=root(T(X)). This simply means that we set the cluster ID of each cell to be the root of the tree that its previous ID was contained in.

Stage 3:

Finally, we process each grid cell c_{ij} on L_i one more time in row major order to ensure that $\text{ID}(L_i) \cap \text{ID}(L_{i-1}) = \emptyset$. Let G the value of *GlobalID* after the end of Stage 2 but before Stage 3 begins. Let R=root(T(id(c_{ij})). Then, if $R \leq G$, we increase *GlobalID* by 1. Then we assign id(c_{ij})=*GlobalID* in CurrentLevel and we set parent(R)=*GlobalID* and parent(*GlobalID*)=*GlobalID* in CurrentTreeList. If $R > G$ then set id(c_{ij})=R.

Once all three stages have been completed for L_i then we do the following:

- We write all the elements of PreviousLevel into Grid at the disk.
- We set PreviousLevel equal to CurrentLevel (this can be implemented efficiently by changing pointers if we don't want to copy the entire list).
- We empty CurrentLevel.
- We add an entry to OffsetIDList that is equal to the last entry of OffsetIDList plus *CurrentLevelCount*. This is how many new cluster IDs were created on that level.
- We write the contents of PreviousTreeList into TreeStruct.
- We set PreviousTreeList equal to CurrentTreeList.
- We set *PreviousTreeIndex* equal to *CurrentTreeIndex* and *CurrentTreeIndex* equal to *GlobalID*+1.

- We empty CurrentTreeList.

Once we have read all the points from InputArray and have performed all 3 clustering stages for each level L_i then the first phase of the clustering is done. After the first phase is over we do not access InputArray again. The value of *GlobalID* at the end of the first phase is the total number of different cluster IDs that were created and saved in TreeStruct.

Second Phase (Tree Flattening)

The main idea of the second phase is the following:

We access the grid levels in **reverse order** (i.e. the last level we access is L_0), however we access the grid cells within each level in **row major order**

At first we bring into memory (specifically into CurrentLevel) the contents of the last level of Grid. We can do so because we know M_1, M_2 and that each grid cell contains an integer. Therefore we know the byte offset of the first and last grid cell at the last level of Grid. In addition, we bring into CurrentTreeList the entries of TreeStruct that correspond to the IDs of the last grid level. This is possible because we saved this information in OffsetIDList and we know that each cluster ID is an integer. We initialize PreviousTreeList to be empty.

Then we start accessing each grid cell c_{ij} in L_{M_1-1} in CurrentLevel in the normal row major order. For each c_{ij} let X=id(c_{ij}). We perform the following:

- We find R=root(T(X)).
- We set id$(c_{ij})=R$ in CurrentLevel.
- We set parent$(X)=R$ in CurrentTreeList. The index of X is always either in CurrentTreeList or in PreviousTreeList as we will show below.

Once we have finished processing CurrentLevel, we write the updated contents of CurrentLevel in Grid (in the same location we read them from) and bring another M_2 entries from Grid into CurrentLevel. Moreover we move the contents of CurrentTreeList intro PreviousTreeList. Finally, we access OffsetIDList[$M_1 - 2$] IDs from TreeStruct into CurrentTreeList. There is no need to write anything back into TreeStruct.

We continue until all grid levels have been processed. This completes the algorithm. In the end of the Tree Flattening phase, if two grid cells c_{ij}, c_{kl} are in the same cluster then id(c_{ij})=id(c_{kl}) in Grid. We can discard the TreeStruct structure entirely at this point.

3.7 Assumptions

In this section we mention the assumptions we made above in the design of our algorithm.

- We assumed that we know the ranges of values $[Min_i, Max_i]$. This is reasonable because, just knowing what the data points are representing, we can determine the ranges. For example, if we are clustering tweets, then the coordinates are just longitudes and latitudes. If we are clustering tweets in Cambridge, MA, then we can restrain our range within the coordinates that encompass Cambridge.

- We assumed that our input data are **sorted** in disk. This is a reasonable assumption. The reason is that many datasets already come sorted or partially sorted. In addition, the sorted order does not provide any benefit in the sense that the clusters can be of any shape. Particularly,since the data is sorted by x coordinate, we may encounter clusters that are elongated along the y coordinate. Hence we receive no benefit in terms of performance.

  If we really wanted to receive unsorted data then we can sort them in $O(\frac{n}{b}\log_{\frac{M}{b}}\frac{n}{b})$ I/O operations ([3]) where n is the number of points to be clustered, b is the size of a disk block and M is the size of the memory. With modern SSD speeds (90k IOPS random access) and moderate sized memories (say M=4Gbyte) and the usual size of disk blocks (b=4Kbyte) sorting 10TBytes of data can be done in around 5 hours. This is definitely not optimal but, if neccessary, it will have to be done once.

3.8 Correctness

Here we prove that our algorithm is correct. To do so we will first prove the following theorem.

Theorem 1. During the first phase (Raster Scan) the following statements are true:

- **1.1** Suppose that L_i and L_{i+1} are two consecutive grid levels. Then, during the first clustering phase, after we have finished clustering both L_i and L_{i+1}, it is true that $\mathrm{ID}(L_i) \cap \mathrm{ID}(L_{i+1}) = \emptyset$.

- **1.2** If two grid cells c_{ij}, c_{kl} belong to the same cluster, and we have finished clustering both of their levels L_i and L_k during the first phase, then $\mathrm{T}(\mathsf{id}(c_{ij})) = \mathrm{T}(\mathsf{id}(c_{kl}))$. Therefore, the cluster IDs in c_{ij}, c_{kl} belong to the same tree in TreeStruct.

- **1.3** During the first clustering phase, for each grid cell c_{ij} in L_i, after we have finished clustering L_i but before we start clustering L_{i+1}, it is true that $\mathsf{root}(\mathrm{T}(\mathsf{id}(c_{ij}))) = \mathsf{id}(c_{ij})$.

- **1.4** After we have finished clustering L_i **and** L_{i+1}, then for $X \in \mathrm{ID}(L_i)$ either $\mathsf{root}(\mathrm{T}(X)) = X$ or $\mathsf{root}(\mathrm{T}(X)) \in \mathrm{ID}(L_{i+1})$.

- **1.5** During the clustering of grid level L_{i+1} and at any of the 3 stages, if X is a cluster ID in $\mathrm{ID}(L_i)$ or in $\mathrm{ID}(L_{i+1})$, then the root R of $\mathrm{T}(X)$ is either in $\mathrm{ID}(L_i)$ or in $\mathrm{ID}(L_{i+1})$. This proves that we can perform the First Clustering Phase while only keeping CurrentTreeList and PreviousTreeList in memory (and not the entire TreeStruct)

Proof of Theorem 1:

We will prove this by induction on last level L_k that we finished clustering during the first clustering phase. For $k = 0$ (i.e. L_0) all of the assertions are obviously true from the description of the algorithm. Assume that the above holds for $k \leq d$. We will show that it is true for $k = d + 1$.

Theorem 1.1 can easily be proven. Suppose that we are in the process of clustering L_{d+1} after we finished clustering L_d. Then, at **Stage 3** of clustering L_{d+1}, each grid cell will be assigned a cluster ID that has never been assigned in L_d (from the definition of *GlobalID* and the constant G which is described in **Stage 3** as the maximum *GlobalID* up to this point). Hence Theorem 1.1 is proven.

To show that **Theorem 1.2** is true we just have to prove it for the cluster IDs that got merged during the clustering of L_{d+1}. Obviously the cluster IDs that were in the same tree before the clustering of L_{d+1}, will remain in the same tree afterwards. All the merging operations happen during **Stage 1** and particularly when we are dealing with **Case 4** of the ones mentioned in the algorithm's description. It is obvious that, if two clusters that had separate trees after the clustering of L_d are in the same cluster when L_{d+1} is taken into account, then there has to be sequence of grid cells belonging in L_{d+1} that connect two cells in L_d that belonged in the two separate clusters. Otherwise, the two clusters are not merged during the clustering of L_{d+1}. Therefore it is obvious that Theorem 1.2 is true.

To show that **Theorem 1.3** is true we just focus on **Stage 2** and **Stage 3** of clustering level L_{d+1}. It is obvious that during **Stage 2** there are no new cluster IDs created, nor are any trees merged. Hence the roots of trees do not change during **Stage 2**. Moreover, during **Stage 2**, every grid cell c_{ij} is modified such that $\mathsf{id}(c_{ij})=\mathsf{root}(\mathrm{T}(\mathsf{id}(c_{ij})))$. Thus, in the end of **Stage 2**, each grid cell will contain a cluster ID that is the root of its tree. Moreover, in **Stage 3** we just add an extra level to each tree by letting the root of each tree point to a new cluster ID *while at the same time* adjusting the cluster ID in each grid cell in L_{d+1} to point to this new root. Hence **Theorem 1.3** is proved.

It is obvious to show that **Theorem 1.4** is true. To do so, assume that for some X $\in$ID(L_d), root(T(X))$\neq$ X. From **Theorem 1.3**, right after L_d was clustered it must have been true that root(T(X))=X. The only way for this to change is if the root of T(X) got changed during clustering of L_{d+1}. But the only **new** cluster IDs that are turned into roots of trees right after the clustering of L_{d+1} are those contained in the grid cells of L_{d+1} after **Stage 3**. Hence this theorem is also proved.

Finally, to prove **Theorem 1.5** we only need to notice that, from **Theorem 1.4**, after the clustering of L_i, for each ID X $\in$ID(L_i) either root(T(X))=X or root(T(X))$\in$ID(L_{i+1}). So when we introduce a new level L_{i+1} there iare only two possibilities for the root of a tree; either it is an ID that appeared in the previous level L_i, hence it is in PreviousTreeList, or it is an ID that was introduced in the current level L_{i+1}, hence it is in CurrentTreeList. Thus, **Theorem 1.5** is also proved.

This completes the proof of **Theorem 1**.

According to Theorem 1.2, after having clustered the last level L_{M_1-1} during the first phase, if two grid cells c_{ij}, c_{kl} belong to the same cluster, then T(id(c_{ij}))=T(id(c_{kl})). To complete the proof of correctness, we ony have to show that after the second clustering phase (the Tree Flattening Phase), if two grid cells c_{ij}, c_{kl} belong to the same cluster, then id(c_{ij})=id(c_{kl}). Namely, all grid cells in a cluster have the same ID.

This is indeed the case and we will show it by proving the following theorem.

Theorem 2. During the second phase (Tree Flattening) the following statements are true:

- **2.1** Suppose that we just completed processing level L_i during the second clustering phase. Then, if a cluster ID X belonged in ID(L_i) at the end of the first clustering phase, it is true that parent(X)=root(T(X)).Equivalently, X is now Spointing directly to the root of its tree.

- **2.2** Suppose that we just completed processing level L_i. Then, for any grid cell c_{ij} in L_i it is true that X=root(T(X)).

Proof of Theorem 2:

We will use induction to prove this theorem. Suppose that both statements are true for all $i \geq d+1$. Then we will show it is true for $i = d$. The second statement is obvious. In essence during the TreeFlattening phase, we set the id(c_{ij}) of each grid cell to be the root ID of the cluster that it belongs in. Since the cluster IDs in each tree don't change and trees don't merge during this phase, **Theorem 2.2** is proven.

To prove **Theorem 2.1**, just notice that as soon as we encounter an ID X in a grid cell c_{ij}, we set its parent to be the root of its tree. Since the Tree Flattening phase doesn't ever change the roots of a tree, it follows that **2.1** is true.

The only reason we care to prove **Theorem 2.1** is because we want to perform the Tree Flattening by only accessing two grid levels worth of cluster IDs at a time (since the *TreeStruct* may be too large to fit in its entirety in main memory). The implication of **2.1** is that when we are processing a grid level L_i during the Tree Flattening phase, the root of the tree of each id in the *CurrentLevel* can be definitely found in either *CurrentTreeList* or *PreviousTreeList*. This follows by combining Theorems **1.4** and **2.1**.

This concludes the proof of **Theorem 2** and thus we have proven the correctness of SingleClus. The reason is that due to Theorem 2.2, if two grid cells belonged to the same cluster, then they were assigned a cluster ID equal to the root of the Tree they belonged in after phase 1. Since the second phase does not merge trees and therefore does not affect the roots of the trees, Theorem 2 follows.

3.9 Performance Analysis

Here we will prove that our algorithm performs an optimal (linear) number of I/O operations and that it has a linear run-time in memory. We also show that we are always using a small amount of memory. We claim that this algorithm is optimal with regards to its I/O performance.

3.9.1 Memory Budget

It is obvious from the description of the algorithm that at each instant, the maximum amount of memory that we are utilizing is for storing $4 \times M_2 + M_1 + 2 \times MaxPoints$ integers. This is because we have 4 buffers of length at most M_2 (namely CurrentLevel,PreviousLevel,CurrentTreeList,PreviousTreeList) and one of length M_1 (namely OffsetIDList) and $MaxPoints$ is a constant defined by the user.

3.9.2 In-Memory Performance

We will show that the in-memory performance of the algorithm is linear in the number of data points and grid cells. In order to achieve this we will prove that, at any point during the two clustering phases, whenever we want to find the root R of the tree that a cluster ID X belongs in, we need to move at most 2 levels up towards the root. That is equivalent to say that X is always at most 2 steps away from its root *at the time instant that we are in need of finding its root.* We make an important note here regarding this claim: We don't state that the height of the trees in TreeStruct never grows beyond 2. We just mean that, from the way SingleClus is designed, *at the time instant* that we access a grid cell c_{ij} with id(c_{ij})=X, then X is at most 2 steps away from its root *at that time.*

To prove this claim we start with some Definitions and Lemmas.

Definition: We say that a set of *full* grid cells in a level L_i constitute a *contiguous section* S if and only if there exist j, k with $0 \leq j \leq k \leq M_2 - 1$ such that:

- $c_{i,h}$ is *full* for all $h = j, j+1, j+2, ..., k$
- $c_{i,j-1}, c_{i,k+1}$ are empty (if they exist based on the choice of j, k)

It is obvious from this definition and the definition of a cluster, that all the grid cells in a contiguous section belong to the same cluster. Hence, according to Theorems **1.2** and **1.3**, for all cells c_{ij} in a contiguous section S on level L_i, after the clustering of L_i during the first phase is complete, all the cells in a contiguous section S contain

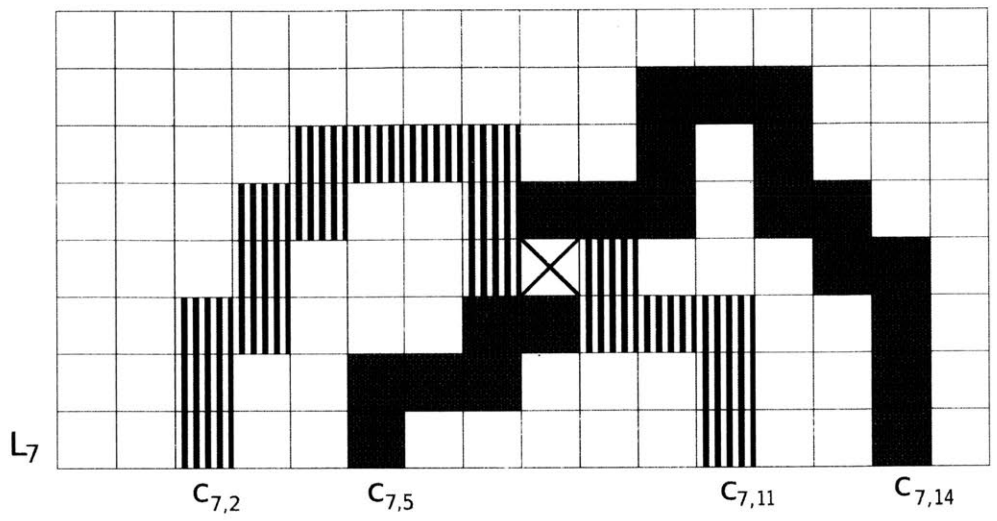

Figure 3-1: The two paths must intersect at some point. We have denoted this point to be the grid cell containing X. Hence, all the grid cells must belong to the same cluster.

the same cluster ID. We will denote by id(S) the (common) cluster ID of all grid cells in S after L_i has been clusterd during phase 1.

Lemma 1: After the end of clustering L_i during the first clustering phase, let $S_1, S_2, ..., S_k$ be the contiguous sections of L_i in row major order of appearance. Then there are no j, k, l, m with $j < k < l < m$ such that id(S_j)=id(S_l), id(S_k)=id(S_m), and id(S_j)≠id(S_k).

Proof of Lemma 1: Suppose that after the clustering of L_i there exist j, k, l, m with $j < k < l < m$ such that id(S_j)=id(S_l), id(S_k)=id(S_m) and id(S_j)≠id(S_k). We will show that we reach a contradiction. Indeed, since id(S_j)=id(S_l), we conclude that all the cells in $S_j \cup S_l$ belong to the same cluster. Therefore, if we choose two grid cells $c_{ij} \in S_j$ and $c_{il} \in S_l$ then there has to exist a path of full cells $c_1 = c_{ij}, c_2, ..., c_t = c_{il}$ such that $c_h \in$Grid[0:i] and c_h, c_{h+1} are neighbours for all $h = 1, 2, ..., t-1$. Similarly, if we choose two grid cells $c_{ik} \in S_k$ and $c_{im} \in S_m$ then there has to exist a path of full cells $c'_1 = c_{ik}, c'_2, ..., c'_s = c_{im}$ such that $c'_h \in$Grid[0:i] and c'_h, c'_{h+1} are neighbours for all $h = 1, 2, ..., s-1$.

But since $j < k < l < m$ this must mean that $c_{ij}, c_{ik}, c_{il}, c_{im}$ appear in this order (i.e. c_{ik} appears between c_{ij} and c_{il} etc). Hence, because the two paths are continuous (since they consist of neighbouring cells) we conclude that they have to intersect at some point (see Figure 3-1). Therefore, since the two paths intersect, we conclude that $c_{ij}, c_{ik}, c_{il}, c_{im}$ actually all belong to the same cluster. Therefore, after the clustering of L_i during the first phase, $\mathsf{id}(c_{ij})$=$\mathsf{id}(c_{ik})$=$\mathsf{id}(c_{il})$=$\mathsf{id}(c_{im})$. But this in turn means that $\mathsf{id}(S_j)$=$\mathsf{id}(S_k)$=$\mathsf{id}(S_l)$=$\mathsf{id}(S_m)$ which contradicts the assumption that $\mathrm{id}(S_j) \neq \mathrm{id}(S_k)$. Hence we have reached a contradiction. Therefore, **Lemma 1** is proven.

Lemma 2: After finishing clustering level L_i during the first clustering phase, let $X, Y \in \mathrm{ID}(L_i)$ such that $X < Y$. Let c_{ij_1}, c_{ij_2} be the first grid cells in row major order such that $\mathrm{id}(c_{ij_1})$=X and $\mathrm{id}(c_{ij_2})$=Y. Then it is true that $j_1 < j_2$. This is equivalent to saying that, if $X < Y$, then the first appearance of X in L_i is before the first appearance of Y.

Proof of Lemma 2: This Lemma can be proved by noticing the operation of **Stage 3** during clustering a level L_i. Specifically, each grid cell c_{ij} in L_i receives its final cluster ID for the first clustering phase during **Stage 3** of the clustering of this level. We will procede by induction and assume that Lemma 2 holds for all c_{ih} with $h < j - 1$.

While we process grid cell c_{ij} during **Stage 3**, we find R=$\mathsf{root}(\mathrm{T}(\mathsf{id}(c_{ij})))$. If $R \leq G$, then we increase *GlobalID* by 1 and we set $\mathsf{id}(c_{ij})$=$GlobalID$. Moreover we set $\mathsf{parent}(R)$=*GlobalID*. Since no other grid cell before c_{ij} can contain this new *GlobalID*, Lemma 2 holds. Alternatively, if $R > G$ then we set $\mathsf{id}(c_{ij})$=R. This means that cell c_{ij} will now contain a cluster ID R that was already assigned to a previous grid cell in that level during **Stage 3**. Hence, by the induction hypothesis, **Lemma 2** holds. This completes the proof.

Lemma 3: During **Stage 1** of clustering a level L_i, whenever we perform a root search for a cluster ID X, X is always at most 1 step away from the root of its tree.

Proof of Lemma 3: For the sake of contradiction, assume that during the first clustering phase of a level L_i, and during **Stage 1**, we will perform a root search for a cluster ID X such that X is two or more steps away from the root of its tree. There are two possible scenarios:

- The first scenario is that X is a cluster ID that did not appear in $\mathrm{ID}(L_{i-1})$. This means that X was created during **Stage 1** of clustering level L_i. When X is first created during this stage, it is obvious from the description of the algorithm that $\mathsf{parent}(X){=}X$. Therefore, at some point during **Stage 1**, a merge has to occur, so that the $\mathsf{parent}(X)$ is changed to some ID $Y \neq X$. By the description of the algorithm during **Stage 1**, it is cleat that if $\mathsf{parent}(X){=}Y$ then $Y < X$. We claim that we never encounter another grid cell with an ID equal to X during **Stage 1** of clustering L_i. The reason is that X can't appear on a grid cell on L_{i-1} (from the assumption of this scenario). In addition X can't appear again on L_i since it now has a parent ID that is different than X. Therefore, there is no way that a root search will take more than one step for an ID in the first scenario.

- In the second scenario, X is an ID in $\mathrm{ID}(L_{i-1})$. Therefore, X was created during **Stage 3** of clustering L_{i-1}. We have already mentioned that, at the end of clustering L_{i-1} during the first phase, for each $X \in \mathrm{ID}(L_{i-1})$, it is true that $\mathsf{parent}(X){=}X$. Therefore, if X is at some point two steps away from the root of $\mathrm{T}(X)$, this could have only happened if there exist indices $j_1 < j_2 < j_3$ such that:

  - During clustering of grid cell c_{i,j_1} during **Stage 1**, it is true that $\mathsf{id}(c_{i-1,j_1}){=}X$ and $\mathsf{id}(c_{i,j_1-1}){=}Y$, where $\mathsf{root}(\mathrm{T}(Y)){=}R$ and $R < X$. Then, after this grid cell has been proccessed, $\mathsf{parent}(X){=}R$. The reason why such a j_1 exists is because, at some point, X will have to change its parent for the first time after the clustering of L_{i-1}. The index j_1 is the first index that this happens during **Stage 1** of clustering L_i.

  - During clustering of grid cell c_{i,j_2} during **Stage 1**, it is true that one of the following two options are true:

    * $\mathsf{root}(\mathrm{T}(\mathrm{id}(c_{i-1,j_2}))){=}R$ and $\mathsf{root}(\mathrm{T}(\mathrm{id}(c_{i,j_2-1}))){=}R'$, where $R' < R$.
    * $\mathsf{root}(\mathrm{T}(\mathrm{id}(c_{i-1,j_2}))){=}R'$ and $\mathsf{root}(\mathrm{T}(\mathrm{id}(c_{i,j_2-1}))){=}R$, where $R' < R$.

After c_{i,j_2} has been proccessed during **Stage 1**, it is true that parent(X)=R and parent(R)=R'. The reason why such an index j_2 has to exist is because we assumed that at some point of clustering L_i during **Stage 1**, X will be at least two steps away from its root. Then we choose j_2 to be the first index at which X is two steps away from its root. Obviously, $j_1 < j_2$.

- Finally, during the clustering of grid cell c_{i,j_3} during **Stage 1**, it is true that id(c_{i-1,j_3})=X and therefore we will be forced to perform a root search for an ID that is at least two steps away from its root. The reason why such an index j_3 exists, is because we assumed that we will actually have to perform a root search for X after its distance from its root has become greater than 1. The reason why X appears on a grid cell c_{i-1,j_3} of the previous level and not on the current level is the same exact reasoning that we provided in evaluating the first scenario of this Lemma. Then, we just let j_3 be the smallest such index. Again, it is obvious that $j_2 < j_3$.

Since $R < X$ and $X \in$ID(L_{i-1}) we conclude that $R \in$ID(L_{i-1}) as well. Moreover, since $R < X$, according to **Lemma 2**, the first appearance of R in L_{i-1} has to be before the first appearance of X. But the latest X appears in L_{i-1} is on position j_1. Hence R appears in some position $j_0 < j_1$ in L_{i-1}. But, according to the above discussion, X also appears on position j_3. There are two cases as we described above:

- **Case 1:** root(T(id(c_{i-1,j_2})))=R and root(T(id(c_{i,j_2-1})))=R', where $R' < R$. Then R appears on position j_2 in L_{i-1}. Therefore R appears on positions j_0, j_2 and X appears on positions j_1, j_3 with $j_0 < j_1 < j_2 < j_3$. But, according to **Lemma 1** this is impossible. Hence we have reached a contradction and, therefore, **Lemma 3** is proved.

- **Case 2:** root(T(id(c_{i-1,j_2})))=R' and root(T(id(c_{i,j_2-1})))=R, where $R' < R$. This case is impossible for the same reasons discussed in scenario 1 of this Lemma.

This concludes the proof of **Lemma 3.**

Lemma 4: During **Stage 2** of clustering a level L_i, whenever we perform a root search for a cluster ID X, X is always at most two steps away from the root of its tree.

Proof of Lemma 4: Let c_{ik} be the last grid cell that we have processed during **Stage 2** of clustering level L_i. We will prove that Lemma 4 is true using induction on k. The induction hypothesis is the following. Assume that for $j = M_2 - 1$ down to $j = k + 1$, during **Stage 2** of clustering a level L_i, it was true that X=id(c_{ij}) after then end of **Stage 1**. Then, right before we start processing c_{ik} during **Stage 2**, it is true that parent(X)=root(T(X)). Equivalently, this is saying that all the IDs that appeared in positions $k + 1$ through $M_2 - 1$ on L_i at the end of **Stage 1** are now at most one step away from their roots.

We will prove that the induction claim holds immediately after we have processed c_{ik}. Let X=id(c_{ik}) immediately after the end of **Stage 1**. If after **Stage 1** X appeared in some position on L_i between positions $k + 1$ and $M_2 - 1$ then we are done because of the induction hypothesis. Alternatively, the last position that X appeared on in L_i was at position k. But then, during **Stage 1**, X must still have been the root of its tree right before c_{ik} was processed.

Let h be the first position where X was no longer the root of its tree during **Stage 1**. Then it is true that $h > k$. In addition, when we processed c_{ih}, parent(X) became equal to some ID Y and id(c_{ih}) was set equal to Y. Therefore, by the induction hypothesis, since $h > k$, Y was at most one step away from its root at the time of processing c_{ik} during **Stage 2**. This would make X be at most two steps away from its root while processing c_{ik} during **Stage 2**. Moreover, after having processed c_{ik} during **Stage 2**, X's parent will now be equal to root(T(X)). Therefore, the induction hypothesis is still true. Meanwhile, we have also proved that at no point during **Stage 2** will we have to search the root of and ID X that is more than two steps away from rootT(X). Hence, **Lemma 4** is proved.

Lemma 5: During **Stage 3** of clustering a level L_i, whenever we perform a root search for a cluster ID X, it is true that X is always at most one step away from the root of its tree.

Proof of Lemma 5: This is obvious from the operation of **Stage 3**. At the end of **Stage 2**, every cluster ID that appears on a grid cell on level L_i, is the root of its tree. During **Stage 3**, these roots change at most once and the new roots are never modified. Hence, **Lemma 5** is proven.

Lemma 6: During the second clustering phase (Tree Flattening), whenever we perform a root search for a cluster ID X, it is true that X is always at most two steps away from the root of its tree.

Proof of Lemma 6: This is also a trivial thing to prove using Theorem 1 and the above Lemmas. Specifically, for any ID X in $\mathrm{ID}(L_{i-1})$, after **Stage 2** of clustering L_i during the first clustering phase, it is true that $\mathsf{parent}(X)$=$\mathsf{root}(\mathrm{T}(X)$. After, **Stage 2** of clustering L_i, in the worst case, X is now one step away from the root of its tree. Therefore after **Stage 3** of clustering L_i, in the worst case, X is now two steps away from the root of its tree. Therefore, during the second clustering phase, X is at most two steps away from the root of its tree.

Moreover, using induction, we claim that from $j = M_1 - 1$ down to $j = k + 1$, all cluster IDs that appeared in a grid cell in levels L_{k+1} through L_{M_1-1} are now one step away from their root. Using the result of the previous paragraph this is also true for $j = k$ after we have processed L_k during the second clustering phase. Hence **Lemma 6** is proved.

Using the above Lemmas, we proceed to find the in-memory runtime of SingleClus. Specifically, during the first clustering phase, we process each grid cell exactly three times (once per each Stage). Each time, we perform at most 2 root searches that take at most O(1) time. Moreover, we proccess each cluster ID at most three times. The reason is that after a cluster ID X is created during clustering L_i, we may only process it once per Stage to change $\mathsf{parent}(X)$.

In the second clustering phase, we process each grid cell in memory exactly once. Moreover, we perform exactly one root search that takes O(1) time. Similarly, for cluster IDs, we change the parent of each cluster ID at most once during the second clustering phase.

Combining these two facts we see that the **in-memory runtime** of the algorithm is O(size(InputArray)+size(Grid)+size(TreeStruct))=O(size(InputArray)+size(Grid)). Therefore SingleClus is operating on time linear to both the input array size and the grid size.

3.9.3 I/O cost

We proceed to determine the number of I/O operations that our algorithm performs.

To determine the I/O cost we just need to observe the following:

- During the first clustering phase (Raster Scan) we read into memory each point in InputArray exactly once.
- During the first clustering phase, we write the contents of each grid cell into Grid in the disk exactly once.
- During the first clustering phase, we write the contents of each created cluster ID into TreeStruct in the disk exactly once.
- During the second clustering phase, we read and write each grid cell in Grid exactly once.
- During the second clustering phase, we read and write each grid cell in TreeStruct exactly once.

Moreover, as we have already mentioned, the size of TreeStruct is always smaller than that of Grid. Hence, the total number of I/O operations that we have to perform is O(size(InputArray)+size(Grid)+size(TreeStruct))=O(size(InputAray)+size(Grid)). Finally, it is important to note that we perform the *theoretically optimal* number of disk reads for InputArray since we have to read each data point at least once. The same is true for Grid and TreeStruct. Specifically, we write each element in those structures in disk exactly once during the first phase (when these structures are being created).

Since we require fast queries, and hence a second tree flattening phase is neccessary, we read and write each element in those structures in disk exactly once more.

This proves that our algorithm is *theoretically optimal* in the number of I/O operations it performs.

3.10 Cluster Membership Query

In this section we just describe the basic query after we have finsihed clustering the dataset.

3.10.1 Description of Query

The basic cluster membership query takes as input two data points $p_i = (p_{i,1}, p_{i,2})$ and $p_j = (p_{j,1}, p_{j,2})$ and determines whether or not they belong to the same cluster. We do this as follows:

For each point p_i, first we determine which grid cell they belong in. We do this using findPointCell function. After we have found the coordinates of the grid cell c_{ij} that $p_i = (p_{i,1}, p_{i,2})$ belongs in, we bring id(c_{kl}) in memory from Grid by reading the contents of Grid at the offset of c_{kl}. For a cell c_{kl} its byte offset will be $i \times M_2 + j \times$sizeof(integer). This completes this step and we now have the cluster ID of the first point (potentially 0 if it wasn't contained in a grid cell that was part of a cluster). We repeat the proccess for p_j .

The code for the cluster membership query is as follows:

```
def findCluster(pointCoords):
  c_1,c_2=findPointCell(pointCoords)
  offset=c_1*M_2+c_2
  f = open("Grid.txt",'r')
  ID=f.seek(offset*sys.getsizeof(int))
  return ID
```

The above function finds the cluster ID of a data point. We assume that clustering has completed and the file Grid is saved in our current directory. sizeof is

```
def sameCluster(pointCoords1,pointCoords2):
    id1=findCluster(pointCoords1)
    id2=findCluster(pointCoords2)
  return ((id1==id2) and (id1!=0))
```

Chapter 4

Qualitative Evaluation

In order to evaluate the quality of our clustering algorithm, we implemented SingleClus and ran experiments to determine the types of datasets that we can use SingleClus on and get qualitatively good clusters. In this chapter, we propose a basic qualitative comparison metric between SingleClus and DBscan, and we discuss our results and methodology.

4.1 Choice of Datasets

We ran our algorithm on two synthetic datasets that were introduced in ([14],[15]). The datasets are shown in Figures 4-2 and 4-3.The first dataset consists of 3000 data points organized in 20 clusters of approximately equal size and shape. The clusters have a spherical shape. The second dataset consists of 8000 data points organized in random shaped clusters that are not equally sized. Moreover the second dataset contains points that are just noise and do not belong to any cluster.

The reasons we decided to use these datasets are the following:

- The first dataset, shown in Figure 4-2, contains clusters that are of approximately equal shape and size. Moreover, the clusters are close to each other, meaning that the boundaries of two clusters can be hard to discern. Therefore, running our algorithm on this dataset is important because we can determine whether SingleClus can effectively discern between clusters that are very close to each other. In practice, many applications were we must cluster very large spatial datasets will involve sparse and skewed data. For example, the locations of ships in the oceans will naturally create clusters around ports. These clusters will be sufficiently far from each other that their boundaries will be easy to discern. However, we chose this dataset to evaluate our algorithm's performance in extreme cases. Our goal is to determine whether SingleClus can be utilized in datsets where the cluster boundaries are very close to each other.

- The second dataset, shown in Figure 4-3 contains clusters that don't have the same shape. Moreover, the dataset contains points that do not belong in any cluster and, hence, they are just noise. Therefore, running our algorithm on this dataset is important because we can determine whether SingleClus can effectively find clusters of various shapes, while at the same time disregarding data points that are just noise.

We expect SingleClus to perform better in the second dataset than in the first dataset. The reason is that the grid-based approach of SingleClus does not take into account the relative densities of regions inside clusters. We explain this using Figure 4-1. In this scenario, all the data points of the one grid cell are placed in the top left part of the cell, whereas all the data points of the second grid cell are placed in the lower right part of the cell. However, since there are enough points in both cells, SingleClus will consider that the cells are full and will place all of their contained points in the same cluster. DBscan on the other hand is not operating relative to a grid. Therefore, using a reasonable *Eps* parameter, it will place the two sets of points in different clusters.

4.2 Choosing Parameters

We experimented with various combinations of $CellSide$ and T_h. To evaluate the performance of our algorithm we decided to compare it against DBscan. An important

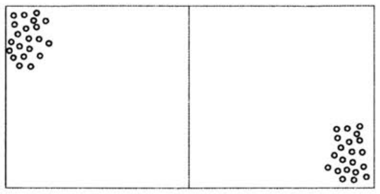

Figure 4-1: This is a scenario where SingleClus would place all the points belong to the same cluster, whereas DBscan would place them in different clusters.

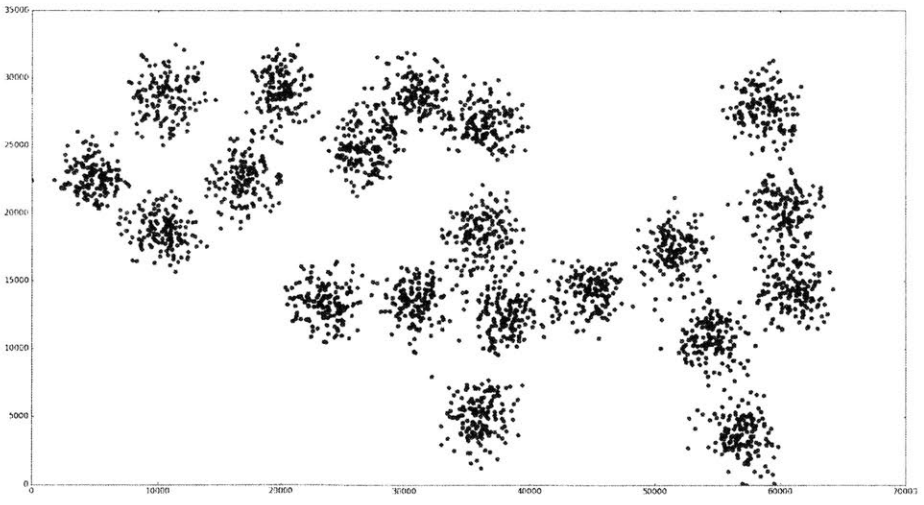

Figure 4-2: First dataset

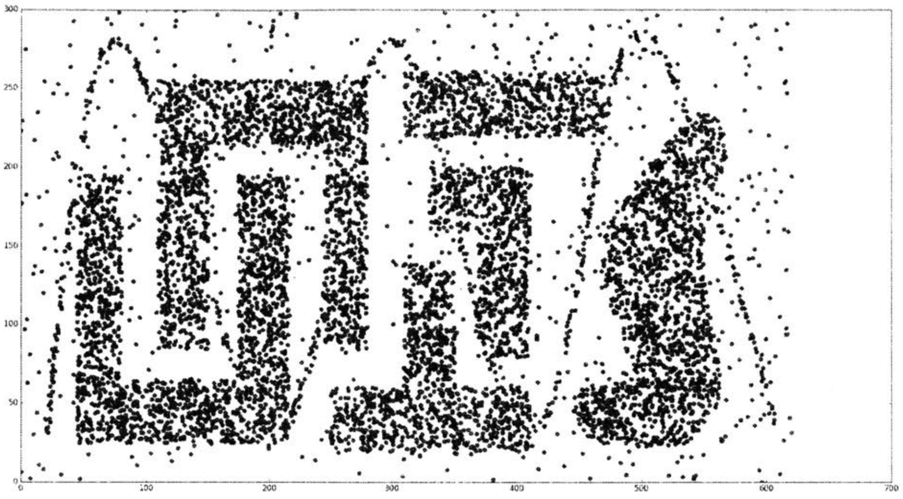

Figure 4-3: Second dataset

observation to make is that the shape of clusters that SingleClus produces is exactly the same as the ones produced by Hoshen-Kopelman. Our difference lies in the theoretical optimality results that were stated and proved in Chapter 3. Hence, what we are trying to achieve here is to propose a basic qualitative measure to compare the grid based approach of SingleClus and Hoshen-Kopelman with that of DBscan. We propose that the quality of clustering is to be measured as follows:

- DBscan uses $MinPts$ to denote the number of data points that must exist within a sphere of radius Eps, centered at some data point p, in order for p to be part of a cluster. SingleClus, on the other hand, uses T_h to determine how many points have to fall within a cell of side $CellSide$, in order for the cell to be part of a cluster. For quality analysis, we make the assumption that all the points inside a cluster are uniformly distributed. Therefore, if we set $MinPts = T_h$ then the equivalent areas that the points must be contained in have to be equal. Hence, we must set $\pi Eps^2 = CellSide^2$ or, equivalently, $CellSide = \sqrt{\pi} Eps$. Since $\sqrt{\pi}$ is approximately equal to 1.77, for any given combination of $(MinPts, Eps)$ in DBscan, we ran SingleClus with $T_h = MinPts$ and $CellSide = 1.77 \times Eps$

- For each pair of $(MinPts, Eps)$ and $(T_h, CellSide)$ that satisfy the relation mentioned above, we determine the quality of SingleClus using the number of discovered clusters as a metric. That is, we determine how many clusters DB-Scan discovers and compare that number to how many clusters SingleClus discovers. There is the possibility that, based on our choice of parameters, DBscan or SingleClus may discover more or less clusters than there exist in our datasets. In this scenario, we only care about the discovered clusters that we knew *a priori* that they were clusters. Suppose SingleClus finds 20 clusters in the first dataset, but does so by merging two different clusters into one and by dividing a single cluster into two clusters. In this scenario, we will consider that SingleClus found 19 of the clusters we were searching for. We will consider that it completely missed the smallest of the two clusters it merged. Moreover, out of the 2 clusters that SingleClus found by dividing the single cluster, we will only consider as valid cluster the one cluster that has the greatest overlap with the original cluster.

4.3 Results

We ran SingleClus and DBScan on the two datasets using an already existing implementation of DBScan ([13]). We present some of our results.

Let us describe our results. In Figures 4-4 and 4-5 we ran SingleClus and DBscan on the first dataset. The parameters we chose are $MinPts = T_h = 4$, $Eps = 600$, and $CellSide = 1080 = 1.8 \times Eps$. With these parameters, DBscan finds 19 clusters and SingleClus finds 17. In the next two figures, we used $MinPts = T_h = 5$, $Eps = 700$, and $CellSide = 1260 = 1.8 \times Eps$. Here, DBscan finds 19 clusters and SingleClus finds 16. Therefore, as we predicted, SingleClus is not well suited for datasets where the distance between clusters is of the same order as the diameter of the clusters.

In Figures 4-8 and 4-8 we ran DBscan and SingleClus on the second dataset. The parameters we chose are $MinPts = T_h = 12$, $Eps = 8$, and $CellSide = 13 = 1.64 \times Eps$. With these parameters, both DBscan and SingleClus find all 6 clusters. In the

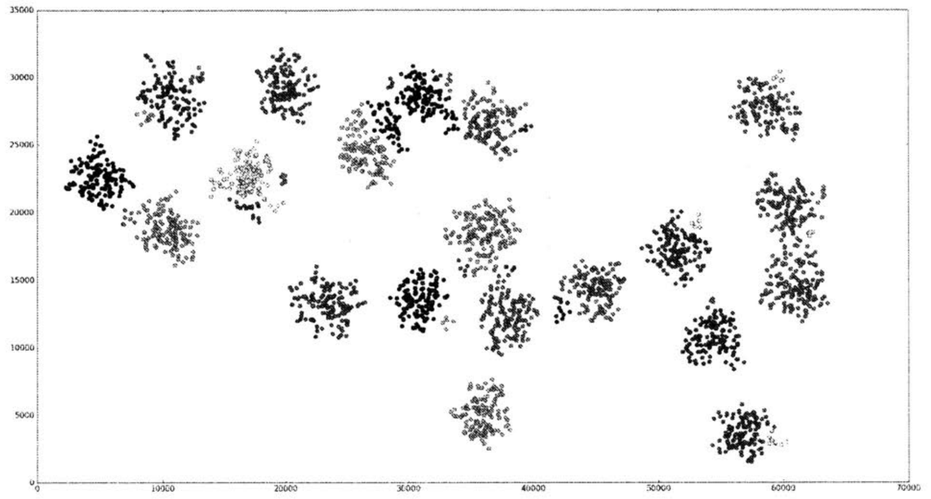

Figure 4-4: DBscan on the first dataset with $Eps = 600$ and $MinPts = 4$

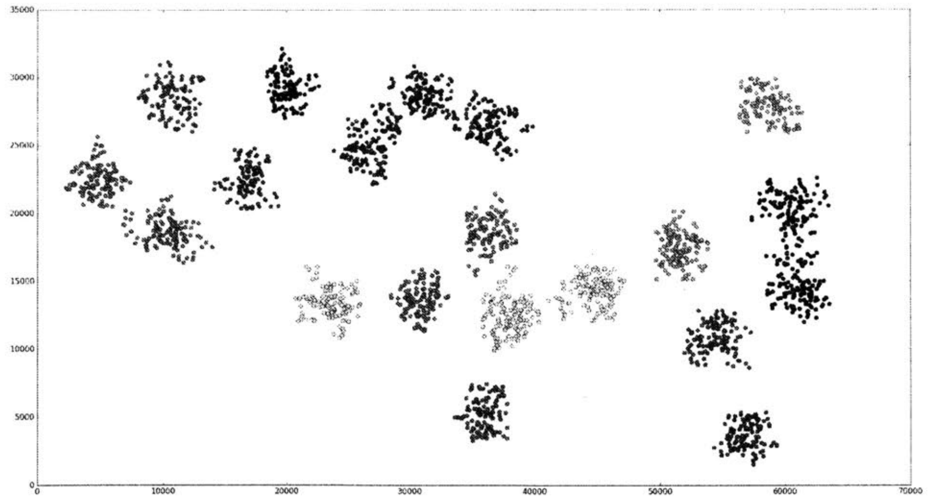

Figure 4-5: SingleClus on the first dataset with $CellSide = 1080$ and $T_h = 4$

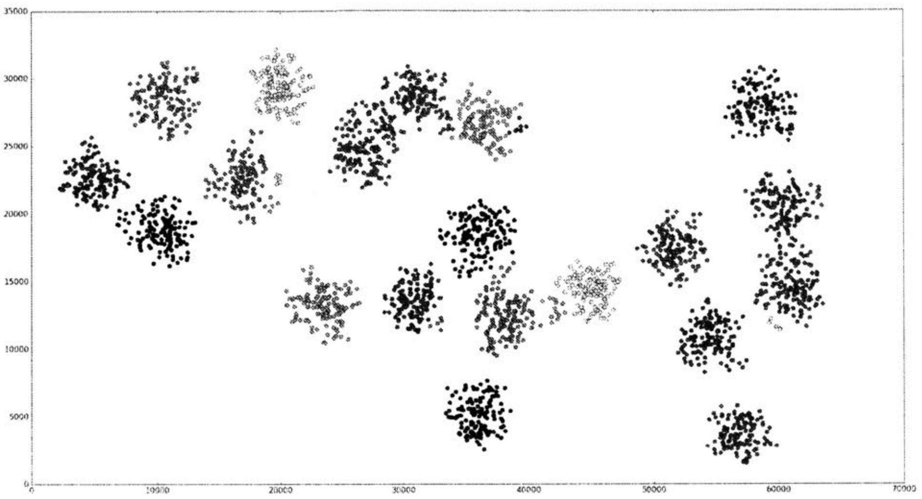

Figure 4-6: DBscan on the first dataset with $Eps = 700$ and $MinPts = 5$

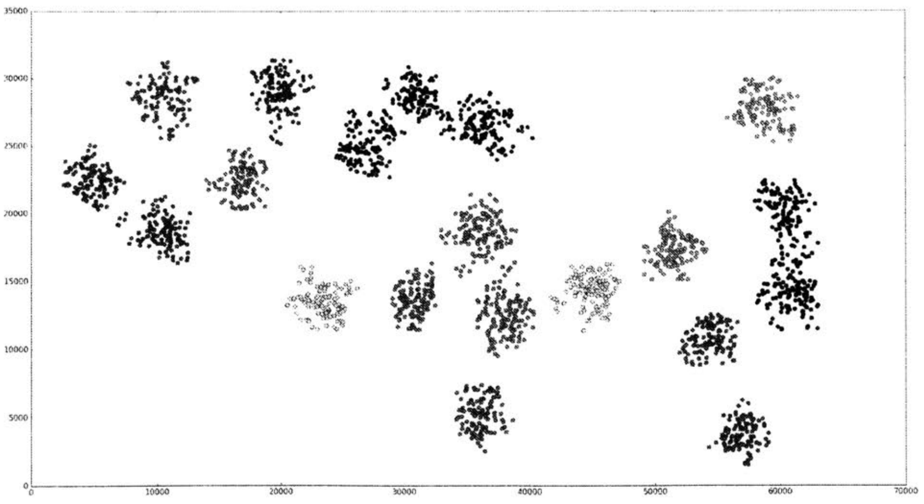

Figure 4-7: SingleClus on the first dataset with $CellSide = 1260$ and $T_h = 5$

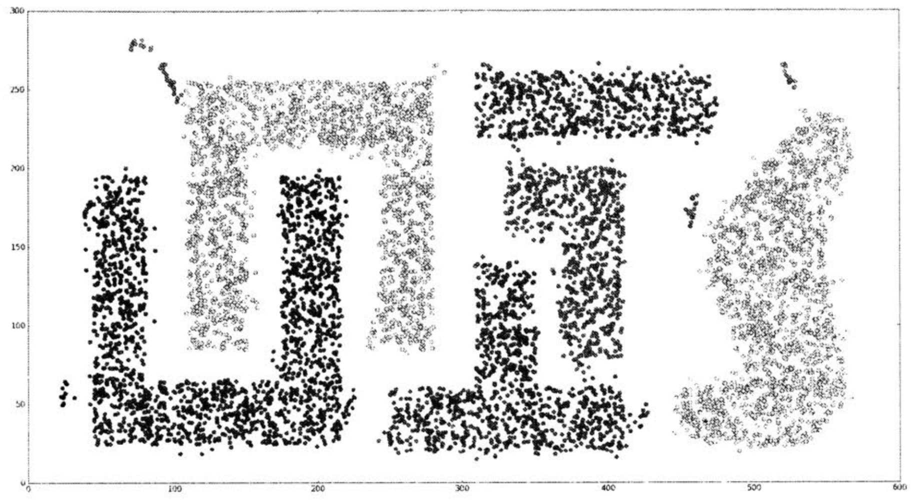

Figure 4-8: DBscan on the second dataset with $Eps = 8$ and $MinPts = 12$

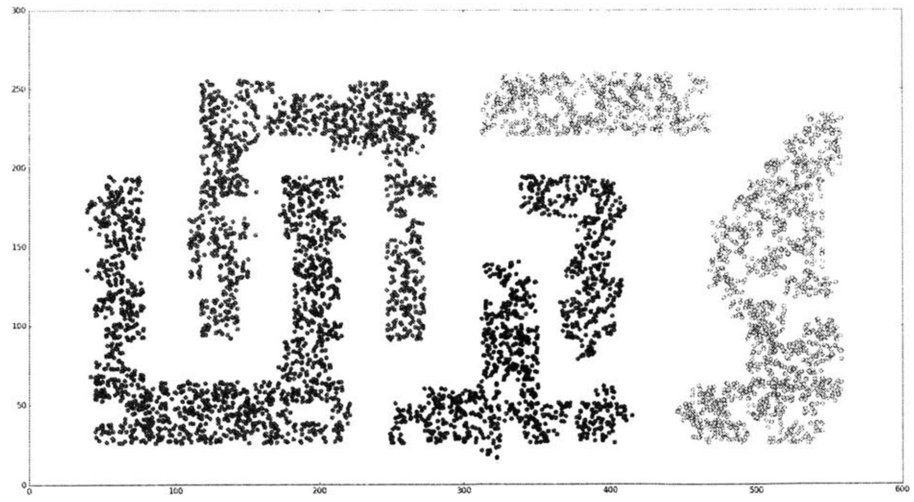

Figure 4-9: SingleClus on the second dataset with $CellSide = 13$ and $T_h = 12$

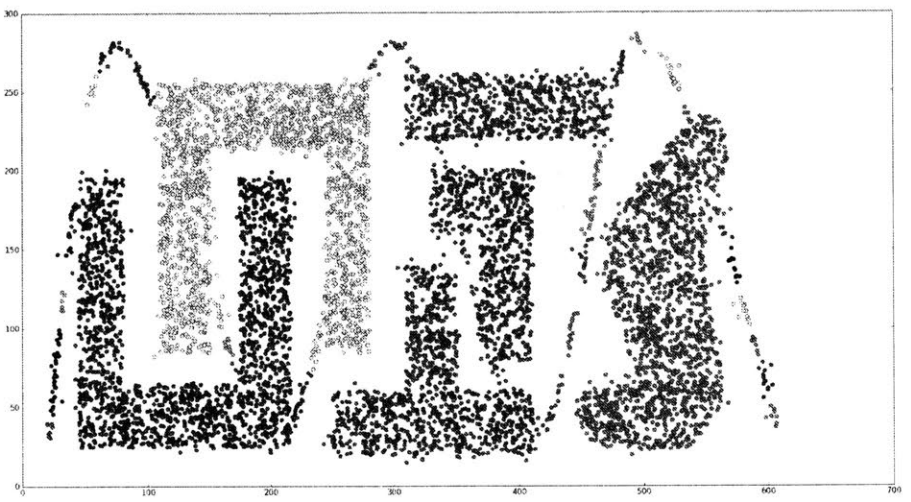

Figure 4-10: DBscan on the second dataset with $Eps = 7$ and $MinPts = 6$

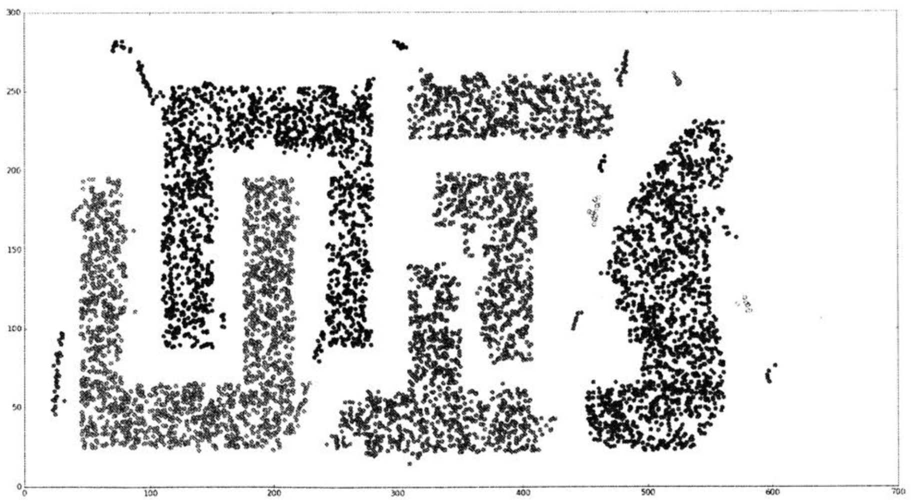

Figure 4-11: SingleClus on the second dataset with $CellSide = 11$ and $T_h = 6$

next two figures, we used $MinPts = T_h = 6$, $Eps = 7$, and $CellSide = 11 = 1.8 \times Eps$. Here, DBscan finds only 5 clusters whereas SingleClus finds all 6. As we predicted, SingleClus performs better in the second dataset than in the first dataset in terms of cluster discovery. Therefore, it is reasonable to claim that SingleClus (and Hoshen-Kopelman by extension) are better suited to clustering sparse and skewed arrays, or datasets where the cluster boundaries are easily discerned.

Chapter 5

Discussion and Future Work

In this section we discuss potential improvements on our algorithm and new directions on clustering Big Data.

5.1 Compression and Minimizing Disk Space

One thing that we mentioned while we were discussing SingleClus in Chapter 3, was that **every** grid cell c_{ij} is saved in Disk structure Grid. In the HK setting, we are already given the entire 0-1 grid. However, as we mentioned in Chapter 2, we are clustering points and the grid is created to facilitate this clustering. Therefore, if we can avoid saving the empty grid cells, it will be an improvement both in disk usage and on clustering performance.

If our dataset is very sparse and skewed, then a great proportion of the grid cells will not contain at least T_h points and hence will contain a cluster ID of 0 (indicating that they are empty). This is bad both for storage purposes (we are storing more grid cells then we potentially need) and slows down the tree flattening step of the clustering algorithm since we have to process more grid cells.

There are 2 ways to solve this problem:

- **Change Algorithm Design** The most obvious solution is to just change the algorithm design so that we don't save some or all of the empty grid cells in disk. However this is not a trivial challenge. In our algorithm, and especially in the tree flattening phase, we rely on the fact that we have saved all the grid cells in row major order. Without this fact, we have to perform searches to bring into memory the relevant portions of the grid. The same is true for querying the data. We have depended on the fact that we know exactly what the offset of a grid cell is in the grid so that we can read it into main memory with a single read from disk. If we don't know the offset of each cell then we have to save some index structure which will allow is to know where each grid cell is in the disk. There may not be a trivial change in the algorithm design that will solve this problem.

- **Compression** The alternative is to use some sort of compression algorithm that will take advantage of empty grid sections. However, again it is not trivial to perform queries on the compressed grid data.

Therefore, it becomes obvious that the problem of storing empty grid cells is not trivial and it is an interesting problem to solve.

5.2 Clustering for general k

5.2.1 The Choice of 2-Dimensions

We have decided to focus on the 2 dimensional case instead of the general k-dimensional problem for the following reasons:

- **Visualization** Unlike higher dimensions, clustering in 2 (and 3) dimensions allows us to inspect our clusters and determine their quality. When dealing with higher dimension ($k \geq 4$) this can only be achieved by projecting the clusters into a lower dimensionality (usually $k = 2$ or $k = 3$) and then perform the visualization in those dimensions.

- **Logical Clusters** When clustering (using the HK model) in 2 dimensions we have the guaraneee that our cluster contains cells that are all in the same plane.

However, when we increase dimensions, we may end up with clusters that extend towards multiple dimensions and hence do not necessarily make intuitive sense.

- 2^k **passes adding complexity to design**. The most important reason why we chose $k = 2$ is because it facilitates the design and the analysis of the correctness of the algorithm. It appears that, to follow the same algorithm design for a higher dimension k, we would first have to cluster completely every $(k - 1)$-dimensional level both during the Raster Scan and during the Tree Flattening phases. Therefore there will be twice as many accesses to each grid cell (and consequently twice as many I/O operations) as there were for solving the problem in $k - 1$ dimensions. This is obviously an exponential increase in the number of disk acceses and in-memory performance as well. Even for moderately small k, the factor 2^k adds a non signifiant factor in the performance of our algorithm and it can't be claimed that it is optimal anymore since the correctness proofs of Chapter 3 were all based on the fact that we operate in 2 dimensions.

Therefore, the problem of modifying our algorithm to cluster multi-dimensional data is an open one and would turn out to be a challenging future project.

5.3 Parallel Version

There have been algorithms that solve the problem of parallelizing HK ([10]). Even though they have produced good results, it remains to be seen if our algorithm can be modified so that it can outperform these algorithms.

5.4 Choosing T_h and CellSide

We mentioned in the previous chapter that T_h and *CellSide* are chosen heuristically by the user. We suggested that unless we start processing the data, there is no way to determine *a priori* what a good value would be for these two parameters. The fact that we are operating on very large datasets that don't fit in memory makes it even harder to solve this problem (say by randomly and uniformly sampling a few points to determing the dataset's properties). Therefore, an interesting problem is to determine whether we can somehow choose T_h and *CellSide* efficiently in the Big Data setting.

Appendix A

Pseudo-Code of SingleClus

In this Appendix we present the pseudo-code of SingleClus. We have utilized the functions that we presented in Chapter 3.

A.1 Code

This section provides the pseudo-code of the algorithm.

The main procedure is as follows:

```
def cluster(InputArray,CellSide,T_h,Min_1,Max_1,Min_2,Max_2):
  M1=(Max_1-Min_1)/CellSide+1
  M2=(Max_2-Min_2)/CellSide+1
  CurrentTreeIndex=0
  PreviousTreeIndex=0
  GlobalID=0
  PtsRead=0
  Level=0
  CurrentLevel=[]
  PreviousLevel=[]
  CurrentTreeList=[]
  PreviousTreeList=[]
  OffsetIDList=[]
  PointsBuffer=[]
  //First Phase - Raster Scan
```

```
while PtsRead<size(InputArray):
  BointsBuffer=read(InputArray, start=PtsRead, end=PtsRead+MaxPoints)
  PtsRead+=MaxPoints
  for point in PointsBuffer: //point=[p_x,p_y]
    TempLevel=findLevel(point)
    if TempLevel==Level: //update count in CurrentLevel
      index=convertCoordsId(findPointCell(point))
      CurrentLevel[index]+=1
    elif TempLevel!=Level: //start clustering current level
      CurrentLevelCount=0
      //Stage 1
      for i in range(M2):
        if CurrentLevel[i]>=T_h: //full cell
          if Level==0:
            if i==0:
              GlobalID+=1
              CurrentLevel[i]=GlobalID
              CurrentTreeList[CurrentLevelCount]=GlobalID
              CurrentLevelCount+=1
            else:
              if CurrentLevel[i-1]!=0:
                CurrentLevel[i]=CurrentLevel[i-1]
              else:
                GlobalID+=1
                CurrentLevel[i]=GlobalID
                CurrentTreeList[CurrentLevelCount]=GlobalID
                CurrentLevelCount+=1
          else:
            if i==0:
              if PreviousLevel[i]!=0:
                CurrentLevel[i]=PreviousLevel[i]
              else:
                GlobalID+=1
                CurrentLevel[i]=GlobalID
                CurrentTreeList[CurrentLevelCount]=GlobalID
                CurrentLevelCount+=1
      else:
        X=PreviousLevel[i]
        Y=CurrentLevel[i-1]
        if (X==0 and Y==0):
          GlobalID+=1
          CurrentLevel[i]=GlobalID
          CurrentTreeList[CurrentLevelCount]=GlobalID
          CurrentLevelCount+=1
        elif (X==0 and Y!=0):
          CurrentLevel[i]=Y
        elif (X!=0 and Y==0):
          R=findRoot(X)
          CurrentLevel[i]=R
        elif (X!=0 and Y!=0):
          R1=findRoot(X)
          R2=findRoot(Y)
          if R1<R2:
            CurrentLevel[i]=R1
            setParent(R2,R1)
          elif R2<R1:
            CurrentLevel[i]=R2
            setParent(R1,R2)
          else: //R1=R2
            CurrentLevel[i]=R1
  else: //empty cell
    CurrentLevel[i]=0
//Stage 2
for i in range(M2):
  X=CurrentLevel[M2-i-1]
  if X!=0:
    R=findRoot(X)
    CurrentLevel[M2-i-1]=R
    setParent(X,R)
//Stage 3
G=GlobalID
for i in range(M2):
```

```
            X=CurrentLevel[i]
            R=findRoot(X)
            if R<=G:
                GlobalID+=1
                CurrentLevel[i]=GlobalID
                CurrentTreeList[CurrentLevelCount]=GlobalID
                CurrentLevelCount+=1
                setParent(R,GlobalID)
            elif R>G:
                CurrentLevel[i]=R
        //Flush Data to Disk
        if Level>0:
            write(PreviousLevel,Grid)
            write(PreviousTreeList,TreeStruct,start=0,end=
                CurrentLevelCount)
        OffsetIDList[Level]=OffsetIDList[Level-1]+CurrentLevelCount
        PreviousTreeIndex=CurrentTreeIndex
        CurrentTreeIndex+=CurrentLevelCount
        PreviousLevel=CurrentLevel
        CurrentLevel=zeros(M2)
        PreviousTreeList=CurrentTreeList
        CurrentTreeList=zeros(M2)
        //Determine Empty Levels
        EmptyLevels=TempLevel-Level-1
        for i in range(EmptyLevels):
            Level+=1
            write(zeros(M2),Grid)
            OffsetIDList[Level]=OffsetIDList[Level+1] \\this indicates
                that the entire level is empty
        CurrentLevelCount=0
        Level=TempLevel
//Second Phase - Tree Flattening
CurrentLevel=[]
PreviousLevel=[]
CurrentTreeList=[]
CurrentTreeIndex=OffsetIDList[M1-1]
```

```
PreviousTreeIndex=GlobalID+1
for i in range(M1):
  Level=M1-1-i
  if CurrentTreeIndex=OffsetIDList[Level][0]==-1:
    continue
  CurrentLevel=read(Grid, start=Level*M2, end=(Level+1)*M2)
  CurrentTreeList=read(TreeStruct, start=OffsetIDList[Level], end=
      OffsetIDList[Level+1])
  CurrentTreeIndex=OffsetIDList[Level]
  for i in range(M2):
    X=CurrentLevel[i]
    R=findRoot(X)
    CurrentLevel[i]=R
    setParent(X,R)
  PreviousTreeList=CurrentTreeList
  PreviousTreeIndex=CurrentTreeIndex
  write(CurrentLevel, Grid, start=Level*M2, end=(Level+1)*M2)
```

Bibliography

[1] M. Ester, H.P. Kriegel, J. Sander, and X. Xu, "A density-based algorithm for discovering clusters in large spatial databases with noise", in *Proceedings of 2nd International Conference on Knowledge Discovery and Data Mining (KDD)*, AAAI Press, 1996, pp. 226-231

[2] J.B. MacQueen, "Some Methods for classification and Analysis of Multivariate Observations", in *Proceedings of 5th Berkeley Symposium on Mathematical Statistics and Probability*, University of California Press, 1967, pp. 281-297

[3] A. Aggarwal, J.S. Vitter, "The input/output complexity of sorting and related problems", in *Communications of the ACM, v.31 n.9*, 1988, pp. 1116-1127

[4] Y. Tao, J. Gan, "DBSCAN Revisited: Mis-Claim, Un-Fixability, and Approximation", in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, 2015, pp. 519-530

[5] R.E. Tarjan, J.V. Leeuwen, "Worst-Case Analysis of Set Union Algorithms", in *Journal of the ACM, v.31 n.2*, 1984, pp. 245-281

[6] S.P. Lloyd, "Least squares quantization in PCM", in *IEEE Transactions on Information Theory, 28*, 1982, pp. 129-137

[7] R. Sibson, "SLINK: an optimally efficient algorithm for the single-link cluster method", *The Computer Journal. British Computer Society. 16*, 1973, pp. 30-34

[8] C. Zhao, J. Song, "GDILC: A grid-based density isoline clustering algorithm", in *Proceedings of the Internet Conf. on Info-Net*, IEEE Press, Beijing, 2001, pp. 140-145

[9] J. Hoshen, R. Kopelman, "Percolation and Cluster Distribution. I. Cluster multiple labeling technique and critical concentration algorithm", in *Physical review. B, Condensed matter*, 1976

[10] J.M. Teuler, J.C. Gimel, "A Direct Parallel Implementation of the Hoshen-Kopelman Algorithm for Distributed Memory Architecture", in *Computer Physics Communications 130*, 2000, pp. 118-129

[11] M. Flanigan, P. Tamayo, "Parallel Cluster Labeling for Large-Scale Monte Carlo Simulations", in *Physica A 215*, 1995, pp. 461-480

[12] National Oceanic and Atmospheric Administration. Marine Cadastre. *http://marinecadastre.gov/ais/*

[13] scikit-learn library of data mining and data analysis tools. *http://scikit-learn.org/stable/modules/clustering.html*

[14] I. Karkkainen, P. Fra nti, "Dynamic local search algorithm for the clustering problem", in *University of Joensuu Research Report A-2002-6*, 2002

[15] G. Karypis, E.H. Han, V. Kumar, "CHAMELEON: A hierarchical 765 clustering algorithm using dynamic modeling", in *IEEE Trans. on Computers, 32 (8)*, 1999, pp. 68-75