UNLV | University Libraries
University of Nevada, Las Vegas

UNLV Theses, Dissertations, Professional Papers, and Capstones

5-1-2015

# Efficient Estimation of Cluster Population

Sanjeev K C
*University of Nevada, Las Vegas*, sanjeev.kc6@gmail.com

Follow this and additional works at: https://digitalscholarship.unlv.edu/thesesdissertations

Part of the Geometry and Topology Commons, and the Theory and Algorithms Commons

## Repository Citation

K C, Sanjeev, "Efficient Estimation of Cluster Population" (2015). *UNLV Theses, Dissertations, Professional Papers, and Capstones*. 2370.
https://digitalscholarship.unlv.edu/thesesdissertations/2370

This Thesis is protected by copyright and/or related rights. It has been brought to you by Digital Scholarship@UNLV with permission from the rights-holder(s). You are free to use this Thesis in any way that is permitted by the copyright and related rights legislation that applies to your use. For other uses you need to obtain permission from the rights-holder(s) directly, unless additional rights are indicated by a Creative Commons license in the record and/or on the work itself.

This Thesis has been accepted for inclusion in UNLV Theses, Dissertations, Professional Papers, and Capstones by an authorized administrator of Digital Scholarship@UNLV. For more information, please contact digitalscholarship@unlv.edu.

# EFFICIENT ESTIMATION OF CLUSTER POPULATION

by

Sanjeev K C

Bachelor of Computer Engineering
Tribhuvan University
Institute of Engineering, Pulchowk Campus
2010

A thesis submitted in partial fulfillment of
the requirements for the

Master of Science – Computer Science

Department of Computer Science
Howard R. Hughes College of Engineering
The Graduate College

University of Nevada, Las Vegas
May 2015

Copyright by Sanjeev K C, 2015

All Rights Reserved

We recommend the thesis prepared under our supervision by

**Sanjeev K C**

entitled

**Efficient Estimation of Cluster Population**

is approved in partial fulfillment of the requirements for the degree of

**Master of Science in Computer Science**
**Department of Computer Science**

Laxmi P. Gewali, Ph.D., Committee Chair

John T. Minor, Ph.D., Committee Member

Ajoy K. Datta, Ph.D., Committee Member

Henry Selvaraj , Ph.D., Graduate College Representative

Kathryn Hausbeck Korgan, Ph.D., Interim Dean of the Graduate College

May 2015

# Abstract

Partitioning a given set of points into clusters is a well known problem in pattern recognition, data mining, and knowledge discovery. One of the well known methods for identifying clusters in Euclidean space is the K-mean algorithm. In using the K-mean clustering algorithm it is necessary to know the value of k (the number of clusters) in advance. We propose to develop algorithms for good estimation of k for points distributed in two dimensions. The techniques we pursue include a bucketing method, g-hop neighbors, and Voronoi diagrams. We also present experimental results for examining the performances of the bucketing method and K-mean algorithm.

# Acknowledgements

I would like to express my sincere gratitude to my supervisor *Dr. Laxmi Gewali* for his valuable guidance and support during the completion of this thesis. I would also like to thank *Dr. Ajoy Datta* for his help in official and academic difficulties and confusions. Furthermore, I would like to thank *Dr. John Minor* and *Dr. Henry Selvaraj* for being a part of my thesis committee.

Moreover, my courteous appreciation goes to my parents, my wife and my family members for their unconditional support and inspiration in each and every steps.

Last but not least, I would like to thank all my friends, juniors and seniors for their love and support.

Sanjeev K C

*University of Nevada, Las Vegas*

*May 2015*

# Table of Contents

**Abstract** **iii**

**Acknowledgements** **iv**

**Table of Contents** **v**

**List of Tables** **vii**

**List of Figures** **viii**

**Chapter 1 Introduction** **1**

**Chapter 2 Review of Clustering Algorithm** **3**

2.1 Generic strategies for Clustering . . . . . . . . . . . . . . . . . . . . . . . . 3

2.2 K-Mean algorithm . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5

**Chapter 3 Estimation of Cluster Centers** **7**

3.1 Chapter Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7

3.2 Adaptive Bucketing . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8

3.3 Aggregating buckets of a cluster . . . . . . . . . . . . . . . . . . . . . . . 13

3.4 Nudging . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 14

3.5 Local Density Estimation . . . . . . . . . . . . . . . . . . . . . . . . . . . 17

3.5.1 g-Hop Neighbor . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17

3.5.2 Voronoi Based g-hop . . . . . . . . . . . . . . . . . . . . . . . . . . 18

3.6 Randomized Approach . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 22

3.7 Measuring Solution Quality . . . . . . . . . . . . . . . . . . . . . . . . . . 23

**Chapter 4 Implementation** **24**

4.1 GUI Description . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 24

4.2 Interface Description . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 25

4.3 Execution of Bucket Clustering algorithm . . . . . . . . . . . . . . . . . . . . 27

4.4 Results and statistics . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 30

**Chapter 5 Conclusion and Discussion** **41**

**Bibliography** **43**

**Curriculum Vitae** **44**

# List of Tables


4.1 File Menu Items Description. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 26

4.2 Checkbox Description. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 27

4.3 Button Description. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 27

4.4 Textbox Description. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 27

4.5 Dataset result Mapping . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 30

4.6 Dataset 1 Experimental results . . . . . . . . . . . . . . . . . . . . . . . . . . . 38

4.7 Dataset 2 Experimental results . . . . . . . . . . . . . . . . . . . . . . . . . . . 38

4.8 Dataset 3 Experimental results . . . . . . . . . . . . . . . . . . . . . . . . . . . 38

4.9 Dataset 4 Experimental results . . . . . . . . . . . . . . . . . . . . . . . . . . . 38

4.10 Dataset 5 Experimental results . . . . . . . . . . . . . . . . . . . . . . . . . . . 39

4.11 Dataset 6 Experimental results . . . . . . . . . . . . . . . . . . . . . . . . . . . 39

4.12 Dataset 7 Experimental results . . . . . . . . . . . . . . . . . . . . . . . . . . . 40

# List of Figures


2.1 Illustrating Hierarchical strategy for Clustering . . . . . . . . . . . . . . . . . . . . 4
2.2 Illustrating tree of cluster combinations . . . . . . . . . . . . . . . . . . . . . . . . . 4

3.1 Illustrating a set of clustered nodes . . . . . . . . . . . . . . . . . . . . . . . . . . . 8
3.2 Illustrating Bucket-Embedding . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9
3.3 Merging of Bucket Captured Clusters . . . . . . . . . . . . . . . . . . . . . . . . . . 9
3.4 Bucket index and point mapping . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11
3.5 Point distribution with two clusters . . . . . . . . . . . . . . . . . . . . . . . . . . 13
3.6 Extraction of coarse cluster . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15
3.7 Cluster refinement by nudging . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 16
3.8 Nudging Types . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 16
3.9 Points in the neighborhood of a point . . . . . . . . . . . . . . . . . . . . . . . . . 17
3.10 Showing the 3-hop neighbor . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 18
3.11 Illustrating Voronoi Diagram of 50 point sites . . . . . . . . . . . . . . . . . . . . . 19
3.12 Illustrating 1-hop and 2-hop rings . . . . . . . . . . . . . . . . . . . . . . . . . . . 20
3.13 Input points . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 22
3.14 Sampled points using Random Approach . . . . . . . . . . . . . . . . . . . . . . . . 23

4.1 Graphical User Interface layout. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 25
4.2 Actual Graphical User Interface . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 26
4.3 Set of input points . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 28
4.4 Cluster obtained using bucket clustering algorithm . . . . . . . . . . . . . . . . . . 29
4.5 Clusters after Nudging . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 29
4.6 Labeling buckets as high and low . . . . . . . . . . . . . . . . . . . . . . . . . . . 30

4.7 Dataset 1 with initial centroids at point 9, 16, 56, 33 . . . . . . . . . . . . . . . . 31
4.8 Dataset 1 after executing bucketing algorithm and nudging . . . . . . . . . . . 31
4.9 Dataset 2 with initial centroids at point 50, 7, 28 . . . . . . . . . . . . . . . . . . 32
4.10 Dataset 2 after executing bucketing algorithm . . . . . . . . . . . . . . . . . . . . 32
4.11 Dataset 3 with initial centroids at point 9, 16, 56, 35 . . . . . . . . . . . . . . . . 33
4.12 Dataset 3 after executing bucketing algorithm and nudging . . . . . . . . . . . 33
4.13 Dataset 4 with initial centroids at point 0, 91, 150, 72, 112 . . . . . . . . . . . 34
4.14 Dataset 4 after executing bucketing algorithm . . . . . . . . . . . . . . . . . . . . 34
4.15 Dataset 5 with initial centroids at point 9, 102, 33, 191, 379, 492 . . . . . . . . 35
4.16 Dataset 5 after executing bucketing algorithm and nudging . . . . . . . . . . . 35
4.17 Dataset 6 with initial centroids at point 168, 55, 104, 392 . . . . . . . . . . . . 36
4.18 Dataset 6 after executing bucketing algorithm and nudging . . . . . . . . . . . 36
4.19 Dataset 7 with initial centroids at point 374, 68, 296, 488 . . . . . . . . . . . . 37
4.20 Dataset 7 after executing bucketing algorithm . . . . . . . . . . . . . . . . . . . . 37

# Chapter 1

# Introduction

Clustering is a technique of identifying 'closely related points' from a collection of large number of data. Closely related points in terms of some distance metric are grouped together as a cluster. In most input data there could be several blocks of clusters. The notion of perceiving clusters in a given distribution of points has been considered from the very dawn of civilization. Distribution of stars in the night sky can be considered as a distribution of points, and groups of stars in the form of zodiacs, ursa-major, ursa-minor, and the milky way can be viewed as star clusters.

Cluster analysis is extensively used in many fields that include statistics, medicine, the social sciences and humanities [6]. In fact, any study that uses collection of data can make productive use of cluster analysis.

Most of the early research on cluster analysis was done by considering the point distribution in Euclidean space, where an Euclidean metric is used to measure distance between points. In this setting, distance between a pair of points in the same cluster is distinctly smaller than the distance between a pair formed by taking one point from the cluster and the other from outside the cluster.

After the advent of computer science, researchers considered the problem of developing efficient algorithms for extracting clusters [3] [6] [7]. The K-Mean algorithm and its variations are examples of practical algorithms for identifying clusters in Euclidean space. In recent years, there has been a surge in research interest for identifying clusters in big-data. In normal data we can assume that all the data is available in the main memory, and al-

gorithms are developed by considering the standard RAM model. In big-data, not all the data can be stored in RAM. The challenge is to develop cluster identification algorithms when data is available in external memory and the cloud storage.

In some applications, an Euclidean metric can not be used to measure distance between points. The data points could be visitors to Las Vegas entertainment sites, and we may be interested to identify a cluster of visitors who visit casino sites and are coming from Hong Kong. Straightforward use of Euclidean metric may not be applicable in such data to extract clusters. We need to come up with an appropriate metric other than Euclidean.

In statistics, a widely used technique for cluster analysis is the method of principal component analysis (pca). In this approach an orthogonal transformation is performed to obtain linearly uncorrelated data from possibly correlated ones [5].

In this thesis we address the issues of estimating the number of clusters for points distributed in Euclidean space. In Chapter 2, we present a critical review of the prominent existing methods for extracting clusters. In Chapter 3, we present the main contribution of the thesis. We present several algorithms for estimating the number of clusters and the location of their centers. The algorithms we present include (i) bucketing method, (ii) g-hop neighbors, and (iii) Voronoi-based g-hop neighbors. In Chapter 4, we present implementation of some of the techniques presented in Chapter 3. The implementation is done in the Java Programming Language. Finally, in Chapter 5, we describe possible extensions and generalizations of proposed algorithms, and avenues and scope for future work.

# Chapter 2

# Review of Clustering Algorithm

In this chapter we present a critical review of well known clustering algorithms reported in computer science and application literature. In our review we particularly focus on the application of the tools from computational geometry for developing efficient clustering algorithms. Clustering algorithm have been reported in engineering and statistics literature for almost one hundred years [6] [7]. Most of the clustering algorithms assume that the input points are distributed in Euclidean space. In recent years there has been extensive interest among big-data researchers to develop clustering techniques in non-Euclidean space. Furthermore, extracting clusters from cloud stored big-data (in the range of Xetabytes) warrants the development of new approaches and insights.

## 2.1 Generic strategies for Clustering

Most of the clustering algorithms reported in the literature can be broadly classified into two kinds. The first kind of algorithms are developed by using a *hierarchical scheme* and the other is the *point assignment*. A detailed discussion of these approaches are found in [6].

In the hierarchical scheme, each of the points $p_i$'s in the input data are considered themselves as clusters. Each cluster is associated with its centroid point which is taken as the arithmetic mean of the coordinates of the points in the cluster. Two clusters are picked to combine by formulating some metric. One simple way of combining clusters is to pick a pair of clusters whose centroids are closest. Another way to combine clusters is to consider

the smallest distance between nodes from one cluster to the other. When a new cluster is formed by combining two smaller clusters, the corresponding centroid is also computed. The process of combining two clusters is continued until all points are grouped into one cluster. In some sense the hierarchical clustering scheme works by following the spirit of the construction of a minimum spanning tree by using Kruskals' algorithm [2]. We can illustrate this strategy by an example shown in Figure 2.1.

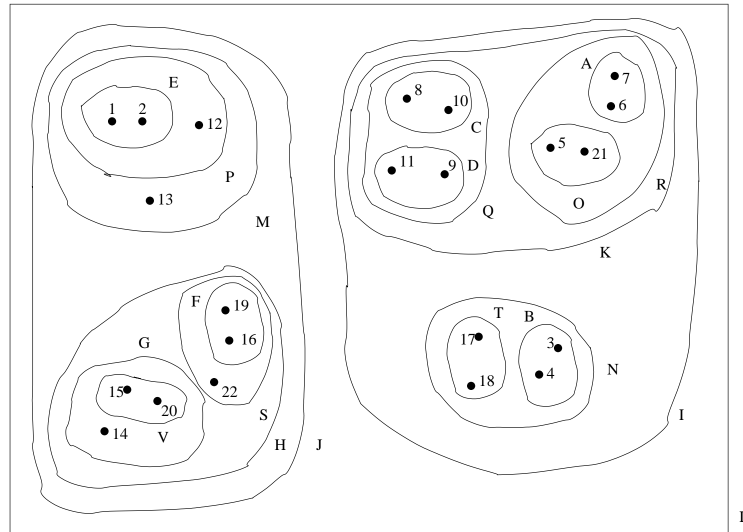


Figure 2.1: Illustrating Hierarchical strategy for Clustering

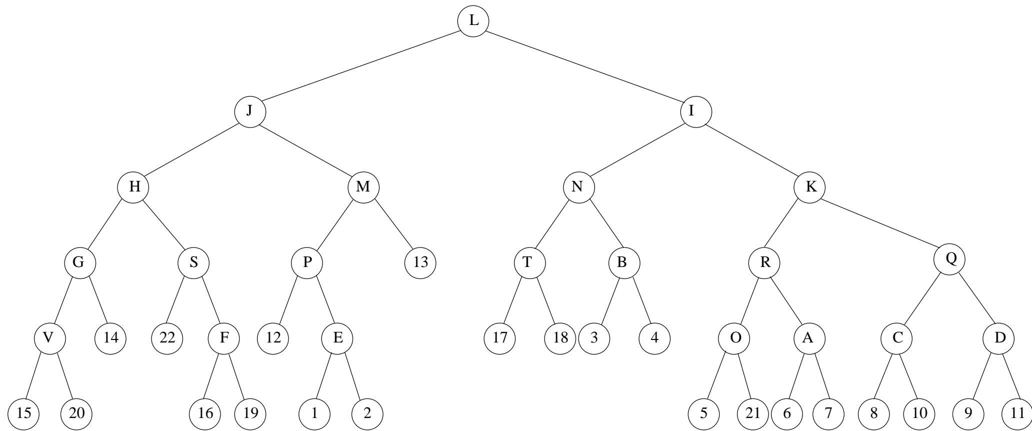


Figure 2.2: Illustrating tree of cluster combinations

In the point assignment strategy, clustering algorithms are developed by making an initial estimate of the number of clusters that are constructed by adding points one by one to the initial partial clusters. The k-mean algorithm described in the next section is an

example of this strategy.

## 2.2 K-Mean algorithm

The K-Mean Algorithm was first formally introduced by Stuart Lloyd [7] in connection with its application to pulse code modulation at Bell Lab. This algorithm is perhaps the most widely referred clustering algorithm for almost 35 years. The algorithm works for points distributed in Euclidean space. The algorithm assumes the number of clusters as a part of the input. The location of the initial $k$ points is also specified by the user of the algorithm. The algorithm grows the clusters by adding carefully selected nodes to one of the clusters. Initially, each of the $k$ clusters have one node. The locations of the initial single member in the clusters are taken as their centroids. The algorithm progresses through a series of steps to grow clusters by adding one node at a time. The nodes outside the clusters are unprocessed nodes . The algorithm examines an unprocessed node $p_i$ as the next candidate point. The candidate point $p_i$ is added to the cluster whose center is closest to $p_i$. This process of "adding a candidate point" is continued until all nodes are processed. When all points are processed, one pass of the "clusters construction" is completed. After the completion of a pass the centroids are recomputed. The updated centroid of a cluster $C_i$ is the centroid of all points included in it. A new pass of computation starts again with respect to the newly updated centroids. In each pass the estimation of centroids and the corresponding cluster is updated. The initiation of the next pass stops when cluster members do not change or the change in the location of centroids is below a certain predetermined threshold value. A formal sketch of the algorithm is listed as K-Mean Algorithm (Algorithm 2.1).

**Algorithm 2.1:** K-Mean Algorithm

**Input:** (i) Set of points $S = p_0, p_1, ...., p_{n-1}$ in 2D

(ii) Integer k

(iii) Threshold value $\delta$

**Output:** Clusters of point sets $C_1, C_2, ...., C_k$

**Step 1:** (i) Pick well-separated k points $q_1, q_2, ....., q_k$

(ii) Let $t_i$ be the centroid for $C_i$.

(iii) Set $t_i$ to $q_i$'s

**Step 2:** (i) CentroidMovement = LargeNumber;

(ii) ClusterChangeFlag = true;

**Step 3:** while (CentroidMovement > $\delta$ and ClusterChangeFlag == true) {

**Step 4:** (i) Mark all points in $S$ 'unprocessed'

(ii) Initialize new clusters $C_i$''s to empty

**Step 5:** for (int $i = 0$; $i < n$; $i$++) {

(a) Let $t_j$ be the centroid closest to $p_i$

(ii) Include $p_i$ into $C_i$'

(iii) Mark $p_i$ 'processed'

}

**Step 6:** Compute new centroids $t_i$''s

**Step 7:** Set ClusterChangeFlag by comparing old $C_i$'s to new clusters $C_i$''s

**Step 8:** Set CentroidMovement by comparing $t_i$'s to $t_i$''s

**Step 9:** Set $C_i$'s to $C_i$''s

**Step 10:** } // end while

# Chapter 3

# Estimation of Cluster Centers

## 3.1 Chapter Summary

One of the most popular methods for constructing clusters from a given set of points distributed in Euclidean space is the $k$-mean algorithm [7]. This algorithm assumes that the number of clusters $k$ is known in advance. If the value of $k$ is not given as a part of the input then we need to estimate it 'somehow'. One straightforward technique would be to repeat the execution of the algorithm for several values of $k$ and evaluate the quality of resulting solutions. The value of $k$ that corresponds to the best value of cluster quality is the desired answer. A brute-force method is to try all values of $k$ = 2,3,4,... $n$. A faster method based on the binary search technique has been suggested [6] for searching for the value of $k$. Obviously the binary search technique is only effective where the quality of cluster as a function of $k$ is a monotone function. An exhaustive searching approach has several demerits: (i) executing the clustering algorithm repeatedly is time consuming, (ii) measuring the quality of a candidate solution is not precise, and (iii) locating the cluster center for a given value of $k$ is itself a difficult and critical problem. We present three approaches for estimating the value of $k$ and their center's locations (co-ordinates) for points distributed in the Euclidean plane. The first approach called 'adaptive-bucketing' estimates $k$ by partitioning the region containing the input points into orthogonal buckets. The second approach called 'g-hop capture' estimates the value of $k$ by examining the g-hop neighbors of the input points. Finally, the third approach we present is based on the principle of randomization. In this approach a subset of input points is randomly selected and these points are processed by us-

ing adaptive-bucketing and/or g-hop capture to estimate the value of $k$ and the co-ordinates of estimated centers.

## 3.2 Adaptive Bucketing

Without loss of generality we can assume that the input point-sites $p_0, p_1, ..., p_n$ are inside a rectangular box $R$ of height $= h$ and width $= w$. The box $R$ can be divided into $nxm$ orthogonal buckets. The value of bucket size $m$ can be pre-determined by examining the distribution of the nearest neighbor distance distribution for $n$ input points. The exact method for estimating the value of $m$ will be described at the end of this chapter. An example of partitioning the bounding box $R$ into orthogonal buckets is shown in Figure 3.1-3.3.

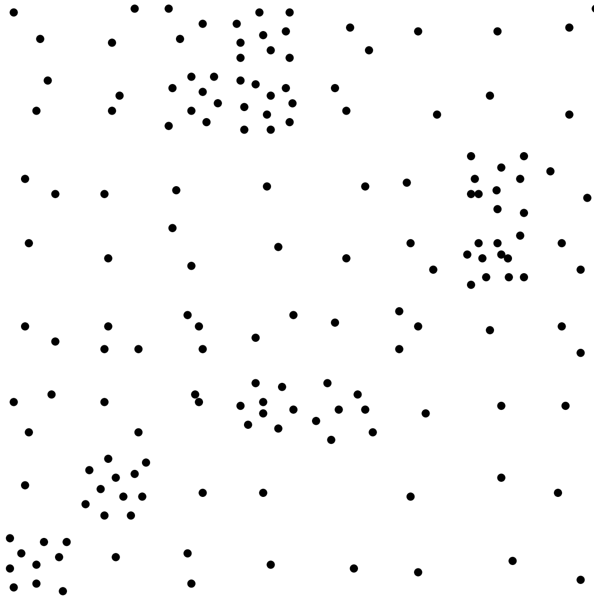

Figure 3.1: Illustrating a set of clustered nodes

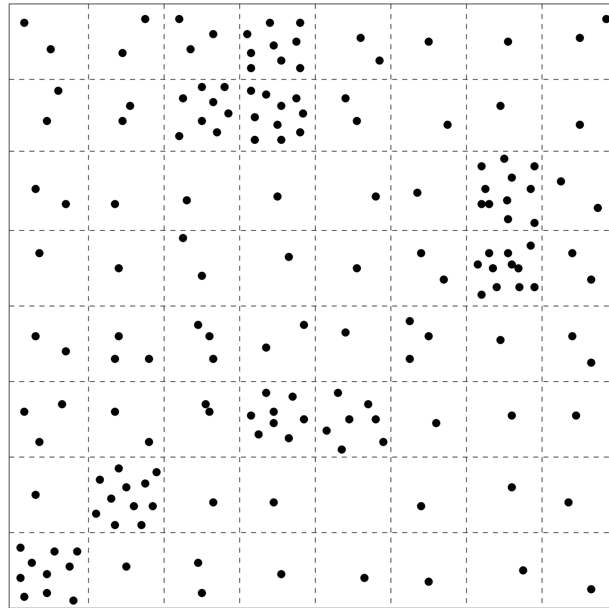

Figure 3.2: Illustrating Bucket-Embedding

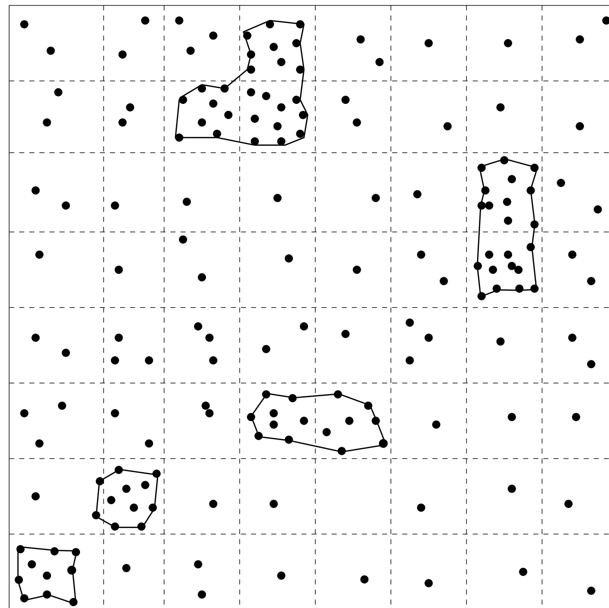

Figure 3.3: Merging of Bucket Captured Clusters

A straight-forward approach for counting the points in each bucket is to check for point inclusion in each $nxm$ buckets. The bucket that returns 'true' for point inclusion is the bucket containing the point. Since the buckets are disjoint, only one bucket will return true for inclusion for a given point.

To implement this approach we maintain a count array $cnt[nm-1]$ whose entries are initialized to zeros at the start. An array $bx[]$ holds the coordinates of the top left corner of buckets. If the inclusion test for point $p_i(x_i, y_i)$ against bucket $bx[j]$ returns true then $cnt[bx[j]]$ is increased by 1. When this check is repeated for all points, point counts for all buckets is complete. A formal sketch of the algorithm based on this approach is listed as Straightforward Count Algorithm (Algorithm 3.1)

**Algorithm 3.1:** Straightforward Count Algorithm

**Input:** (i) *p[N]*; // Input points in 2D

(ii) int *n, m*; // Number of bucket rows and columns

(iii) int *bx[n,m]*; //Array to hold top left corner of buckets

(iv) int $k_v$, $k_h$; // length and width of each bucket

**Output:** *cnt[]*; // Array to hold count of bucket

**Step 1:** // Read input

read *p[N], n, m* ,$k_v$,$k_h$

**Step 2:** // Initialize *cnt[]* to 0's

```
for (int i = 0; i < n*m; i++)
    cnt[i] = 0;
```

**Step 3:**

```
for (int i = 0; i <N; i++) {
    for (int j = 0; j < n*m; j++) {
        if (inside(bx[j], p[i]))
            cnt[bx[j]]++;
    }
}
```

**Step 4:** Output *cnt[]*

The time complexity of Algorithm 3.1 can be done as follows. Step 1 takes $O(N+nm)$. Step 2 takes $O(nm)$. Step 3 takes $O(Nnm)$ which is the dominating step in terms of

complexity. Hence the overall time complexity is $O(Nnm)$. If $n$*$m$ is comparable to $N$ then the time complexity becomes $O(N^2)$ which is rather high.

**Mapping Count Approach**

This approach is used to directly map $p_i$ to the bucket $b[j]$ where it falls. Since the size of buckets are the same and rectangular, the index of the bucket where point $p_i$ falls can be computed in term of the row number, column number, width, and height of the bucket. It is given that the outer rectangle $R$ bounding the input points is partitioned into $n$ columns and $m$ rows of buckets, each of size $k_v$*$k_h$. Here $k_v$ is the vertical extent of the bucket and $k_h$ its horizontal width. For a given point $p_i(x_i, y_i)$, its row number $r_n$ is given by $r_n = y_i/k_v$ + 1 and column number $c_n = x_i/k_h$ +1. We can index buckets left to right and top to bottom as 1, 2, ......, $n$*$m$ as shown in Figure 3.4. Then the bucket index corresponding to point $p_i(x_i, y_i)$ is $(r_n - 1) * n + c_n$. As an example, point $p_1(55, 25)$ is mapped bucket (3-1)*5 + 4 = 14.

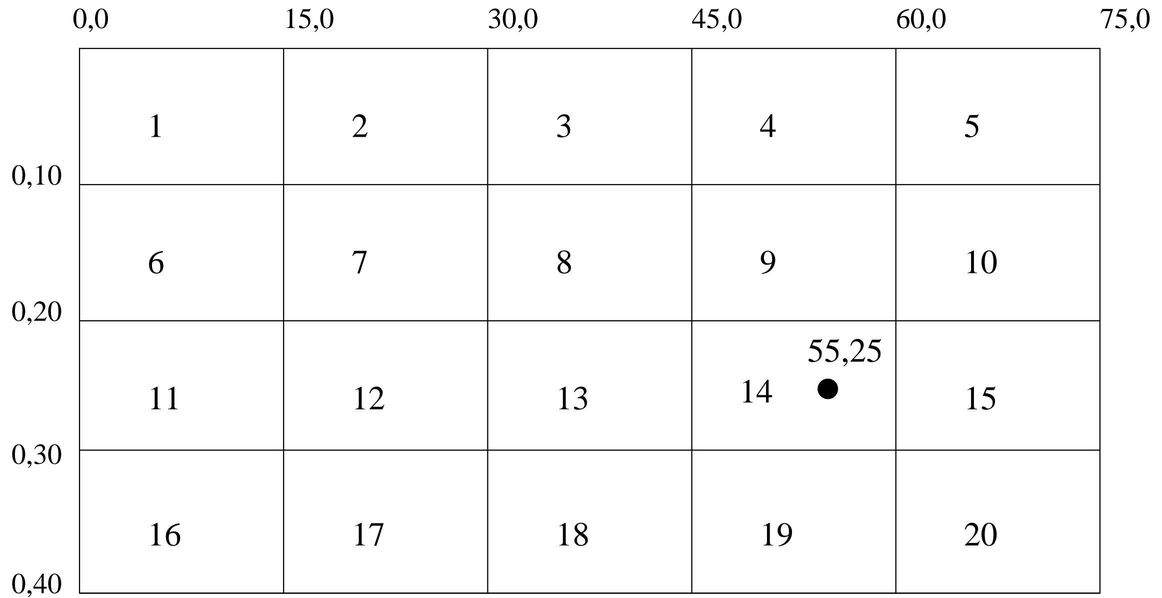


Figure 3.4: Bucket index and point mapping

Based on this mapping, the following is a faster algorithm (Algorithm 3.2) called Mapping Count Algorithm.

**Algorithm 3.2:** Mapping Count Algorithm

**Input:** (i) $p[N]$; // Input points in 2D

(ii) int $n, m$; // Number of bucket rows and columns

(iii) int $bx[n, m]$; //Array to hold top left corner of buckets

(iv) int $k_v$,$k_h$; // length and width of each bucket

**Output:** *cnt[n*m]*; // Array to hold count of bucket

**Step 1:** read *p[N], n, m,* $k_v$, $k_h$ // Read input

**Step 2:** for(int *i = 0; i <n*m; i++*) // Initialize cnt[] to 0's

*cnt[i] = 0*;

**Step 3:** for(int *i = 0; i < N; i++*) {

$r_n = y_i/k_v + 1$;

$c_n = x_i/k_h + 1$;

$j = (r_n - 1) * n + c_n$

*cnt[bx[j]]++*;

}

**Step 4:** Output *cnt[n*m]*

The time complexity of Algorithm 3.2 can now be analyzed. Step 1 and Step 2 each take $O(N + nm)$ and $O(nm)$, respectively. The for loop of Step 3 executes $O(N)$ times and one execution of the body of the for loop takes $O(1)$ time. Hence Step 1 is the dominant step and hence the total time complexity of Algorithm 3.2 is $O(N + nm)$. This time complexity is optimal in the sense that it takes $O(N)$ time to read the points and $n*m$ is at most $N$ (Remark 3.1).

**Remark 3.1 (Number of buckets):** The very purpose of using buckets fails if there are too many buckets. For making the bucketing approach efficient we do not want to have many empty buckets. At the same time to identify the boundaries of clusters we should have enough buckets. A good upper bound for the number of rows and columns in bucket partitioning is $\sqrt{N}$. In some applications, the number of rows and columns is much smaller than $\sqrt{N}$, and in some cases it's even constant.

### 3.3 Aggregating buckets of a cluster

After identifying buckets containing a high concentration of points, it is now necessary to aggregate buckets together belonging to the same cluster. We can clarify this with the following example in Figure 3.5

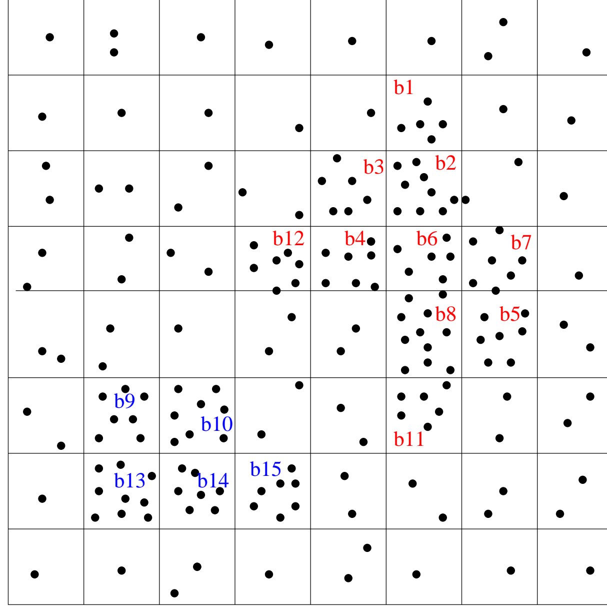


Figure 3.5: Point distribution with two clusters

In this example there are two clusters $C_1$ and $C_2$. Cluster $C_1$ has 10 buckets $b_1$, $b_2$, $b_3$, $b_4$, $b_5$ , $b_6$, $b_7$, $b_8$, $b_{11}$, $b_{12}$ and cluster $C_2$ has five buckets $b_9$, $b_{10}$, $b_{13}$, $b_{14}$, $b_{15}$. Suppose the starting bucket is $b_6$. The algorithm proceeds by initializing a queue $Q_b$ by inserting the starting bucket $b_6$ to $Q_b$. The algorithm then repeats the following generic task until all buckets of the cluster are aggregated.

> **Generic Task :** Pick the bucket $b_j$ from the front of the queue $Q_b$ and enqueue all 4-connected neighbors of $b_i$ that are marked H. Bucket $b_j$ is pushed onto stack $S_b$ and $b_j$ is marked processed.

In our running example, bucket $b_6$ is removed from the front of the queue $Q_b$ and its 'H' marked neighbors that have not been processed yet($b_2$, $b_4$, $b_8$ and $b_7$) are enqueued onto queue $Q_b$. Bucket $b_6$ is marked processed. Next bucket $b_2$ is removed from the queue and

its unprocessed neighbors that are marked 'H'($b_1$ and $b_3$) are enqueued onto the queue. Bucket $b_2$ is marked processed. These operations on stack and queue are repeated until the queue is empty. When the queue is empty all the buckets of the cluster in the context are present in the stack. A formal sketch of the algorithm which we refer to as Bucket Clustering Algorithm is listed as Algorithm 3.3

**Algorithm 3.3:** Bucket Clustering Algorithm

**Input:** (i) An array $b[]$ of size $m * n$ representing the top left co-ordinates of buckets

(ii) A given starting bucket index q that belongs to current cluster

**Output:** A stack containing the buckets representing the cluster counting b[q]

**Step 1:** Q = b[q]; // Initialize queue Q

// Initialize stack $S_b$ to be empty

**Step 2:** while (Q is not empty) {

a. $P_x$ = Q.delete();

b. Let $R_c$ be set of unprocessed h-neighbors of $P_x$

c. Insert points in $R_c$ into Q

d. Push $P_x$ into stack $S_b$

e. Mark points in $R_c$ 'processed'

}

**Step 3:** Output $S_b$

## 3.4 Nudging

A straightforward application of the bucketing technique aggregates high count buckets (H-buckets) to extract a cluster. We refer to the clusters constructed in this way as *coarse clusters* and their boundaries as *coarse boundaries.* Some points in L-clusters adjacent to coarse boundaries are not included in the cluster even if they are very close to the fence of a H-bucket. Of course, points in L-buckets adjacent to a coarse boundary should not be included in the cluster if such points are farther away from the boundary and appear disconnected to the cluster. In Figure 3.6, the cluster at the center is formed by aggregating 8 buckets [4,4], [5,4], [4,5], [5,5], [6,5], [3,6], [4,6] and [5,6]. However, boundary points in

low count buckets [4,7], [5,7], [6,7] and [6,6] should be included in the cluster. When such boundary points are included in the cluster we get better estimation of the cluster as shown in Figure 3.7.

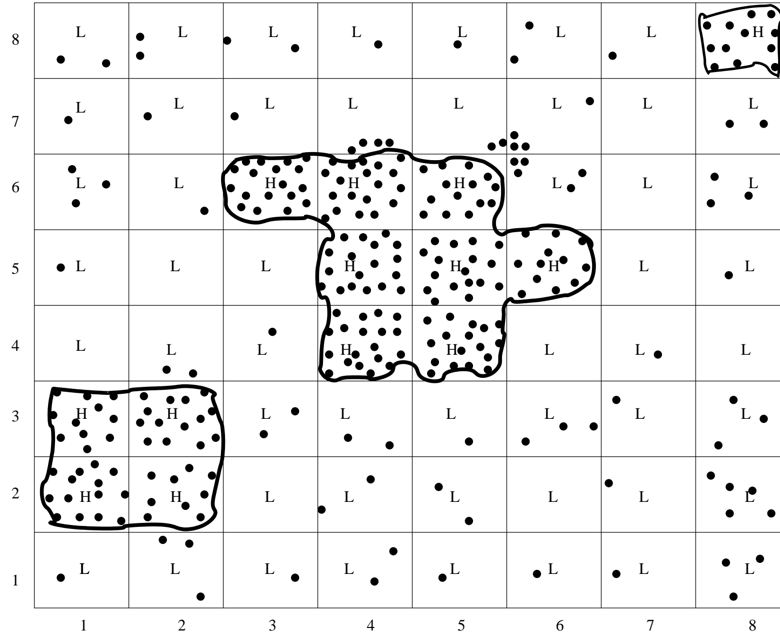


Figure 3.6: Extraction of coarse cluster

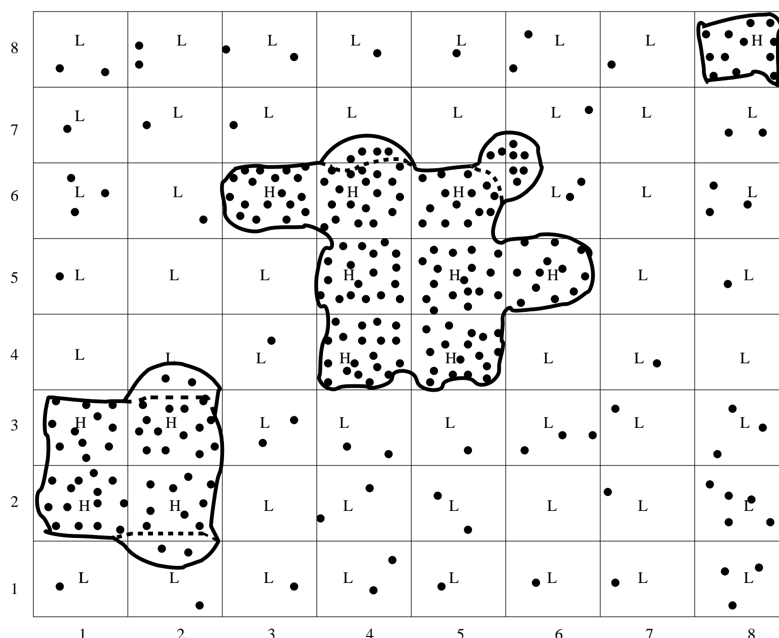


Figure 3.7: Cluster refinement by nudging

Now we describe a formal way of identifying points near the coarse boundary that can be included in the cluster. Our approach is to nudge coarse boundaries to capture proximity points in the corresponding cluster.

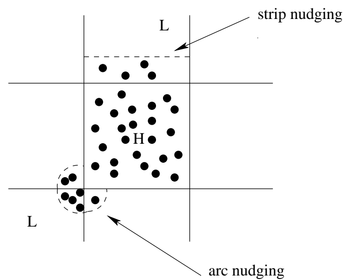


Figure 3.8: Nudging Types

Consider a H-bucket adjacent to a coarse boundary as shown in Figure 3.8. If a L-bucket shares an edge with a H-bucket, then we can inspect points inside a rectangle of size $l$ x $l/4$ (*strip rectangle*) as shown in Figure 3.8 to possibly include in the cluster, where $l$ is the side length of the bucket. This is called *strip nudging*. If a L-bucket is adjacent to a corner of a H-bucket then we should inspect points inside an arc of radius $l/4$ and angle $3\pi/4$, as shown in the lower left of Figure 3.8. This technique is called *arc nudging*.

## 3.5 Local Density Estimation

A point $p_i$ is a very good candidate for the cluster center if there are a lot of points in its neighborhood. In other words, the region around $p_i$ has a higher number of points per unit area, i.e. higher density region. So, to estimate the density of points in the neighborhood of $p_i$ we need to count points inside a small region enclosing $p_i$ as shown in Figure 3.9.

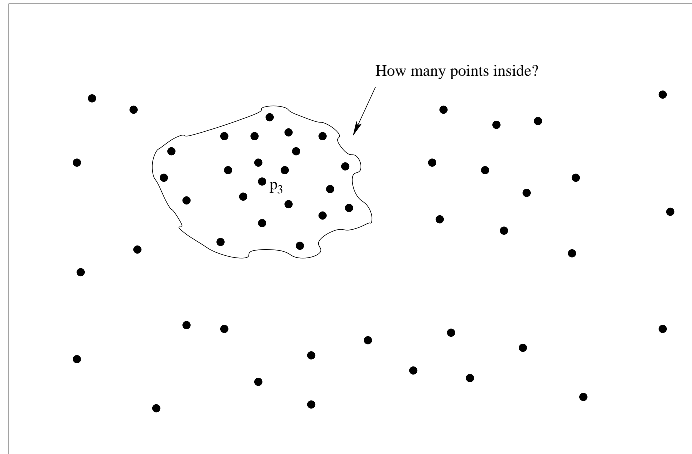


Figure 3.9: Points in the neighborhood of a point

The first issue here is how to specify the small local region around $p_i$ and the second issue is to find ways to compute the points in such an area quickly. The easiest and the most logical way to specify the region enclosing $p_i$ is a circle with $p_i$ as center and a small radius. The radius should be comparable to the side length of the bucket. To determine the number of points inside the circle we need to perform an inclusion check for all $N$ points. A slightly different approach is to use the concept of *g-hop* as discussed next.

### 3.5.1 g-Hop Neighbor

For each point site $p_i$, we can define its g-Hop neighbor, if it exists. This can be defined iteratively as follows. The 1-hop neighbor of $p_i$ denoted as *1-hop($p_i$)* is the nearest neighbor of $p_i$. Let $dsk(p_i, 1)$ denote the disk with center at $p_i$ and radius equal to the distance between $p_i$ and *1-hop($p_i$)*. The 2-hop neighbor of $p_i$ is the point site closest to *1-hop($p_i$)* that lies outside the disk $dsk(p_i, 1)$. In general, the g-hop neighbor of $p_i$, denoted by *g-hop($p_i$)*, is the point site closest to *(g-1)-hop($p_i$)* that lies outside of $dsk(p_i, g-1)$. These concepts are illustrated in Figure 3.10.

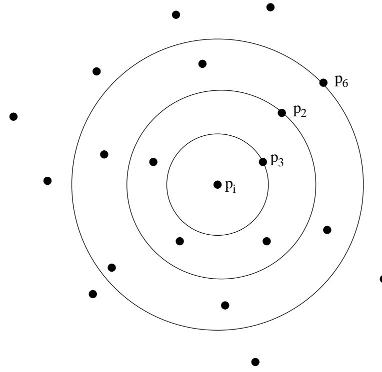


Figure 3.10: Showing the 3-hop neighbor

**Observation 3.1** : It immediately follows that for a given point site $p_i$, *g-hop($p_i$)* is farther away from *g'-hop($p_i$)* for g > g'

A straightforward way of computing *g-hop($p_i$)* is to use the iterative definition. To compute *(g+1)-hop($p_i$)* we need to check the distance of all point sites from *g-hop($p_i$)* that lie outside of disk $dsk(p_i, g-1)$. This checking takes $O(n)$ time. Hence the total time taken by this approach is $O(gn)$. So to determine the g-hop neighbors of all point sites it takes $O(g^2n^2)$ time.

### 3.5.2 Voronoi Based g-hop

A faster algorithm for computing a variation of g-hop neighbors of all point sites can be developed by using the Voronoi diagram[8] induced by the input points. The Voronoi diagram of $n$ point sites partitions the plane into $n$ cells $V(i)$, $1 \leq i \leq n$ such that all points in a cell $V(j)$ are nearer to site $p_j$ than all other sites. All Voronoi cells are convex polygons. Some Voronoi cells are bounded and others are unbounded. An example of a Voronoi diagram induced by 50 point sites is shown in Figure 3.11.

Given the Voronoi diagram of $n$ point sites, *Voronoi based g-hop neighbor* of $p_i$ can be computed by navigating the Voronoi cells starting from $V(i)$, the Voronoi cell for point site $p_i$. To describe the algorithm in a convenient way we assume that the Voronoi diagram is available in a Doubly Connected Edge List (DCEL) data structure [1] [8]. The edges of $V(i)$ are traversed by using the dcel data structure to check Voronoi neighbors of $p_i$. The Voronoi induced *1-hop neighbor* of $p_i$ is its nearest neighbor point site which is one of the point sites corresponding to adjacent cells of $V(i)$. Voronoi based g-hop neighbors can be

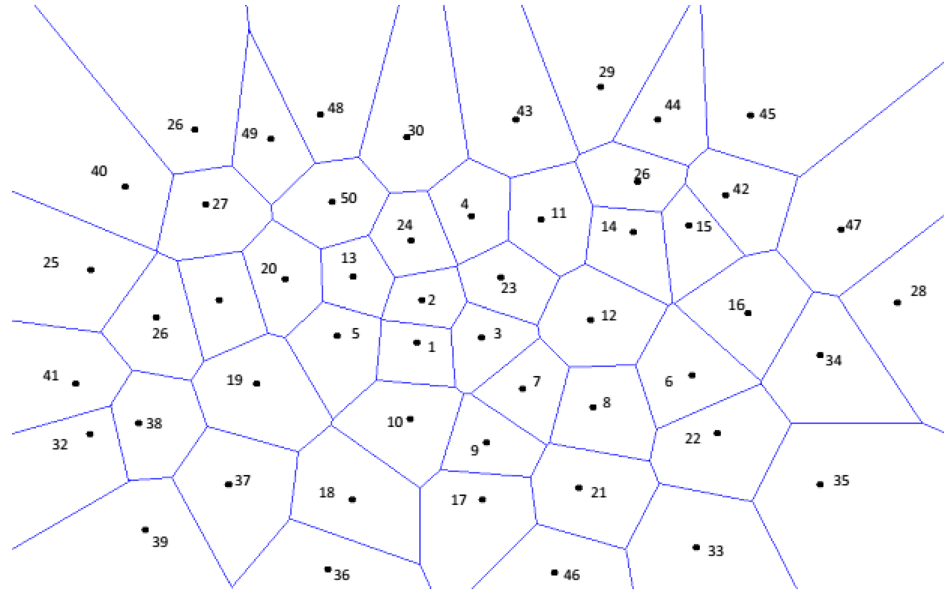


Figure 3.11: Illustrating Voronoi Diagram of 50 point sites

conveniently defined in term of *g-hop ring* as follows:

*g-hop ring* : For a given point site $p_i$, its *1-hop ring* is the chain of cells adjacent to $V(i)$. The *g-hop ring* of point site $p_i$ is the closed or open chain(s) of cells adjacent to the cells of *(g-1)-hop ring*, away from $p_i$. In Figure 3.12, *1-hop ring* and *2-hop ring* are shown for point site 3.

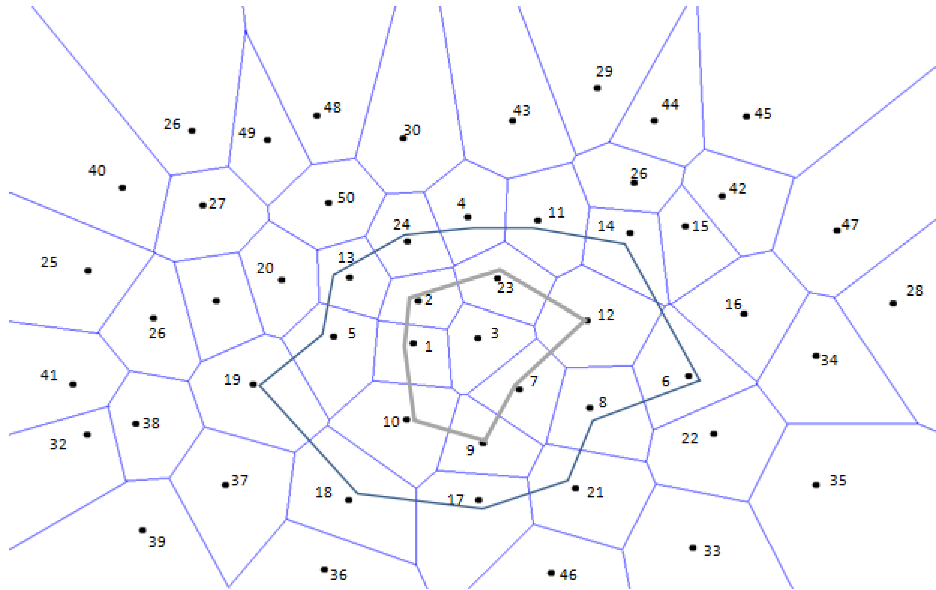


Figure 3.12: Illustrating 1-hop and 2-hop rings

Voronoi based g-hop neighbor of a point site $p_i$ is the closest site corresponding to cells in g-hop ring. If the i-hop ring contains an unbounded cell then the outward cells adjacent to the cells of i-hop ring do not form closed chains. In such situations we need to consider chains of cells and proceed.

To compute Voronoi based g-hop for a point site $p_i$ we start from cell $V(i)$ and process cells from inner rings to outer rings starting from the *1-hop ring*. Since the Voronoi diagram is available in doubly connected edge list form, we can navigate from one cell to adjacent cell by following twin edges of the dcel structure. The details of dcel data structure is in [1] [8].

A formal sketch of the algorithm which we refer to as Voronoi Based g-hop Estimation Algorithm is listed as Algorithm 3.4.

**Algorithm 3.4:** Voronoi Based g-Hop Estimation

**Input:** (i) A set point sites $p_0, p_1, p_2, ..., p_{N-1}$; // Input points in 2D

(ii) Threshold distance $g$

(iii) Candidate node $p_j$

**Output:** Number of points $k$ within distance $g$

**Step 1:** Compute the Voronoi diagram of input point sites and represent it in dcel form.

**Step 2:** (i) Initialize queue $Q$ to empty queue.

(ii) Insert the cell corresponding to $p_j$ into $Q$

(iii) Cell $u = Q$.delete(); Mark $u$ as 'processed';

(iv) d = 0;

(v) bool done = false; $k = 1$;

(vi) Mark all unbounded cells 'processed'

**Step 3:** while ((not done) and d <$g$) {

(ii) Let W be the set of unprocessed cells adjacent to $u$

(iii) If( W is not Empty) {

(a) Insert the cells in W into $Q$

(b) $u = Q$.deleteItem(); d = dist($p_j, u$); $k$++;

}

Else done = true;

}

**Step 4:** Output $k$;

**Theorem 3.1:** Voronoi based g-hop estimation algorithm can be executed in $O(NlogN)$ time.

**Proof:** Step 1 can be done in $O(NlogN)$ time by using Fortune's sweep line algorithm [1] [8]. Within the same time complexity the Voronoi diagram can be made available in DCEL data structure form. The most expensive operation in Step 2 is marking cells, adjacent to unbounded region, which takes $O(n)$ time. In Step 3 each cell is processed a constant

number of time due to the fact that a cell is inserted into the queue only once. Hence the total time can be charged to the edges processed in the cells which is bounded by $O(n)$. Hence Step 1 is the most expensive step and the total time is $O(NlogN)$.

### 3.6 Randomized Approach

If the number of data is very large we can use a randomly generated sample to construct an input data set of smaller size. If there are N input points $p_0, p_1, ..., p_{N-1}$ then a sample set of input points $q_0, q_1, ..., q_{k-1}$ of size $k$ can be constructed by using a random number generator such as Random() function in Java. The Random function can be used to randomly generate an integer between 0 and N. If the generated integer is j then $p_j$ is taken as one member of the sample. This process of generating a random integer can be repeated k times to obtain a random sample of size k. When generating the next random integer j, we include it in the sample set if it was not generated previously. We can use the bucketing method or g-hop method for estimating cluster centers on the sampled input. The result of a distribution of sample points and input points for 20% sample size is shown in Figure 3.14. Sampled points are drawn slightly bigger.

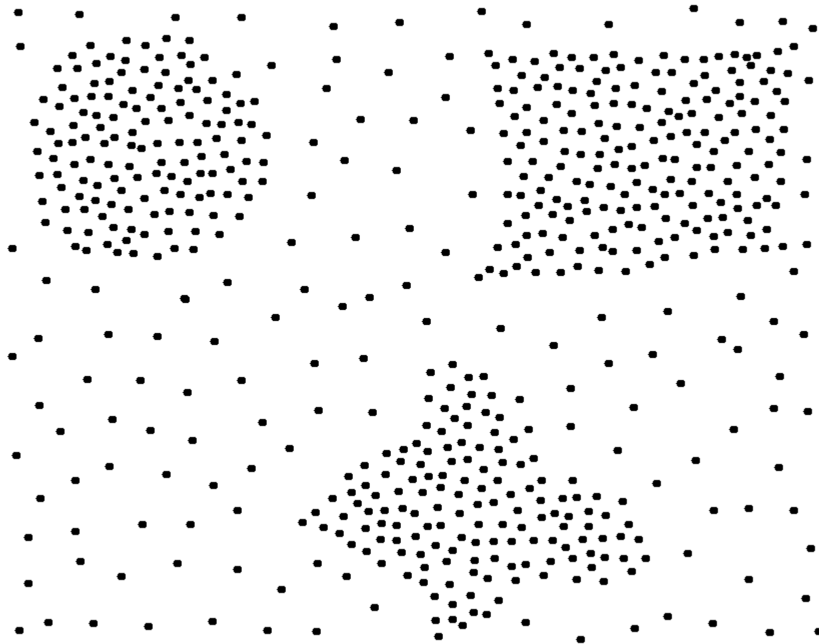

Figure 3.13: Input points

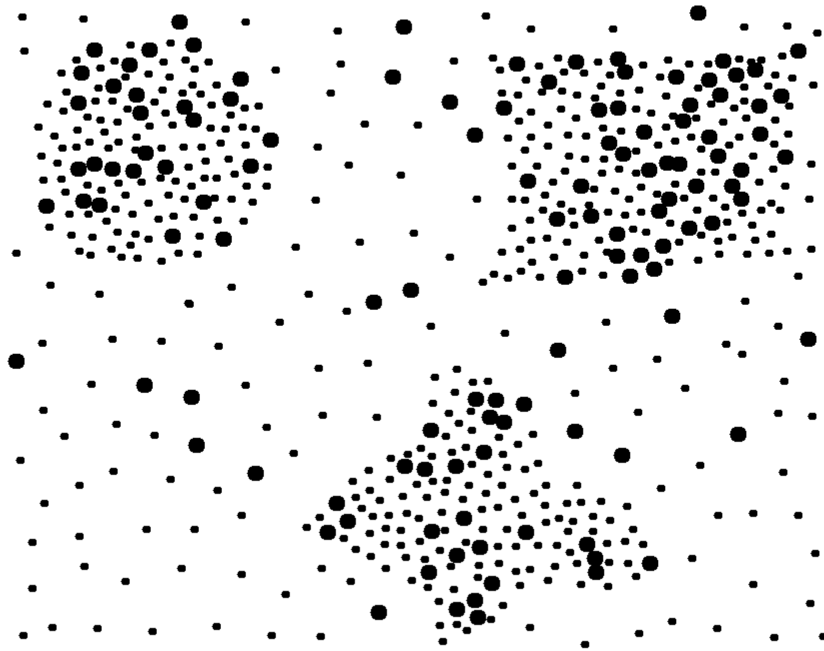

Figure 3.14: Sampled points using Random Approach

### 3.7 Measuring Solution Quality

To measure the quality of the solution obtained using the bucket clustering algorithm, we use the **sum of squared error (SSE)** as our objective function[3] [9]. We first calculate the squared error of each point to its closest centroid and compute the total sum of the squared errors for the clusters. Mathematically SSE can be defined as :

$$\text{SSE} = \sum_{i=0}^{K} \sum_{x \in C_i} dist(c_i, x)^2$$

where *dist* is the standard **Euclidean distance** between two points in Euclidean space, $c_i$ is the mean of cluster $C_i$ and x is a point belonging to cluster $C_i$. A small SSE means the generated clusters truly represent the points in the cluster. Therefore when selecting between two different set of clusters, we select one that minimizes the total SSE.

# Chapter 4

# Implementation

In this chapter, we present the implementation of bucket clustering algorithm that was presented in chapter 3 and use it to determine the value of k in K-means algorithm. Java programming language is used for the implementation. A nice and user friendly graphical interface is built on the top of the program for better user interaction.

## 4.1 GUI Description

The main graphical user interface, created using the JFrame object from javax.swing package, is divided into five panels: top, left, center, right and bottom panels as shown in Figure 4.1. The top panel contains the menu bar which handles the file operations and program termination.The left panel contains different checkboxes that allow user to perform operations like drawing nodes, editing nodes, showing the clusters, nudging and showing high low buckets. The center panel, being the main part of our GUI, displays the graphics for both input data and generated output. The right panel contains different buttons like clear canvas, refresh canvas, random sites etc and different textboxes like threshold, nudge of the mouse and number of points in the center panel. All the panels are extended from JPanel class of javax.swing package.

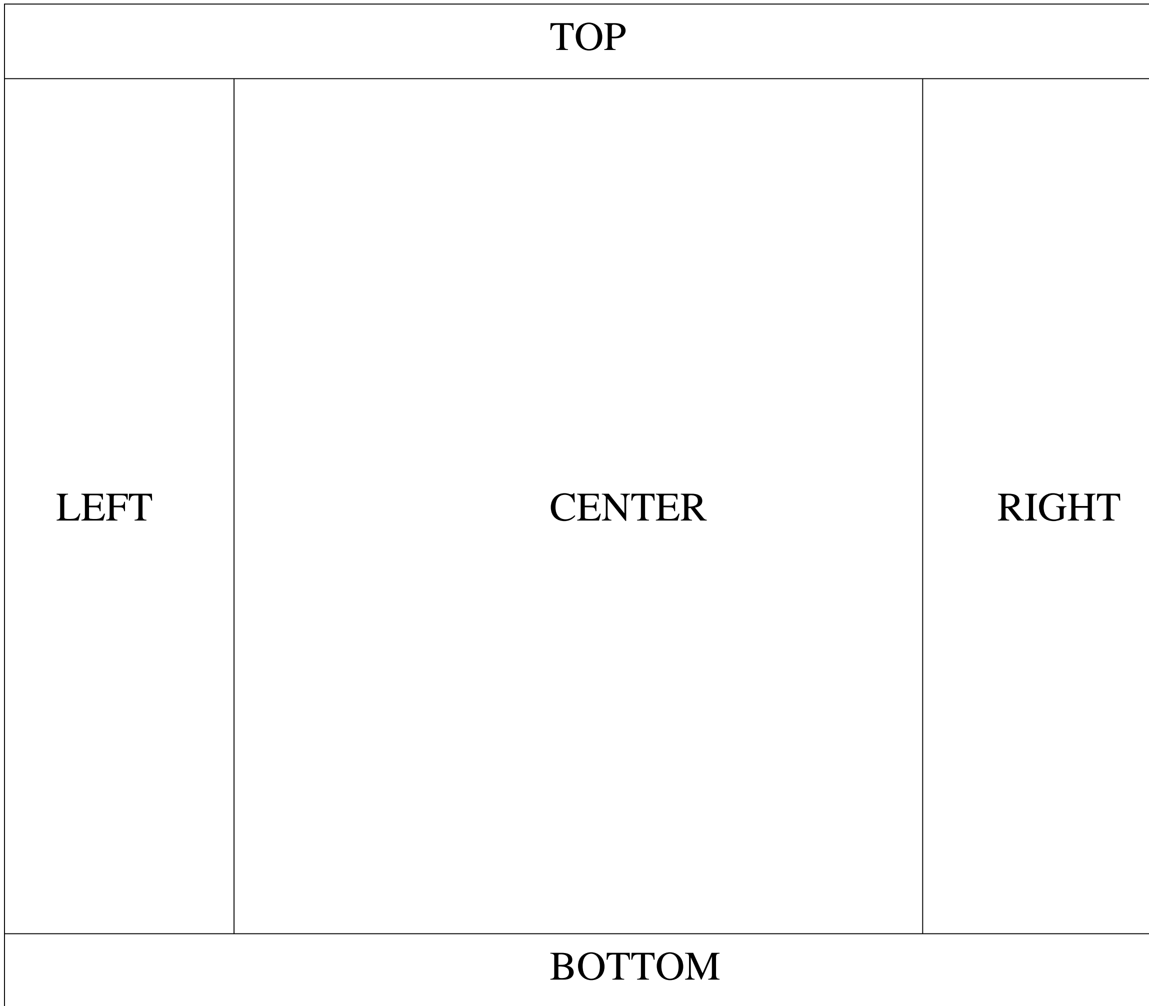


Figure 4.1: Graphical User Interface layout.

### 4.2 Interface Description

The actual graphical user interface of our program is shown in Figure 4.2. The top panel holds the file menu dropdown which allows users to open an existing file, save object data to file, and exit the application. A point can be drawn in the canvas by checking *Draw Vertex* checkbox and clicking the left button of the mouse. When the user clicks the left button of the mouse on the canvas, a small black-filled point is drawn. The corresponding $x$ and $y$ co-ordinates of the point are displayed on the Vertex Coordinates textbox located on the right panel. Figure 4.2 is a snap-shot from the program showing 20 vertices entered by a user via mouse clicks. *Edit Vertex* checkbox on the left panel can be used to edit the

position of the drawn point. When *Edit Vertex* checkbox is checked and the left button of the mouse is pressed and dragged, the point nearest to the cursor changes its position to the current position of the mouse cursor. A brief description of the functionalities of file menu items on the top panel, the check box items on the left panel and the buttons and textboxes on the right panel are listed in Table 4.1, Table 4.2, Table 4.3, and Table 4.4 respectively.

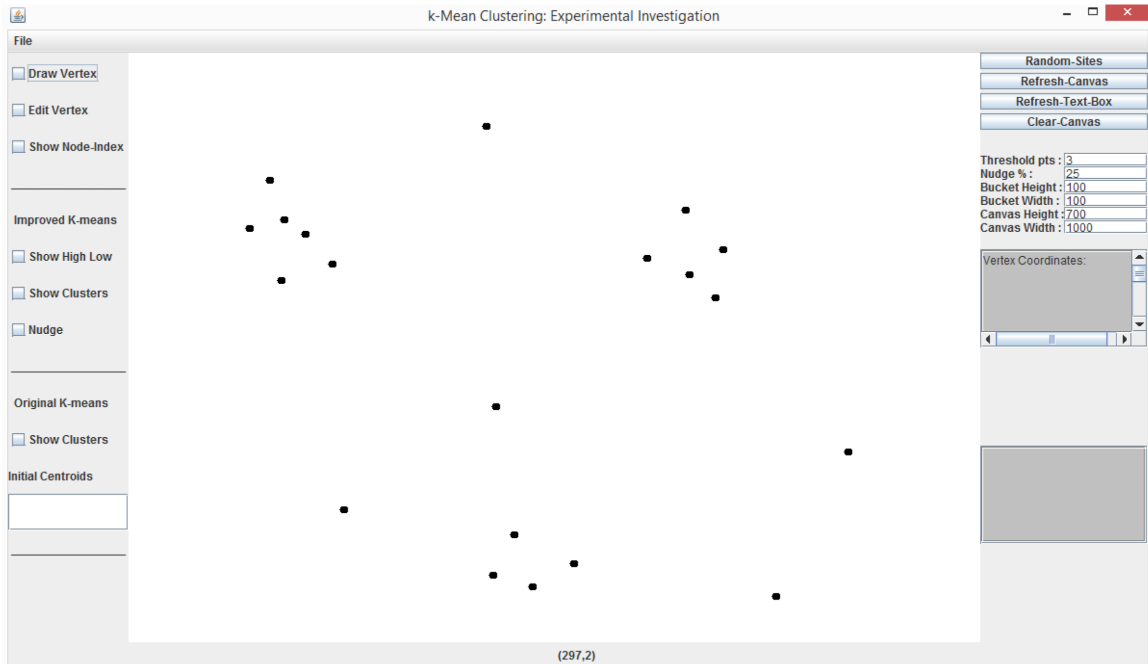


Figure 4.2: Actual Graphical User Interface

Table 4.1: File Menu Items Description.

| **S.N.** | **File Menu Items** | **Functionalities** |
|---|---|---|
| 1 | Read File | Allows user to open an existing file. |
| 2 | Save File | Allows user to save the diagram to a file. |
| 3 | Exit | Exits the application |

Table 4.2: Checkbox Description.

| S.N. | Check boxes | Functionalities |
|---|---|---|
| 1 | Draw Vertex | Allows users to draw vertices on the canvas. |
| 2 | Edit Vertex | Allows users to edit previously drawn vertices. |
| 3 | Show High Low | Divides the canvas into buckets and shows high low buckets. |
| 4 | Show Clusters | Displays the clusters for the given set of points on canvas. |
| 5 | Nudge | Expands the area of the cluster by a small amount to include points that are part of low buckets but are near the boundary of high buckets. |

Table 4.3: Button Description.

| S.N. | Buttons | Functionalities |
|---|---|---|
| 1 | Random Sites | Draws random set of points on the canvas |
| 2 | Refresh Canvas | Draws points on the canvas using vertex co-ordinates from Vertex Coordinates textbox |
| 3 | Refresh Textbox | Refreshes Vertex Coordinates text box with co-ordinates of current points on the canvas |
| 4 | Clear Canvas | Clears everything in the canvas |

Table 4.4: Textbox Description.

| S.N. | Textboxes | Functionalities |
|---|---|---|
| 1 | Threshold | Sets the threshold of points required for a bucket to be high |
| 2 | Nudge | Specifies the nudge percent to be applied when Nudge checkbox is checked |
| 3 | Bucket Width | Sets the width for a individual bucket |
| 4 | Bucket Height | Sets the height for a individual bucket |
| 5 | Canvas Width | Sets the width for the canvas |
| 6 | Canvas Height | Sets the height for the canvas |
| 7 | Vertex Coordinates | Displays the vertex co-ordinates of points on the canvas |
| 8 | Initial Centroids | Gets the initial centroids required for K-means algorithm |

## 4.3 Execution of Bucket Clustering algorithm

The bucket clustering algorithm can be executed once the user draws a set of points on the canvas either by mouse clicks or reads them from an existing file. When the *Show Clusters* checkbox is checked and mouse is moved on the canvas, the canvas is divided into buckets taking the height and width of buckets from two respective textboxes on the right panel. Then the points are assigned to respective buckets based on their $x$ and $y$ co-ordinates, and

high and low buckets are determined. Once high and low buckets are calculated, the high buckets are aggregated, whenever and wherever possible, to create a cluster. Figure 4.4 shows the clusters obtained by using the bucket clustering algorithm for the set of input points in Figure 4.3. The value of k is displayed below the Vertex Coordinates textbox on the right panel. The points in clusters are colored so as to make them different from other points. Also different colors are assigned to points from different clusters. The cluster centers are marked with 'X'. Figure 4.5 shows the clusters after nudging is applied and Figure 4.6 shows buckets with high and low labels.

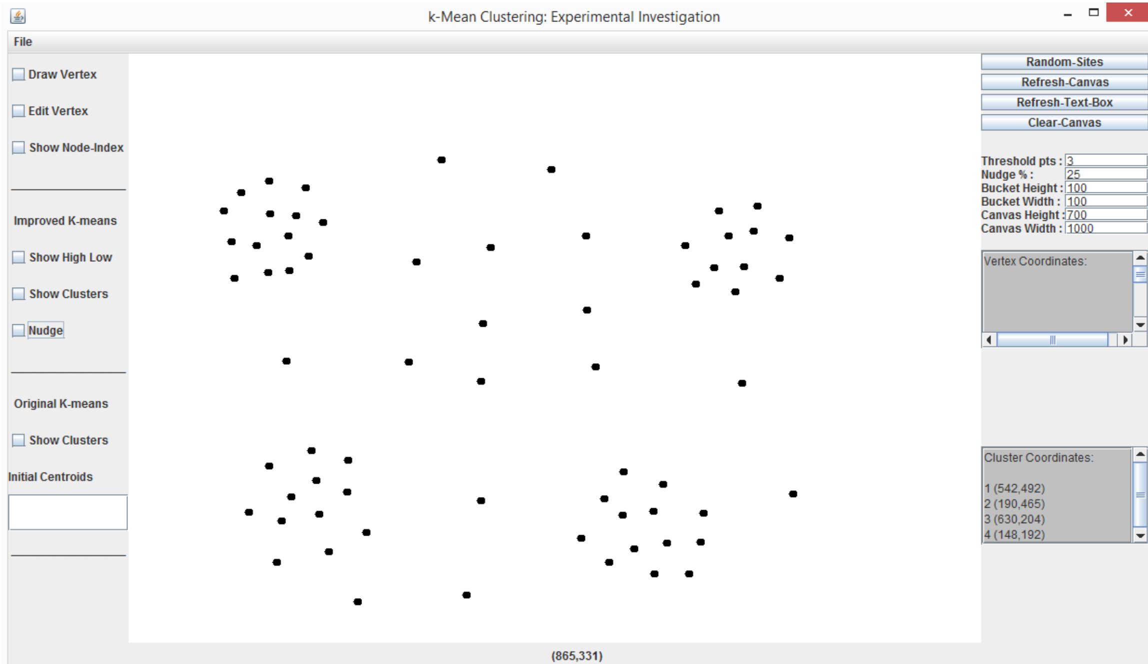


Figure 4.3: Set of input points

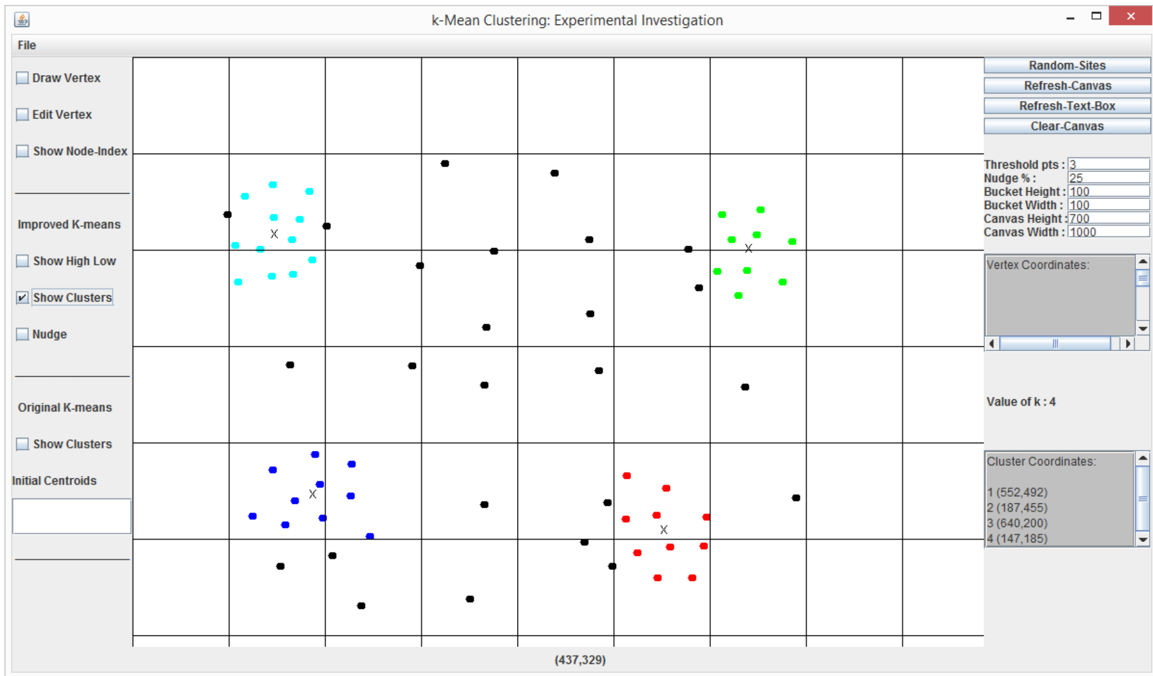


Figure 4.4: Cluster obtained using bucket clustering algorithm

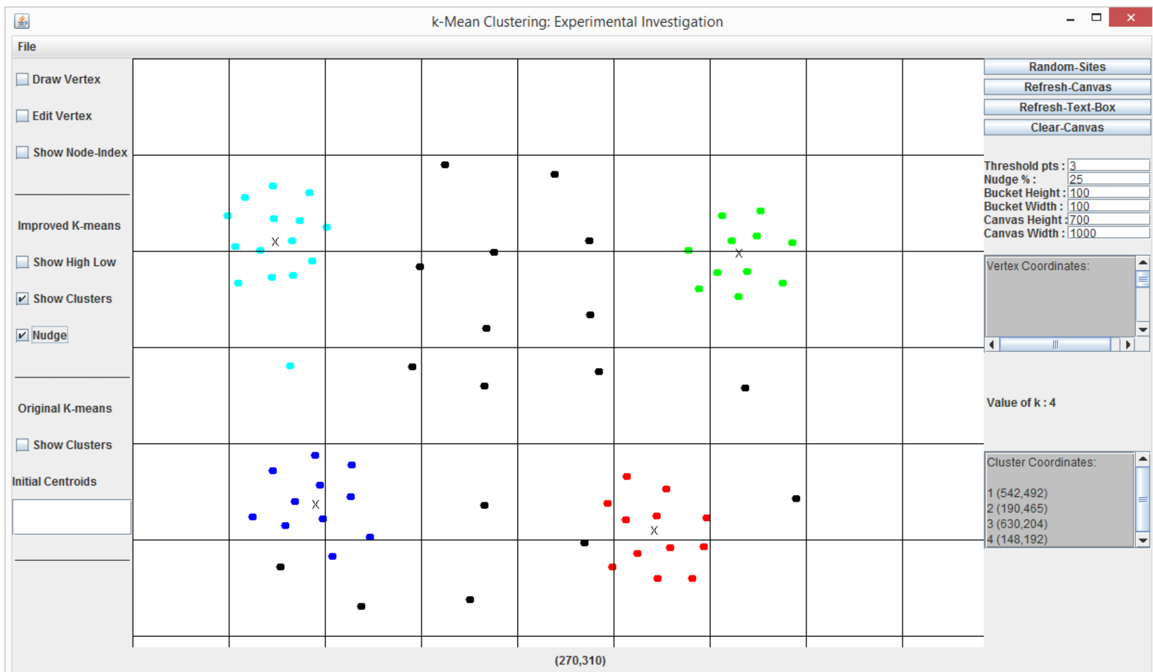


Figure 4.5: Clusters after Nudging

Figure 4.6: Labeling buckets as high and low

## 4.4 Results and statistics

We generated different examples with varying number of clusters and points to test the performance of the bucketing algorithm. We used *SSE* technique as described in Chapter 3 Section 3.6 for this purpose. We calculated SSE for clusters generated using both the standard K-means algorithm and bucketing algorithm. The cluster centers are marked with 'X' whereas the initial centroids are marked with '+'. Figure 4.7 to Figure 4.18 show the results of experimental investigations whereas the details of these experimental results are shown in Table 4.6 to Table 4.11.

Table 4.5: Dataset result Mapping

| Dataset No. | No. of points | Result Table | Result Figures |
|---|---|---|---|
| 1 | 72 | Table 4.6 | Figure 4.7 - 4.8 |
| 2 | 100 | Table 4.7 | Figure 4.9- 4.10 |
| 3 | 66 | Table 4.8 | Figure 4.11 - 4.12 |
| 4 | 208 | Table 4.9 | Figure 4.13 - 4.14 |
| 5 | 551 | Table 4.10 | Figure 4.15 - 4.16 |
| 6 | 400 | Table 4.11 | Figure 4.17 - 4.18 |
| 7 | 1000 | Table 4.12 | Figure 4.19 - 4.20 |

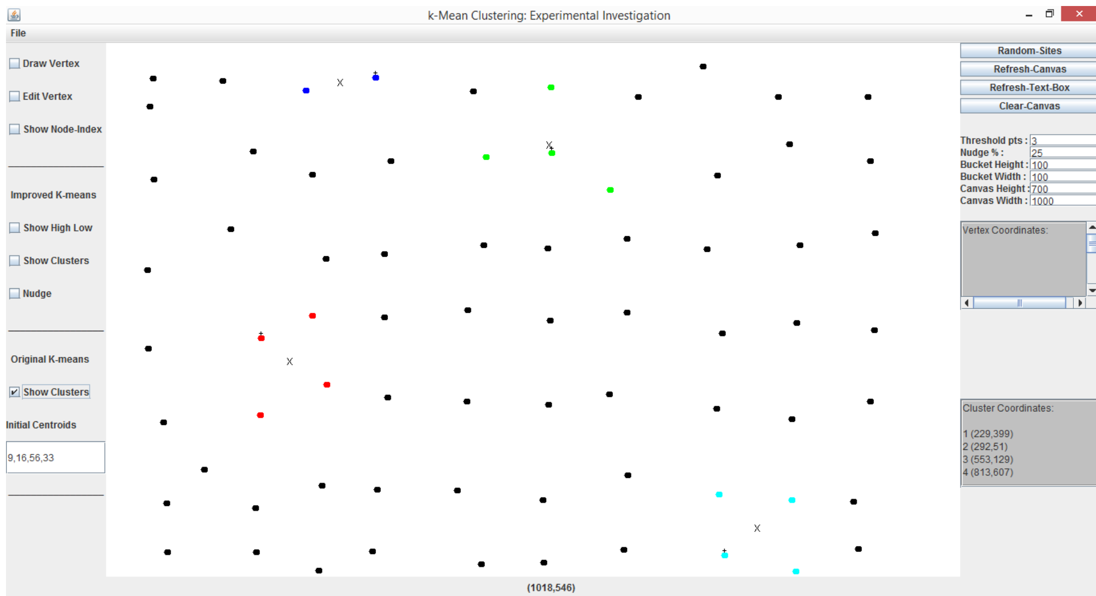


Figure 4.7: Dataset 1 with initial centroids at point 9, 16, 56, 33

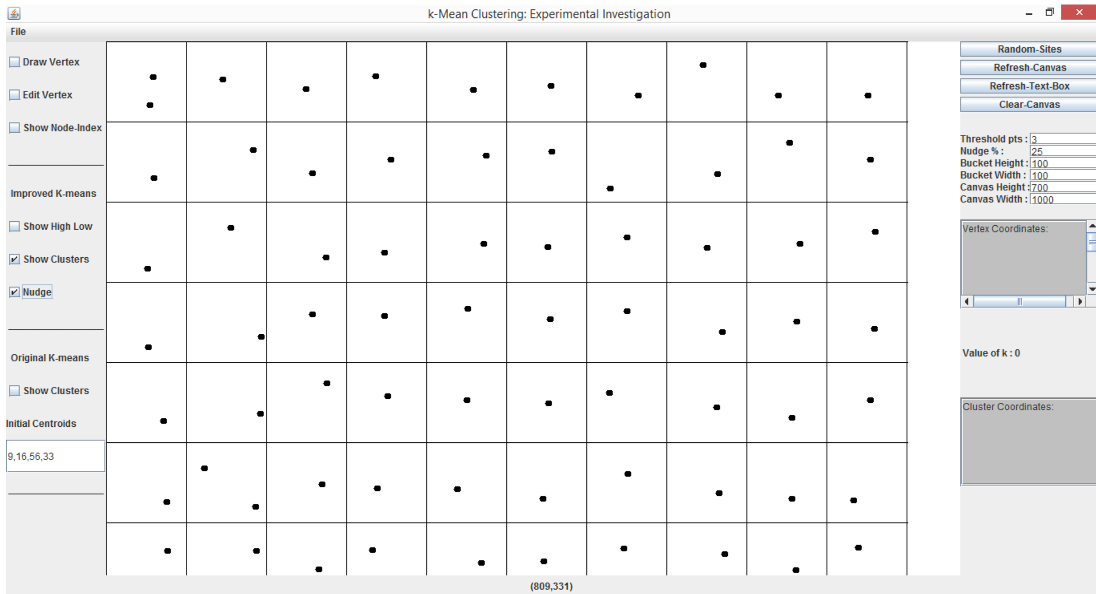


Figure 4.8: Dataset 1 after executing bucketing algorithm and nudging

Figure 4.9: Dataset 2 with initial centroids at point 50, 7, 28

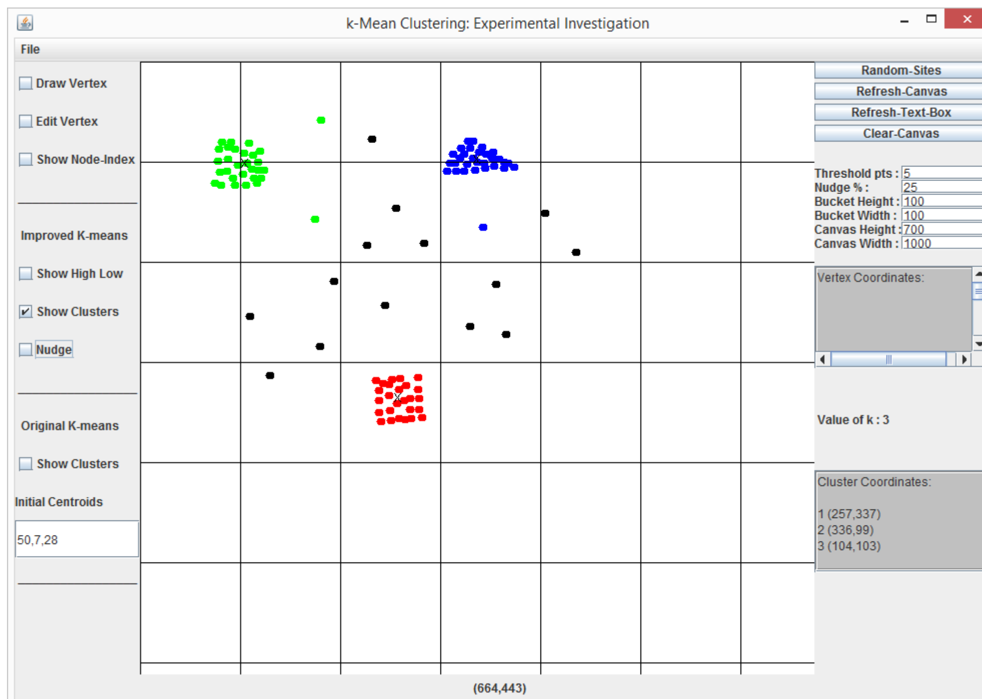


Figure 4.10: Dataset 2 after executing bucketing algorithm

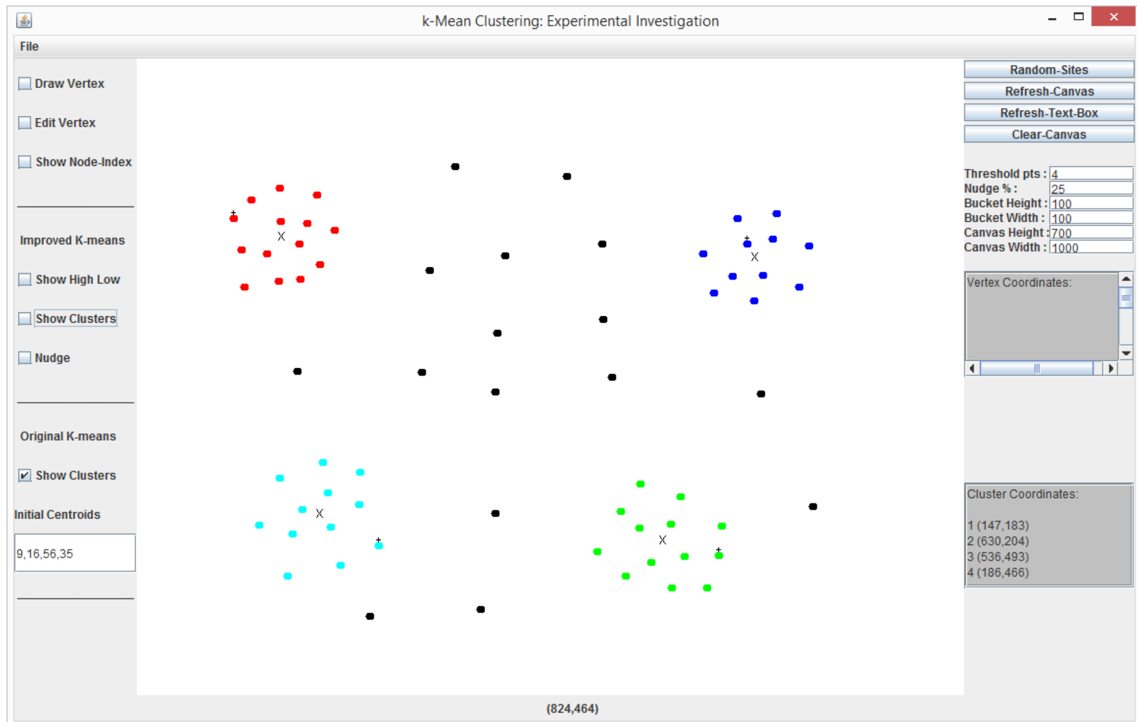


Figure 4.11: Dataset 3 with initial centroids at point 9, 16, 56, 35

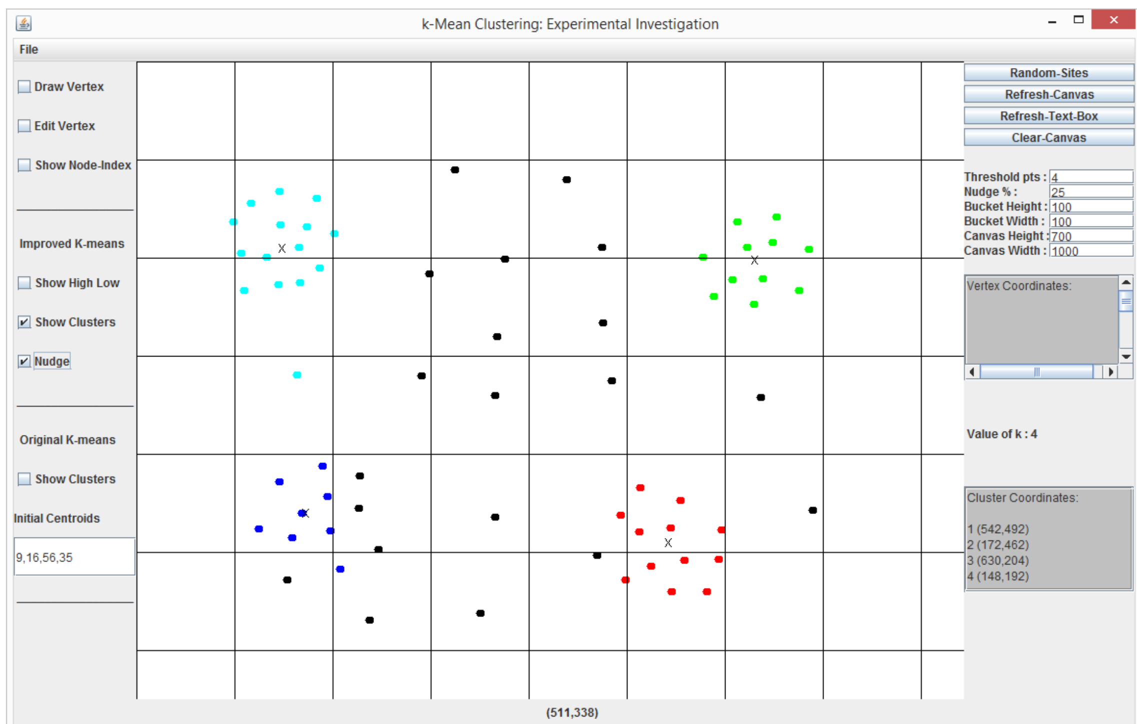


Figure 4.12: Dataset 3 after executing bucketing algorithm and nudging

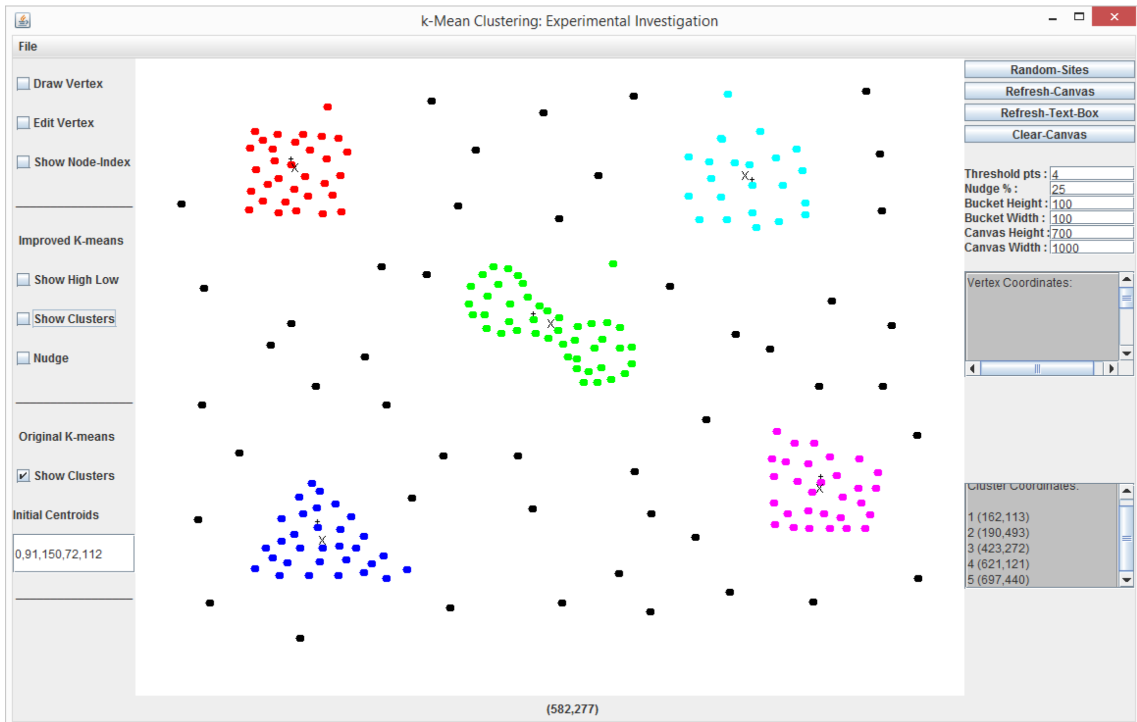


Figure 4.13: Dataset 4 with initial centroids at point 0, 91, 150, 72, 112

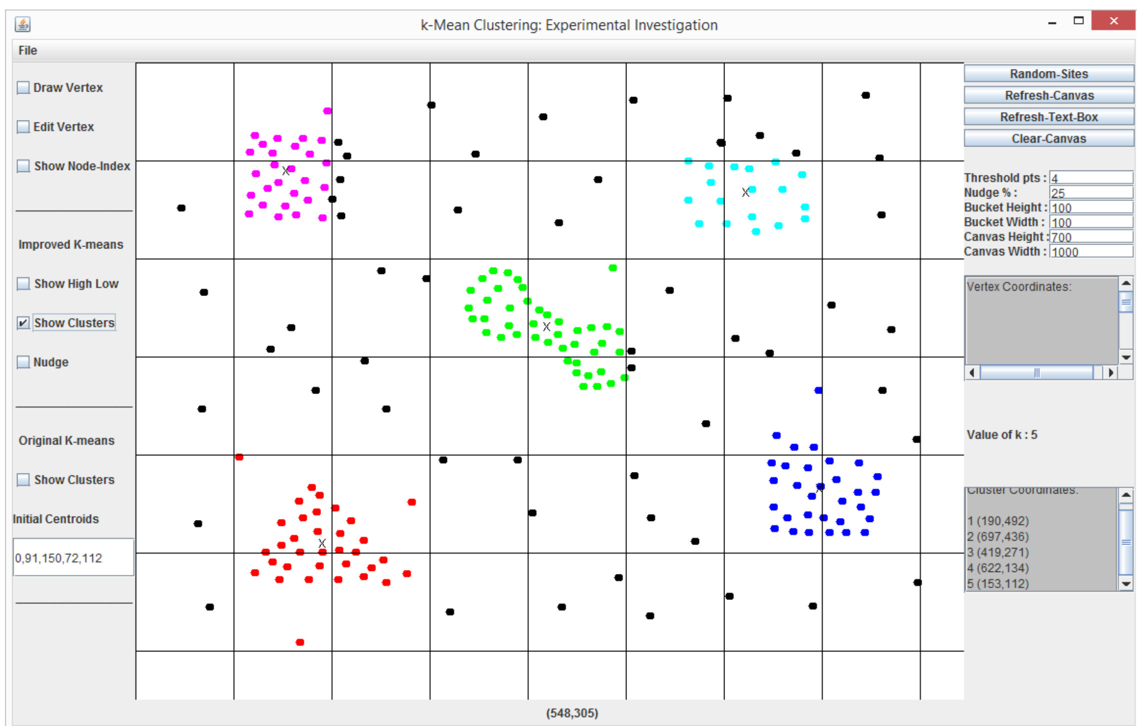


Figure 4.14: Dataset 4 after executing bucketing algorithm

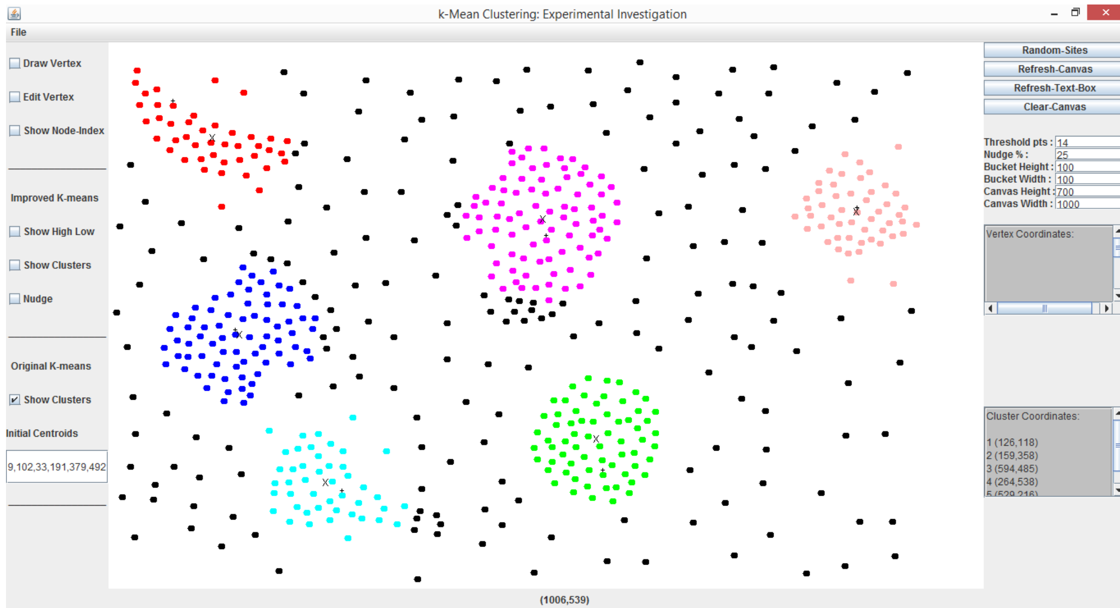


Figure 4.15: Dataset 5 with initial centroids at point 9, 102, 33, 191, 379, 492

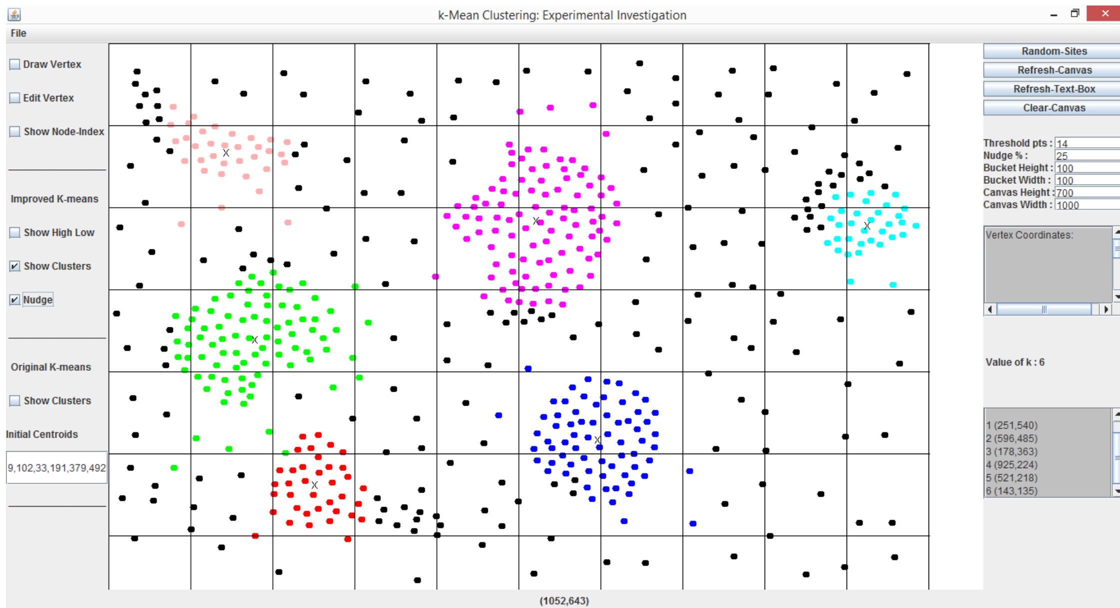


Figure 4.16: Dataset 5 after executing bucketing algorithm and nudging

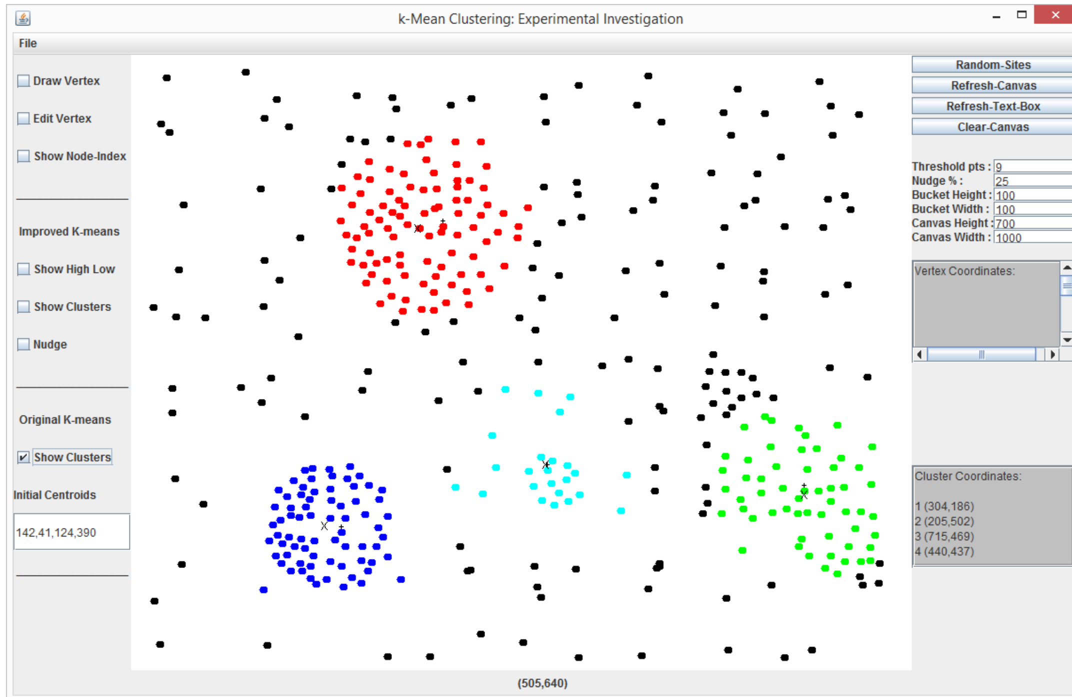


Figure 4.17: Dataset 6 with initial centroids at point 168, 55, 104, 392

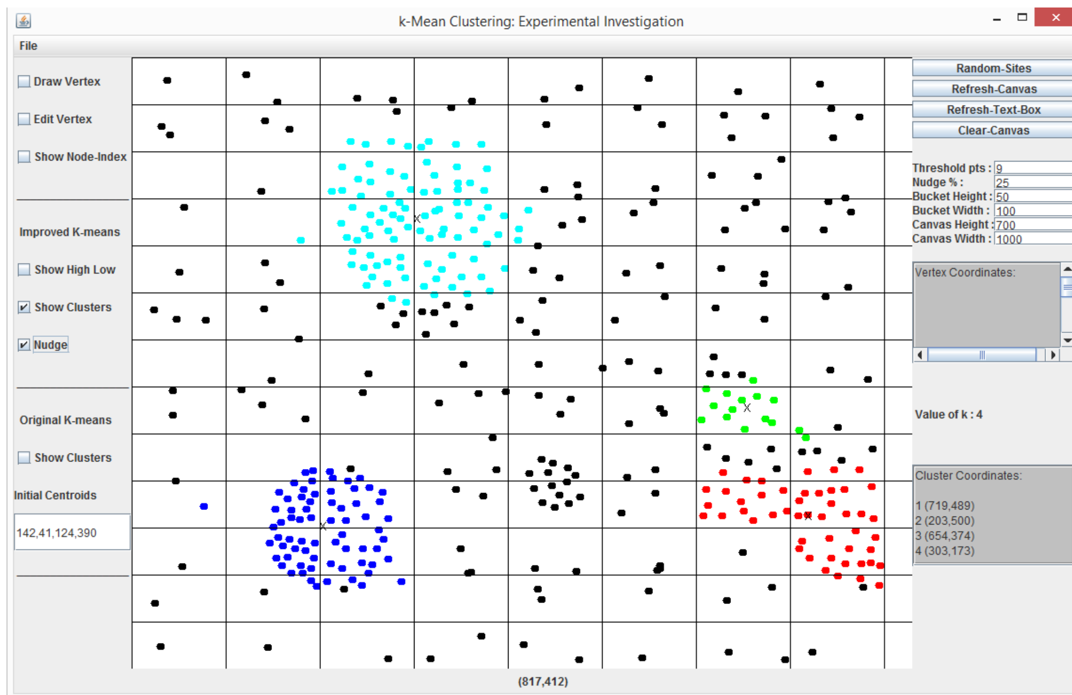


Figure 4.18: Dataset 6 after executing bucketing algorithm and nudging

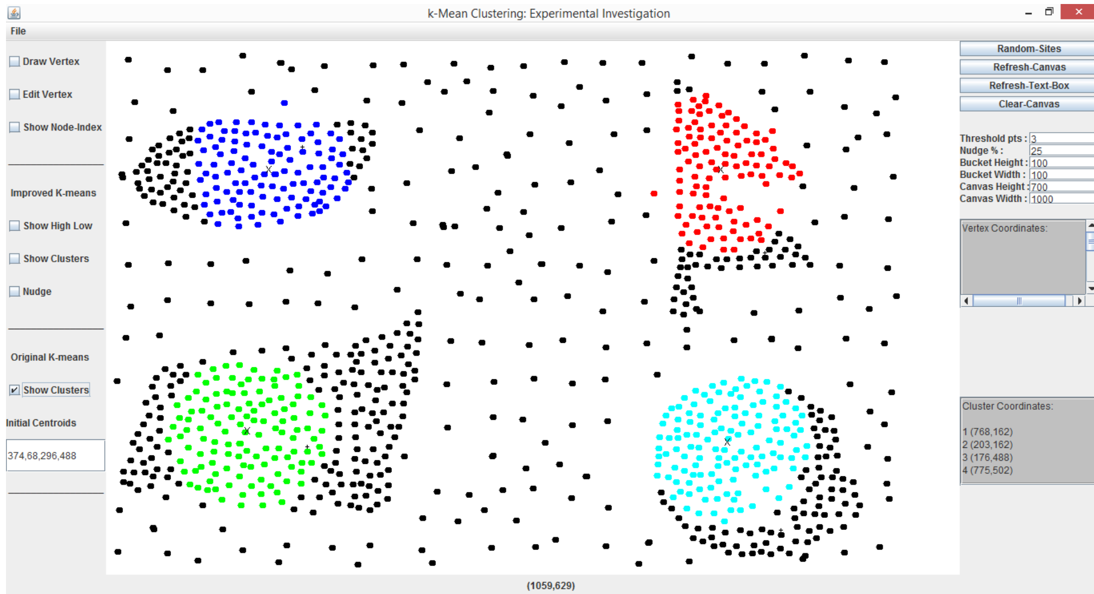


Figure 4.19: Dataset 7 with initial centroids at point 374, 68, 296, 488

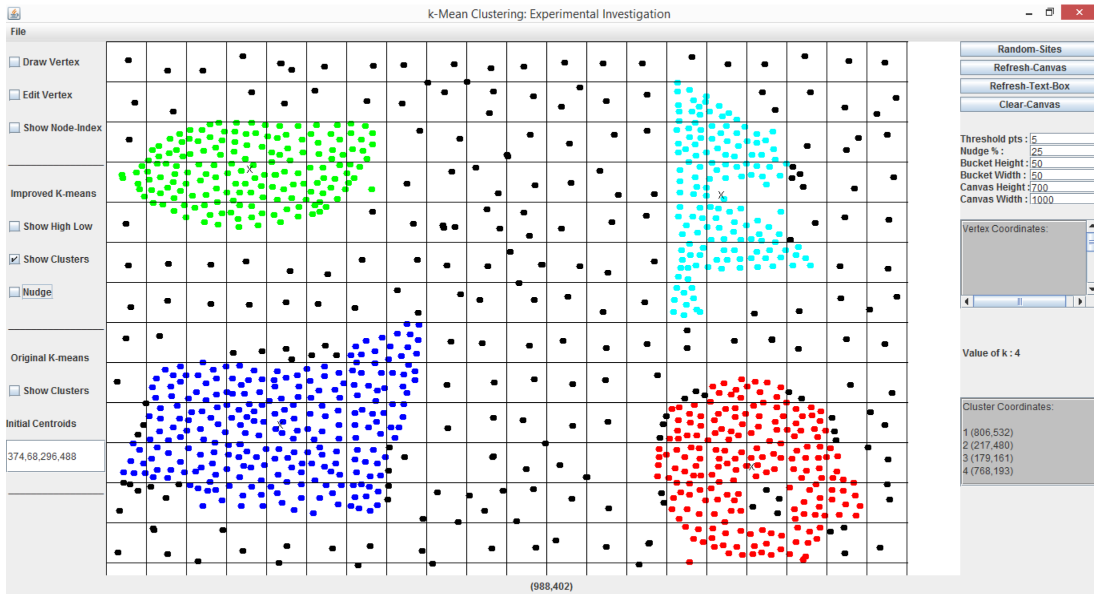


Figure 4.20: Dataset 7 after executing bucketing algorithm

Table 4.6: Dataset 1 Experimental results

| **Method** | **Initial Centroids** | **Avg. SSE** | **Total Clusters** | **Threshold pts.** |
|---|---|---|---|---|
| K-Means | 9, 16, 56, 33 | 50681 | 4 | |
| K-Means | 9, 16, 56, 35 | 40638.5 | 4 | |
| Bucketing | | 0 | 0 | 3 |
| Bucketing with Nudging | | 0 | 0 | 3 |

Table 4.7: Dataset 2 Experimental results

| **Method** | **Initial Centroids** | **Avg. SSE** | **Total Clusters** | **Threshold pts.** |
|---|---|---|---|---|
| K-Means | 10, 2, 26 | 85490.5 | 3 | |
| K-Means | 50, 7, 28 | 88399.5 | 3 | |
| Bucketing | | 55138 | 3 | 5 |
| Bucketing with Nudging | | 114537 | 3 | 5 |

Table 4.8: Dataset 3 Experimental results

| **Method** | **Initial Centroids** | **Avg. SSE** | **Total Clusters** | **Threshold pts.** |
|---|---|---|---|---|
| K-Means | 9, 16, 56, 35 | 109076 | 4 | |
| K-Means | 9, 16, 56, 33 | 103137 | 4 | |
| Bucketing | | 65265 | 4 | 4 |
| Bucketing with Nudging | | 107791 | 4 | 4 |

Table 4.9: Dataset 4 Experimental results

| **Method** | **Initial Centroids** | **Avg. SSE** | **Total Clusters** | **Threshold pts.** |
|---|---|---|---|---|
| K-Means | 0, 91, 150, 72, 112 | 399204 | 5 | |
| K-Means | 2, 93, 153, 63, 57 | 403163 | 5 | |
| Bucketing | | 403710 | 5 | 4 |
| Bucketing with Nudging | | 652767 | 5 | 4 |

Table 4.10: Dataset 5 Experimental results

| **Method** | **Initial Centroids** | **Avg. SSE** | **Total Clusters** | **Threshold pts.** |
|---|---|---|---|---|
| K-Means | 9, 102, 33, 191, 379, 492 | 1319661 | 6 | |
| K-Means | 335, 115, 30, 213, 379, 490 | 1394179 | 6 | |
| Bucketing | | 1072883 | 6 | 14 |
| Bucketing with Nudging | | 1952540 | 6 | 14 |

Table 4.11: Dataset 6 Experimental results

| **Method** | **Initial Centroids** | **Avg. SSE** | **Total Clusters** | **Threshold pts.** |
|---|---|---|---|---|
| K-Means | 142, 41, 124, 390 | 901703.25 | 4 | |
| K-Means | 168, 55, 104, 392 | 907126.25 | 4 | |
| Bucketing | | 24114140 | 1 | 4 |
| Bucketing with Nudging | | 24566045 | 1 | 4 |
| Bucketing | | 1279606 | 4 | 5 |
| Bucketing with Nudging | | 1595448 | 4 | 5 |
| Bucketing | | 599789 | 4 | 9 |
| Bucketing with Nudging | | 881676 | 4 | 9 |
| Bucketing | | 373290 | 3 | 13 |
| Bucketing with Nudging | | 592832 | 3 | 13 |
| Bucketing | | 122360 | 2 | 15 |
| Bucketing with Nudging | | 282545 | 2 | 15 |
| Bucketing | | 15715 | 1 | 17 |
| Bucketing with Nudging | | 68559 | 1 | 17 |
| Bucketing with Nudging | | 0 | 0 | 18 |

Table 4.12: Dataset 7 Experimental results

| **Method** | **Initial Centroids** | **Avg. SSE** | **Total Clusters** | **Threshold pts.** |
|---|---|---|---|---|
| K-Means | 374, 68, 296, 488 | 2102307 | 4 | |
| K-Means | 316, 29, 169, 446 | 2146726 | 4 | |
| Bucketing | | 7165562 | 4 | 5 |
| Bucketing with Nudging | | 7945554 | 4 | 5 |

From the experimental results it is clear that when the threshold points are carefully selected, the clusters obtained using the bucketing algorithm, in most cases, have either less or almost equal SSE compared to the standard K-means algorithm. Due to the wrong selection of threshold points, in some cases, the SSE obtained from the bucketing algorithm is higher than the standard K-means as in Dataset 6. Overall, the bucketing algorithm provides almost the same or better SSE compared to original K-means. In addition, bucketing algorithm removes the necessity of providing the number of clusters at the beginning.

# Chapter 5

# Conclusion and Discussion

We presented a review of important existing approaches for identifying clusters in Euclidean space. In particular we described a critical evaluation of the hierarchical method for recognizing clusters by using bottom-up nesting. We also presented a detailed examination of the most popular cluster constructing algorithm called the K-mean algorithm. We articulated one of the main difficulties of the K-mean algorithm which is the estimation of the number of clusters $k$. In most variations of the K-mean algorithm the value of $k$ is taken as part of the input. This motivated us to seek ways of estimating the value of $k$ efficiently.

We proposed two main methods for estimating the values of $k$ for points distributed in two dimensions. The first method we presented is based on using a bucketing technique to approximately identify the number of clusters. The buckets used are the rectangular boxes obtained by embedding an orthogonal grid on the 2-d Euclidean space. The algorithm is easy to understand and implement. One of the benefits of the bucketing method is that large size data can be sampled in the bucket to substantially reduce the size of input data. Existing K-mean algorithms can be used on the reduced dataset.

In the g-hop method, we developed algorithms for determining the number of input points $m(g, p_i)$ within distance $g$ from a given test point $p_i$. We first considered the straightforward method of estimating $m(q, p_i)$ based on distance-sorting, which takes $O(N^2 logN)$ time in total. We then presented a faster Voronoi based algorithm for computing $m(q, p_i)$. This algorithm runs in $O(NlogN)$ time.

We presented an experimental investigation of clustering algorithms by implementing

a prototype program in Java, supporting a friendly graphical user interface. The experimental investigation includes the standard K-mean algorithm and bucketing method. The experimental results show that the bucketing method is quite effective in estimating cluster population in 2-d.

Several extensions and generalizations of the technique proposed in this thesis can be suggested. The bucketing method can be generalized in a straightforward way to three and higher dimensions. The rectangular buckets in two dimensions become rectangular prisms in three dimensions. The point inclusion test for rectangles can be modified to a point inclusion test for rectangular prisms. It would be very interesting to perform an experimental investigation in three dimensions.

We could have implemented and investigated *g-hop* method but, due to time constraints, we were unable to do so. It would be worth implementing and investigating *g-hop* method.

Another extension of the investigation would be its generalization to big-data. In situation when all data cannot be loaded into RAM, how can we use the locality hashing paradigm [4] for estimating cluster populations in two and three dimensions this would be worth investigating.

# Bibliography

[1] Berg, Mark de, Mark van Krevald, Mark Overmars, and Otfried Schwarzkopf. *Computational Geometry: Algorithms and Applications*. 2nd ed. Berlin: Springer, 2000.

[2] Cormen, Thomas H., Charles E. Lieserson, Ronald L. Rivest, and Cliford Stein. *Introduction to Algorithms*. 3rd ed. Cambridge, Mass.: MIT Press, 2009.

[3] Guha, Sudipto, Rajeev Rastogi, and Kyuseok Shim. "CURE: An Efficient Clustering Algorithm for Large Databases." *Information Systems* 26, no. 2 (2001): 35-58. MIT Press, 2009

[4] Indyk, Piotr, and Rajeev Motwani. "Approximate Nearest Neighbors: Towards Removing the Curse of Dimensionality." *Proceedings of STOC*, 1998, 604-13.

[5] Jolliffe, I. T. *Principal Component Analysis*. 2nd ed. New York: Springer, 2002.

[6] Leskovec, Jure, Jeffrey D. Ullman, and Anand Rajaraman.*Mining of Massive Datasets*. New York, N.Y.: Cambridge University Press, 2014.

[7] Lloyd, S. P. "Least Square Quantization in PCM." *IEEE Transaction on Information Theory*, 1982, 129-37.

[8] Rourke, Joseph. *Computational Geometry in C*. 2nd ed. Cambridge, UK: Cambridge University Press, 1998.

[9] Tan, Pang-Ning, Michael Steinbach, and Vipin Kumar. *Introduction to Data Mining*. Boston: Pearson Addison Wesley, 2005.

# Curriculum Vitae

Graduate College

University of Nevada, Las Vegas

Sanjeev K C

Degrees:

Bachelor of Computer Engineering 2010

Tribhuvan University, Institute of Engineering, Pulchowk Campus

Thesis Title: Efficient Estimation of Cluster Population

Thesis Examination Committee:

Chairperson, Dr. Laxmi Gewali, Ph.D.

Committee Member, Dr. Ajoy Datta, Ph.D.

Committee Member, Dr. John Minor, Ph.D.

Graduate Faculty Representative, Dr. Henry Selvaraj, Ph.D.