Cyclic Exchange Neighborhood Search Technique for the K-means Clustering Problem

by

Nattavude Thirathon

S.B., Electrical Engineering and Computer Science
Massachusetts Institute of Technology, 2003

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

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2004

Author .....
Department of Electrical Engineering and Computer Science
January 30, 2004

Certified by .....
James B. Orlin
Professor of Management Science and Operations Research
Thesis Supervisor

Certified by .....
Nitin Patel
Professor of Operations Research
Thesis Supervisor

Accepted by .....
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Cyclic Exchange Neighborhood Search Technique for the K-means Clustering Problem

by

Nattavude Thirathon

Submitted to the Department of Electrical Engineering and Computer Science
on January 30, 2004, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Cyclic Exchange is an application of the cyclic transfers neighborhood search technique for the k-means clustering problem. Neighbors of a feasible solution are obtained by moving points between clusters in a cycle. This method attempts to improve local minima obtained by the well-known Lloyd's algorithm. Although the results did not establish usefulness of Cyclic Exchange, our experiments reveal some insights on the k-means clustering and Lloyd's algorithm. While Lloyd's algorithm finds the best local optimum within a thousand iterations for most datasets, it repeatedly finds better local minima after several thousand iterations for some other datasets. For the latter case, Cyclic Exchange also finds better solutions than Lloyd's algorihtm. Although we are unable to identify the features that lead Cyclic Exchange to perform better, our results verify the robustness of Lloyd's algorithm in most datasets.

Thesis Supervisor: James B. Orlin
Title: Professor of Management Science and Operations Research

Thesis Supervisor: Nitin Patel
Title: Professor of Operations Research

Acknowledgments

This thesis is done with great help from my co-supervisors: Professor Jim Orlin and Professor Nitin Patel. Not only did they teach me technical knowledge, but they also guided me on how to do research and live a graduate student life. My transition from an undergraduate student to a graduate student would not be as smooth without their kindness.

I also would like to thank Ozlem Ergun (now a professor at at Georgia Tech). She introduced me to the Operation Research Center in my second semester at MIT. I worked for her as a undergraduate research assistant on neighborhood search techniques for traveling salesman problem and sequential ordering problem. Her research work (under supervision of Professor Orlin) ignited my interests in operations research and resulted in me coming back to do my master thesis at ORC.

Lastly, I am grateful to papa, mama, and sister Ann. Nothing rescues tireness, confusion, and long nights better than a warm family back home.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Clustering is a technique for classifying patterns in data. The data are grouped into *clusters* such that some criteria are optimized. Clustering is applicable in many domains such as data mining, knowledge discovery, vector quantization, data compression, pattern recognition, and pattern classification. Each application domain has different kinds of data, cluster definitions, and optimization criteria. Optimization criteria are designed such that the optimal clustering is meaningful for the applications.

Clustering is a combinatorially difficult problem. Although researchers have devised many algorithms, there is still no known method that deterministically finds an exact optimal solution in polynomial time for most formulations. Thus, the clustering problems are usually approached by heuristics — relatively fast methods that find near-optimal solutions. [7] gives an overview of many clustering techniques, most of which are heuristics that find suboptimal but good solutions in limited time.

This thesis explores a heuristic for solving a specific class of clustering problem — *the k-means clustering.* In the k-means clustering, data are represented by n points in d-dimensional space. Given a positive integer k for the number of clusters, the problem is to find a set of k *centers* such that the sum of squared distances from each point to the nearest center is minimized. Equivalently, the sum of squared distances from cluster centers can be viewed as total scaled variance of each cluster.

A conventional and very popular algorithm for solving the k-means clustering is

Lloyd's algorithm, which is usually referred to as the *k-means algorithm.* Lloyd's algorithm has an advantage in its simplicity. It uses a property of any locally optimal solution that each data point has to belong to the cluster whose center is the nearest. Beginning with a random solution, Lloyd's algorithm converges quickly to a local optimum. Nonetheless, the k-means clustering has tremendous number of local optima, and Lloyd's algorithm can converge to bad ones, depending on the starting point. Our algorithm tries to overcome this limitation of Lloyd's algorithm.

The technique that we propose is an application of a wide class of optimization heuristic known as neighborhood search or local search. Neighborhood search techniques try to improve the quality of solution by searching the *neighborhood* of the current solution. The neigborhood is generally a set of feasible solutions that can be obtained by changing small parts of the current solution. The algorithm searches the neighborhood for a better solution and uses it as a new current solution. These two steps are repeated until the neighborhood does not contain any better solutions. The definition of neighborhood and the search technique are important factors that affect the quality of final solution and time used by the neighborhood search.

This thesis is a detailed report of our work in developing the Cyclic Exchange algorithm. We first give a background on the k-means clustering and neighborhood search, upon which Cyclic Exchange is developed. The detailed descriptions of the algorithm and software development are then presented. This software is used as a back-end for our experiments in the subsequent chapter. We conducted three experimental studies in order to improve the performance of Cyclic Exchange. Each experiment reveals insights on the k-means clustering and Lloyd's algorithm.

Chapter 2

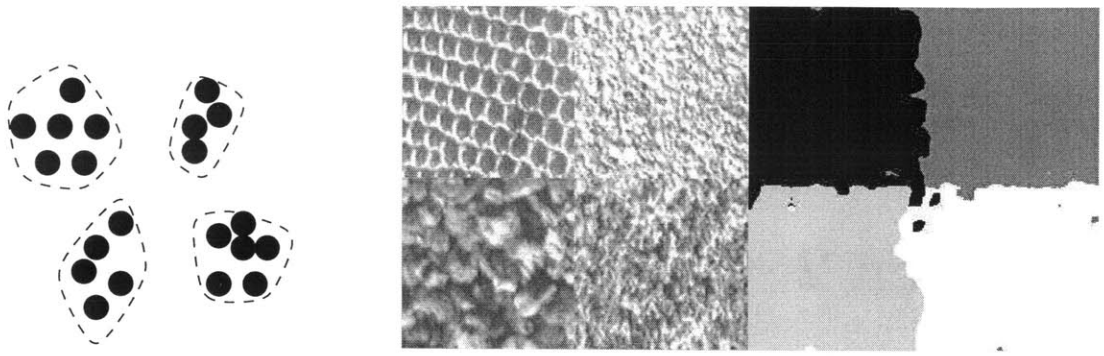
Background

2.1 Data Clustering

The problems of data clustering arise in many contexts in which a collection of data need to be divided into several groups or clusters. Generally, the data in the same clusters are more similar to one another than to the data in different clusters. The process of clustering may either reveal underlying structures of the data or confirm the hypothesized model of the data. Two examples of data clustering are shown in Figure 2-1. In Figure 2-1(a), the data are represented by points in two dimensional space, and each cluster consists of points that are close together. Figure 2-1(b) from [7] shows an application of data clustering in image segmentation.

Another field of data analysis, discrimination analysis, is similar to but fundamentally different from data clustering. [7] categorizes data clustering as unsupervised classification, and discrimination analysis as supervised classification. In unsupervised classification, unlabeled data are grouped into some meaningful clusters. In contrast, in supervised classification, a collection of previously clustered data is given, and the problem is to determine which cluster a new data should belong to.

There are generally three tasks in data clustering problems. First, raw data from measurements or observations are to be represented in a convenient way. We then have to define the similarity between data, i.e. the definition of clusters. Lastly, a suitable clustering algorithm is applied. All three tasks are interrelated and need to



(a) (b)

Figure 2-1: Data Clustering Examples: (a) Distance-based clustering. (b) Clustering in image segmentation application [7] — original image and segmented image.

be carefully designed in order to efficiently obtain useful information from clustering.

Data can be categorized roughly in to two types: quantitative and qualitative. Quantitative data usually come from physical measurements, e.g. positions, brightness, and number of elements. They can be discrete values (integers) or continuous values (real numbers). The use of computer to solve clustering problems, however, restricts the continuous-value data to only rational numbers, which can be represented by finite digits in computer memory. Since measurements usually involve more than a single number at a time, e.g. a position in three dimensional space consists of three numbers, quantitative data are usually represented by vectors or points in multidimensional space.

Qualitative data come from abstract concepts (e.g. colors and styles), from qualitative description of otherwise measureable concepts (e.g. good/bad, high/low, and bright/dim), or from classification (e.g. meat/vegetable/fruits). [5] proposed a generalized method of representing quantitative data. Qualitative data clustering is beyond the scope of the k-means clustering and is mentioned here only for completeness.

The notion of similarity is central to data clustering as it defines which data should fall into the same cluster. In the case of quantitative data which are represented by points in multidimensional space, similarity is usually defined as the distance between two data points. The most popular distance measure is the Euclidean distance, which can be interpreted is proximity between points, in a usual sense. Other kinds of

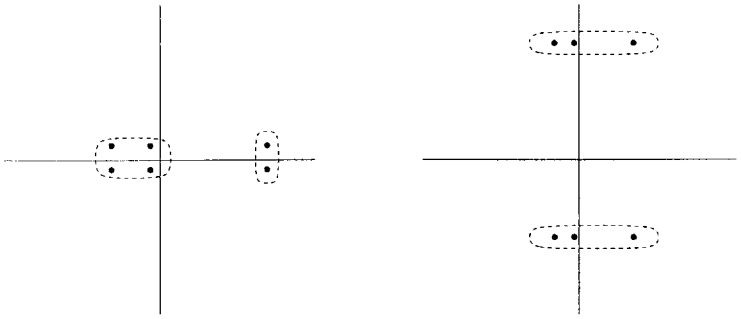


Figure 2-2: Effects of Units of Measurement on Distance-based Clustering: The horizontal unit in (b) is half of those in (a), and the vertical unit in (b) is eight times of those in (a).

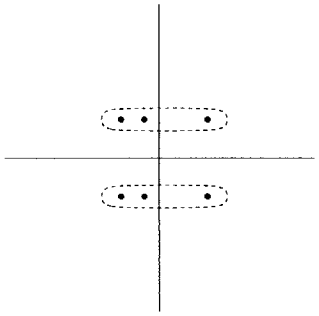


Figure 2-3: Z-score Normalization

distance measures include Manhattan distance and higher order norm, for example.

The use of distance as similarity function is subject to scaling in units of measurements. Figure 2-2 shows two sets of the same data measured in different units. For instance, the horizontal axis represents positions in one dimension, and the vertical axis represents brightness. If the data are grouped into two clusters, and the similarity is measured by the Euclidean distance, the optimal solutions are clearly different as shown.

Z-score normalization is a popular technique to overcome this scaling effect. The data is normalized separately for each dimension such that the mean equals to zero and the standard deviation equals to one. Both measurements in Figure 2-2 are normalized to the data in Figure 2-3. If normalization is done every time, the resulting clustering will be the same under all linear units of measurement.

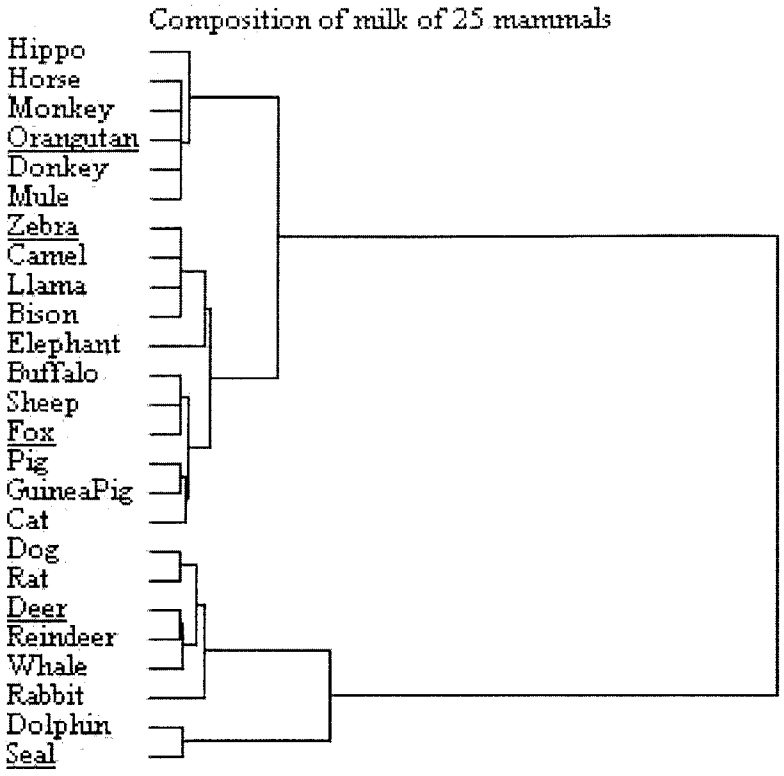


Figure 2-4: Heirachical Data Clustering [4]

Given the representation of data and the definition of similarity, the remaining task is to find or develop a suitable algorithm. Each algorithm differs in several aspects. [7] gives a taxonomy of data clustering algorithms. Some of the different aspects of algorithms are:

- Heirachical v.s. Partitional: A partitional approach is designed to cluster data into a given number of cluster, such as those in Figure 2-1. In contrast, heirachical approach produces a nested structure of clusters. Figure 2-4 shows an example of results for heirachical clustering of a sample dataset in [4]. By examining this nested structure at different level, we can obtain different number of clusters. Determining the number of clusters for partitional algorithms is a research problem in its own and much depends on the application domains.

- Hard v.s. Fuzzy: A hard clustering assigns each data point to one and only one cluster, while a fuzzy clustering assigns to each data point the degrees of

membership to several clusters.

- Deterministic v.s. Stochastic: A deterministic approach proceeds in a preditable manner, giving the same result everytime the program is run. A stochastic approach incorporates randomness into searching for the best solution. It may terminate differently each time the program is run.

- Incremental v.s. Non-incremental: An incremental approach gears toward large scale data clustering or online data clustering, where all data are not available at the same time. A non-incremental approach assumes that all of the data are readily available.

As we will see later, Cyclic Exchange algorithm and Lloyd's algorithm are partitional, hard, stochastic, and non-incremental.

2.2 K-means Clustering

2.2.1 Notations and Problem Definition

The input of k-means clustering problem is a set of n points in d-dimensional real space and a positive integer k, $2 \leq k \leq n$. The objective is to cluster these n points into k exhaustive and disjoint sets (i.e. hard clustering), such that the total sum of the squared distances from each point to its cluster center is minimized.

Denote the input points set by $\mathcal{X} = \{\mathbf{x}_1, \ldots, \mathbf{x}_n\}, \quad \mathbf{x}_i \in \Re^d, \quad \forall i : 1 \leq i \leq n$

Let each cluster be described by a set of index of the points that belong to this cluster, namely

$$\mathcal{C}_j = \{i \mid \mathbf{x}_i \text{ belongs to cluster } j\}, \quad \forall j : 1 \leq j \leq k$$

Let n_j be the cardinality of $\mathcal{C}_j$, then the center or the mean of each cluster is

$$\mathbf{m}_j = \frac{1}{n_j} \sum_{i \in \mathcal{C}_j} \mathbf{x}_i$$

Note that the condition that all clusters are disjoint and exhaust all the input data points means

$$\begin{aligned} & \mathcal{C}_{j_1} \bigcap \mathcal{C}_{j_2} = \emptyset, \quad \forall j_1, j_2 : 1 \leq j_1 < j_2 \leq k \\ \text{and} \qquad & \mathcal{C}_1 \bigcup \cdots \bigcup \mathcal{C}_k = \{1, \ldots, n\} \end{aligned}$$

Using the notations above, the total sum of the squared distances from each point to its cluster center is

$$\psi = \sum_{j=1}^{k} \sum_{i \in \mathcal{C}_j} \|\mathbf{x}_i - \mathbf{m}_j\|^2 \tag{2.1}$$

where $\|\cdot\|$ denotes a Euclidean norm. This cost function can also be interpreted as the sum of scaled variances in each cluster since $\frac{1}{n_j} \sum_{i \in \mathcal{C}_j} \|\mathbf{x}_i - \mathbf{m}_j\|^2$ is the variance of all points in cluster j. These clusters variances are then scaled by cluster sizes n_j and added together.

Our problem is then to find $\mathbf{m}_1, \ldots, \mathbf{m}_k$, or equivalently $\mathcal{C}_1, \ldots, \mathcal{C}_k$, to minimize ψ in Eq. 2.1. Since moving a point from one cluster to another affects two clusters, namely $\mathcal{C}_{j_1}$ and $\mathcal{C}_{j_2}$, m_{j_1} and m_{j_2} must be recomputed. The inner sum $\sum_{i \in \mathcal{C}_j} \|\mathbf{x}_i - \mathbf{m}_j\|^2$ must also be recomputed for the two clusters. Thus, the definition of the cost in Eq. 2.1 is rather inconvenient for the description and the implementation of algorithms that often move data points around. We instead derive a more convenient cost function as follow.

Let $\bar{\mathbf{x}}$ be the mean of all data points, namely $\bar{\mathbf{x}} = \frac{1}{n} \sum_{i=1}^{n} \mathbf{x}_i$, and let τ be the total sum of squared distance from $\bar{\mathbf{x}}$, i.e.,

$$\begin{aligned} \tau &= \sum_{i=1}^{n} \|\mathbf{x}_i - \bar{\mathbf{x}}\|^2 \\ &= \sum_{j=1}^{k} \sum_{i \in \mathcal{C}_j} \|\mathbf{x}_i - \bar{\mathbf{x}}\|^2 \\ &= \sum_{j=1}^{k} \sum_{i \in \mathcal{C}_j} \|(\mathbf{x}_i - \mathbf{m}_j) + (\mathbf{m}_j - \bar{\mathbf{x}})\|^2 \end{aligned}$$

$$
\begin{aligned}
&= \sum_{j=1}^{k}\sum_{i\in\mathcal{C}_j}((\mathbf{x}_i-\mathbf{m}_j)'(\mathbf{x}_i-\mathbf{m}_j)+2(\mathbf{x}_i-\mathbf{m}_j)'(\mathbf{m}_j-\bar{\mathbf{x}})+(\mathbf{m}_j-\bar{\mathbf{x}})'(\mathbf{m}_j-\bar{\mathbf{x}}))\\
&= \sum_{j=1}^{k}\sum_{i\in\mathcal{C}_j}\|\mathbf{x}_i-\mathbf{m}_j\|^2+2\sum_{j=1}^{k}\sum_{i\in\mathcal{C}_j}(\mathbf{m}_j-\bar{\mathbf{x}})'(\mathbf{x}_i-\mathbf{m}_j)+\sum_{j=1}^{k}\sum_{i\in\mathcal{C}_j}\|\mathbf{m}_j-\bar{\mathbf{x}}\|^2\\
&= \sum_{j=1}^{k}\sum_{i\in\mathcal{C}_j}\|\mathbf{x}_i-\mathbf{m}_j\|^2+2\sum_{j=1}^{k}(\mathbf{m}_j-\bar{\mathbf{x}})'\sum_{i\in\mathcal{C}_j}(\mathbf{x}_i-\mathbf{m}_j)+\sum_{j=1}^{k}n_j\|\mathbf{m}_j-\bar{\mathbf{x}}\|^2\\
&= \psi+\sum_{j=1}^{k}n_j\|\mathbf{m}_j-\bar{\mathbf{x}}\|^2
\end{aligned}
$$

where the last equality follows from the definition of ψ in Eq. 2.1 and the fact that the sum of the difference from the cluster mean to all points in the cluster is zero. The second term in the last equality can be further simplified as follow.

$$
\begin{aligned}
\sum_{j=1}^{k}n_j\|\mathbf{m}_j-\bar{\mathbf{x}}\|^2 &= \sum_{j=1}^{k}n_j\mathbf{m}_j'\mathbf{m}_j-2\sum_{j=1}^{k}n_j\mathbf{m}_j'\bar{\mathbf{x}}+\sum_{j=1}^{k}n_j\bar{\mathbf{x}}'\bar{\mathbf{x}}\\
&= \sum_{j=1}^{k}n_j\mathbf{m}_j'\mathbf{m}_j-2\left(\sum_{j=1}^{k}n_j\mathbf{m}_j\right)'\bar{\mathbf{x}}+n\bar{\mathbf{x}}'\bar{\mathbf{x}}\\
&= \sum_{j=1}^{k}n_j\mathbf{m}_j'\mathbf{m}_j-n\|\bar{\mathbf{x}}\|^2\\
&= \sum_{j=1}^{k}n_j\left(\frac{1}{n_j}\sum_{i\in\mathcal{C}_j}\mathbf{x}_i\right)'\left(\frac{1}{n_j}\sum_{i\in\mathcal{C}_j}\mathbf{x}_i\right)-n\|\bar{\mathbf{x}}\|^2\\
&= \sum_{j=1}^{k}\frac{1}{n_j}\left\|\sum_{i\in\mathcal{C}_j}\mathbf{x}_i\right\|^2-n\|\bar{\mathbf{x}}\|^2
\end{aligned}
$$

Substitute this term back, we get

$$
\begin{aligned}
\tau &= \psi+\sum_{j=1}^{k}\frac{1}{n_j}\left\|\sum_{i\in\mathcal{C}_j}\mathbf{x}_i\right\|^2-n\|\bar{\mathbf{x}}\|^2\\
\psi &= \tau+n\|\bar{\mathbf{x}}\|^2-\sum_{j=1}^{k}\frac{1}{n_j}\left\|\sum_{i\in\mathcal{C}_j}\mathbf{x}_i\right\|^2
\end{aligned}
$$

Since τ and $n\|\bar{\mathbf{x}}\|^2$ are constant, minimizing ψ is equivalent to maximizing ϕ where

$$\phi = \sum_{j=1}^{k} \frac{1}{n_j} \left\| \sum_{i \in \mathcal{C}_j} \mathbf{x}_i \right\|^2 \tag{2.2}$$

This equivalent cost function is convenient since we can maintain the linear sums of all points in each cluster. Moving a point between two clusters requires updating only two linear sums, their norms, and a few elementary operations. The description of the cyclic exchange algorithm will be upon solving the problem of maximizing ϕ as defined in Eq. 2.2.

2.2.2 Existing Works

Lloyd's Algorithm

Lloyd's algorithm was originated in the field of communication. [11] presented a method for finding a least squares quantization in pulse code modulation (a method for representing data as signals in a communication channel). The data of this problem is the probability distribution of the signal, rather than discrete data points. However, the methods are generalized to work with data points in multidimensional space.

[11] notes that any local minima of quantization problem must satisfy two conditions:

- A representative value must be the center of mass of the probability distribution in the region it represents, i.e. the representative values are the conditional expected values in the respective regions.
- A region boundary must be the mid-point of two adjacent representative values.

These two conditions can be stated, respectively, in data clustering context as:

- Each cluster center must be the center of mass of the data in that cluster.
- Each data point must belong to the cluster with the nearest center.

Based on these two conditions, Lloyd's algorithm starts with a random initial cluster centers. Then, it assigns each data point to the nearest cluster and recomputes the center of mass of each cluster. These two steps are repeated until no data point can be moved. Hence, the two conditions above are satisfied, and we reach a local minimum.

However, these two conditions are only neccessary but not sufficient for the global minimum. The local minimum that Lloyd's algorithm obtains depend much on the initial solutions [12].

Variants of Lloyd's Algorithm

[10] proposed a deterministic method for running multiple (nk) iterations of Lloyd's algorithm to obtain the claimed global optimum. The method incrementally finds optimal solutions for $1, \ldots, k$ clusters. It relies on the assumption that an optimal solution for k clustering can be obtained via local search from an optimal solution for $k-1$ clustering with an additional center point to be determined. Although [10] claims that this method is "experimentally optimal", it does not give a rigorous proof of its underlying assumption.

[8] and [10] use k-dimensional tree to speed up each iteration of Lloyd's algorithm. K-dimensional tree is a generalized version of one-dimensional binary search tree. It expedites searching over k-dimensional space and can help Lloyd's algorithm find the closest center to a given point in time $O(\log n)$, as opposed to $O(n)$ by a linear search.

[9] combines Lloyd's algorithm with a local search algorithm, resulting in better solutions. The local search algorithm is based on swapping current cluster centers with candidate centers, which are in fact data points. The algorithm is theoretically proven to give a solution within about nine times worse than the optimal solution, and empirical studies show that the algorithm performs competitively in practice.

2.3 Neighborhood Search Techniques

Many combinatorial optimization problems are computationally hard. Exact algorithms that deterministically find optimal solutions usually take too much time to be practical. Therefore, these problems are mostly approached by heuristics. Neighborhood search is a class of heuristics that seek to incrementally improve a feasible solution. It is applicable to many optimization problem. [1] gives an overview of the neighborhood search.

Neighborhood search begins with an initial feasible solution, which may be obtained by randomizing or by another fast heuristic. Then, the algorithm searches the *neighborhood* — a set of feasible solutions obtained by deviating the initial solution. The solutions in neighborhood include both better and worse solutions. As a rule of thumb, the larger the neighborhood, the more likely it contains better solutions, but at the expense of a longer search time. The algorithm then searches the neighborhood for a better solution, uses it as a current solution for generating neighborhood. When there is no better solutions in the neighborhood, the current solution is callled locally optimal, and the algorithm terminates.

Chapter 3

Cyclic Exchange Algorithm

3.1 Motivation

The popular Lloyd's algorithm converges quickly to local optima. At each step, Lloyd's algorithm moves only one data point to a different cluster and thus limits the search space. Sometimes the local optimum can be improved by moving multiple points at the same time. Consider data in Figure 3-1. Lloyd's algorithm converges to the solution portrayed in the figure. There is no way to move a single point and further improve this solution. Then, consider a moving technique depicted in Figure 3-2. In this case, the solution is of better quality than the previous Lloyd's local optimum. This method moves multiple points *at once* (the intermediate solutions are of worse quality than the original one). As will be seen, we model this movement as a cycle in an improvement graph, hence the name Cyclic Exchange algorithm. The use of cyclic transfers in combinatorial optimization problem was pioneered by [13].

One technique to improve the solution quality of Lloyd's algorithm is to run the

Figure 3-1: Undesirable Local Optimum from Lloyd's Algorithm

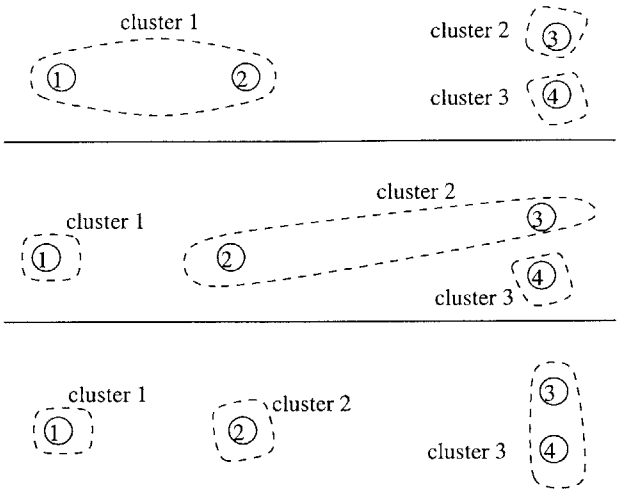


Figure 3-2: Improvement Upon Local Optimum from Lloyd's Algorithm by Cyclic Exchange: Node 2 is moved to cluster 2. Node 3 is moved to cluster 3.

algorithm multiple times using different initial solutions and choose the best local optimum solution as the final answer. This stil leaves room for improvement in many cases. Potentially better solutions may be found in the set of local optimums from Cyclic Exchange.

3.2 Descriptions of the Algorithm

In neighborhood search, we must first define the neighborhood of any feasible solution. Then, we have to find an efficient way to search the neighborhood for a better solution. In the case of Cyclic Exchange, we define a neighborhood of a solution as a set of solutions that can be obtained by moving multiple data points in a cycle. This movement is described by a cycle in an improvement graph, which we now define.

Let the current feasible solution be $(\mathcal{C}_1, \ldots, \mathcal{C}_k)$. We define a weighted directed graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$. $\mathcal{V} = \{1, \ldots, n\} \bigcup \{d_1, \ldots, d_k\}$, where the set $\{1, \ldots, n\}$ are nodes for each data point, and the second set are dummy nodes for each cluster. For each pair of non-dummy nodes i_1 and i_2, there is an edge (i_1, i_2) if and only if the corresponding data points are in different clusters. In other words, $(i_1, i_2) \in \mathcal{E} \iff i_1 \in \mathcal{C}_{j_1}, i_2 \in \mathcal{C}_{j_2}$, and $j_1 \neq j_2$.

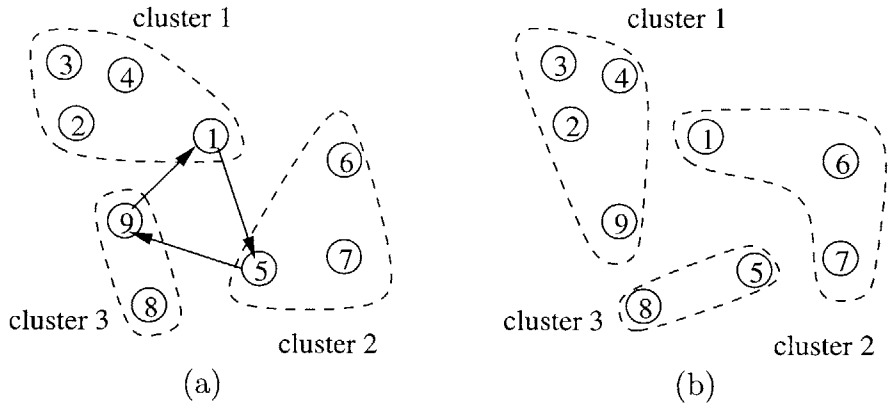


Figure 3-3: Definition of Edges in the Improvement Graph: (a) Current solution and a cycle $(1, 5, 9)$. (b) Resulting solution after applying the cycle.

An edge (i_1, i_2) represents moving x_{i_1} from cluster j_1 to cluster j_2 and moving x_{i_2} out of cluster j_2. Figure 3-3 shows the effect of cycle $(1, 5, 9, 1)$ if it is applied. The edge $(1, 5)$ substitutes node 1 for node 5 in cluster 2. Node 5 is then left afloat (not in any particular cluster). The edge $(5, 9)$ then puts node 5 in cluster 3, again leaving node 9 afloat. The edge $(9, 1)$ puts node 9 in cluster 1, and hence closes the cycle.

The weight of the cycle must relate the currect cost of the solution to the new cost. We let the change in cost (in term of ϕ in Eq. 2.2) be the negative of the weight of the cycle. Let the weight of (i_1, i_2) be the cost of substituting node i_1 for node i_2 in the cluster where i_2 is currently in, i.e., $\mathcal{C}_{j_2}$, and leave i_2 afloat. Note that this edge affects only the cluster $\mathcal{C}_{j_2}$, and does not affect $\mathcal{C}_{j_1}$. We find the weight $w(i_1, i_2)$ as follow.

$$\begin{aligned} w(i_1, i_2) &= \phi_{\text{current}} - \phi_{\text{new}} \\ &\quad \text{(only the term for cluster } \mathcal{C}_{j_2} \text{ is affected)} \\ &= \frac{1}{n_j} \left\| \sum_{i \in \mathcal{C}_{j_2}} \mathbf{x}_i \right\|^2 - \frac{1}{n_j} \left\| \left(\sum_{i \in \mathcal{C}_{j_2}} \mathbf{x}_i \right) - \mathbf{x}_{i_2} + \mathbf{x}_{i_1} \right\|^2 \end{aligned}$$

With this definition of the improvement graph, however, we are constrained to move exactly k nodes in a cycle. This restriction is undesirable as any neighbor has cluster of the same size. Also, we want Cyclic Exchange to be more general than

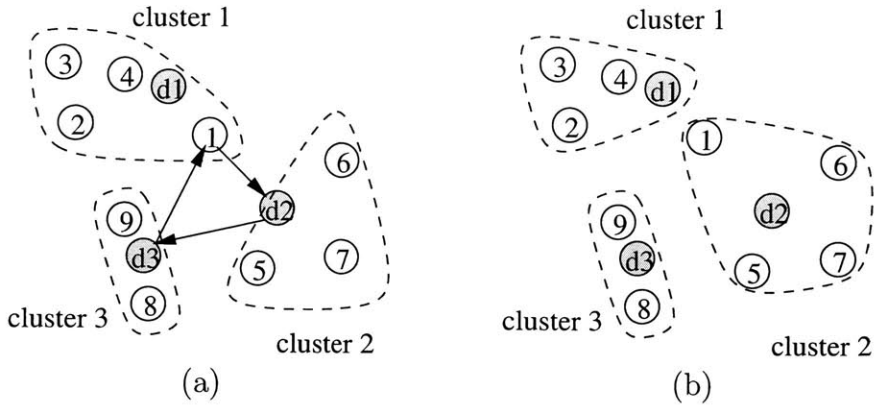


Figure 3-4: Definition of Dummy Nodes in the Improvement Graph: (a) Current solution and a cycle $(1, d_2, d_3, 1)$. (b) Resulting solution after applying the cycle.

Lloyd's algorithm. Moving only one data point at at time should also be permitted. Therefore, we introduce dummy nodes $d_1, \ldots, d_k$, which do not represent data points but represent clusters. More precisely, an edge (i, d_j) represents moving of node i to cluster j and ejecting no points from cluster j. Similarly, an edge (d_j, i) represents moving no points to the cluster where i is currently in, but instead ejecting i from that cluster. An edge (d_{j_1}, d_{j_2}) represents no movement but is present to allow no movement in some clusters. Figure 3-4 shows the effect of a cycle $(1, d_2, d_3, 1)$, which simply moves node 1 to cluster 2.

The costs of edges to and from a dummy node are:

$$
\begin{aligned}
w(i, d_j) &= \frac{1}{n_j} \left\| \sum_{i \in \mathcal{C}_j} \mathbf{x}_i \right\|^2 - \frac{1}{n_j + 1} \left\| \left(\sum_{i \in \mathcal{C}_j} \mathbf{x}_i \right) + \mathbf{x}_i \right\|^2 \\
w(d_j, i) &= \frac{1}{n_j} \left\| \sum_{i \in \mathcal{C}_{j_2}} \mathbf{x}_i \right\|^2 - \frac{1}{n_j - 1} \left\| \left(\sum_{i \in \mathcal{C}_{j_2}} \mathbf{x}_i \right) - \mathbf{x}_i \right\|^2 \\
&\quad \text{(let } i \text{ be currently in cluster } j_2\text{)} \\
w(d_{j_1}, d_{j_2}) &= 0 \quad \text{(no movement)}
\end{aligned}
$$

With this definition of nodes and edges, the number of nodes in the improvement graph is $|\mathcal{V}| = n + k$, and the number of edges is $|\mathcal{E}| = O((n + k)^2)$. Now, we have to find a way to search the neighborhood for a better solution. Since the cost of the

procedure Negative-cost Cycle Detection:
input: improvement graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, number of cluster k
output: a negative-cost cycle W^*
begin
  $\mathcal{P}_1 \leftarrow \{(i,j) \in \mathcal{E} \mid w(i,j) < 0\}$
  $l \leftarrow 1$
  $W^* \leftarrow \emptyset$
  while $l < k$ and $w(W^*) \geq 0$ **do**
  begin
    while $\mathcal{P}_k \neq \emptyset$ **do**
    begin
      remove a path P from $\mathcal{P}_l$
      let $i \leftarrow$ head$[P]$ and $h \leftarrow$ tail$[P]$
      if $(i,h) \in \mathcal{E}$ and $w(P) + w(i,h) < w(W^*)$ **then** $W^* \leftarrow P \bigcup \{(i,h)\}$
      for each $(i,j) \in \mathcal{E}(i)$ **do**
      begin
        if label$(j) \notin$ LABEL(P) and $w(P) + w(i,j) < 0$ **then**
        begin
          add the path $P \bigcup \{(i,j)\}$ to $\mathcal{P}_{k+1}$
          if $\mathcal{P}_{k+1}$ contains another path with the same key as $P \bigcup \{(i,h)\}$
          then remove the dominated path
        end
      end
    end
    $k \leftarrow k + 1$
  end
end

Figure 3-5: Subset-disjoint Negative-cost Cycle Detection Algorithm (reproduction of Fig.2 in [2])

new solution is equal to the current cost minus the cost of the improvement cycle, we search for a cycle in the improvement graph with a negative cost.

Instead of enumerating the improvement graph explicitly and searching for a negative-cost cycle, we take an advantage of a special structure of the improvement graph and generate only portions of the improvement graph in which we are actively searching. There are no edges between nodes within the same clusters, and all clusters are disjoint. Therefore, we are looking for a negative-cost cycle with all nodes in different clusters, in a graph with $n+k$ nodes. We employ a subset-disjoint negative-cost cycle detection algorithm from [2] and is shown in Figure 3-5. The subsets in the algorithm is the current clusters $\mathcal{C}_1, \ldots, \mathcal{C}_k$.

This algorithm carefully enumerates all, but not redundant, subset-disjoint negative-cost paths in the graph. Using the fact that every negative-cost cycle must contain a negative-cost path, the algorithm enumerate all negative-cost paths incrementally from length one (negative-cost edges). For all negative-cost paths found, the algorithm connects the start and end nodes of the paths and check if the resulting cycles have negative costs.

Although this algorithm is essentially an enumeration, [2] speeds up the process by noting that if two paths contains nodes from the same subsets and the start and end nodes of the two paths are also from the same subsets, the path with higher cost can be eliminated. The remaining path will certainly give a lower cost, provided that it eventually makes a negative-cost cycle. A "LABEL", as called in Figure 3-5, is a bit vector representing the subsets to which all nodes in the path belong. By hashing the LABELs, start node, and end node, we can check if we should eliminate a path in $O(k/C)$ on average, where C is the word size of the computer.

The step of finding a negative-cost cycle and applying it to the improvement graph is then repeated until there is no negative-cost cycle in the graph. We call the process of obtaining an initial solution and improving until a local optimum is reached, an *iteration* of Cyclic Exchange. Figure 3-6 lists the pseudo-code for an iteration of Cyclic Exchange.

procedure Cyclic Exchange:
input: data points $\mathcal{X} = \{\mathbf{x}_1, \ldots, \mathbf{x}_n\}$, number of clusters k
output: cluster description $(\mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_k)$
begin
  obtain an initial solution $(\mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_k)$
  construct the improvement graph $\mathcal{G}$ (implicitly)
  while $\mathcal{G}$ contain a negative-cost cycle **do**
  begin
    improve the solution according to such negative-cost cycle
    update improvement graph $\mathcal{G}$ (implicitly)
  end
end

Figure 3-6: An Iteration of Cyclic Exchange Algorithm

3.3 Test Data

The Cyclic Exchange algorithm is implemented and evaluated against Lloyd's algorithm. We use both synthetic data and data from applications of k-means clustering. This section describes characteristics of the data that we use.

3.3.1 Synthetic Data

Synthetic data are random data generated according to some models. These models are designed to immitate situations that can happen in real applications or are designed to discourage naive algorithms. We use two models for synthetic data that [9] uses. The codes for generating these data are given in Appendix A. The characteristics of these two types of data are as follow.

ClusGauss

This type of synthetic data are generated by choosing k random centers uniformly over a hypercube $[-1, 1]^d$. Then $\frac{n}{k}$ data points are randomly placed around each of these k centers using symetric multivariated Gaussian distribution.

Data set name: c-n-d-k-std

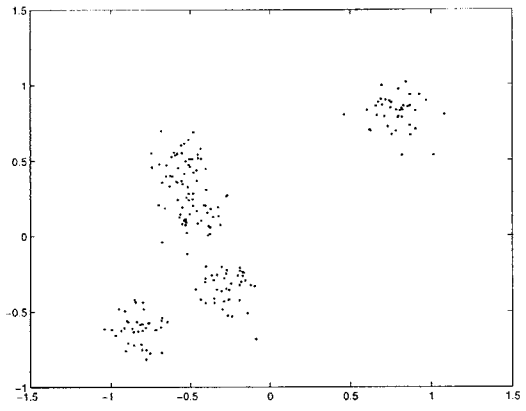


Figure 3-7: ClusGauss Synthetic Data: $n = 200$, $d = 2$, $k = 5$, $std = 0.1$.

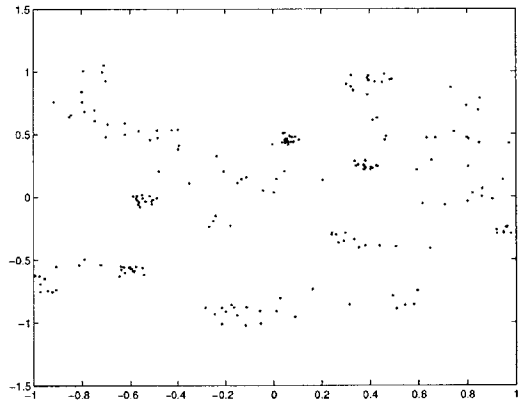


Figure 3-8: MultiClus Synthetic Data: $n = 200$, $d = 2$, $std = 0.1$.

where n is the number of data, d is the dimensionality of data, k is the number of random centers, and std is the standard deviation in each dimension of the Gaussian distribution. An example of ClusGauss data in two dimensions is shown in Figure 3-7.

MultiClus

MultiClus data consists of a number of multivariate Gaussian clusters of different sizes. Each cluster center is again uniformly random over a hypercube $[-1, 1]^d$. The size of the cluster is a power of 2, and the probability that we generate a cluster with 2^t points is $\frac{1}{2^t}$, for $t = 0, 1, \ldots$.

Data set name: `m-`n`-`d`-`std

where n is the number of data, d is the dimensionality of data, and std is the standard

Table 3.1: List of Applications Data

Data set	Description	Application	n	d
lena32-22	32 × 32 pixels, 2 × 2 block	Vector quantization	256	4
lena32-44	32 × 32 pixels, 4 × 4 block	Vector quantization	64	16
covtype-100	Forest cover type	Knowledge discovery	100	54
covtype-200	Forest cover type	Knowledge discovery	200	54
covtype-300	Forest cover type	Knowledge discovery	300	54
covtype-400	Forest cover type	Knowledge discovery	400	54
covtype-500	Forest cover type	Knowledge discovery	500	54
ionosphere	Radar reading	Classification	351	34
zoo	Artributes of animals	Classification	101	17
machine	Computer performance	Classification	209	8
glass	Glass oxide content	Classification	214	9
soybean	Soybean disease	Classification	47	35
lenses	Hard/soft contact lenses	Classification	24	4
diabetes	Test results for diabetes	Classification	768	8
wine	Chemical analysis	Classification	178	13
housing	Housing in Boston	Classification	506	14

deviation in each dimension of the Gaussian distribution. An example of MultiClus data in two dimensions is shown in Figure 3-8

3.3.2 Applications Data

Applications Data are drawn from various applications of k-means clustering, including vector quantization, knowledge discovery, and various classification problems. Table 3.1 shows a list of these data and brief descriptions. `lena` is extracted from the well-known Lena image, using 2 × 2 and 4 × 4 sample blocks, similar to [9]. `covtype` data is from [6]. All others are from [3].

Chapter 4

Software Development

A preliminary version of Cyclic Exchange is implemented for initial testing. The program is written in C++ and compiled by g++ compiler version 3.2.2 on Linux operating system. All the results are obtained by running the programs on IBM IntelliStation EPro machine with 2 GHz Pentium IV processor and 512 MB of RAM.

The software is structured in three layers as depicted in Figure 4-1. The bottom layer — elementary layer — contains basic elements such as vectors, graph nodes, paths, and clusters. The middle layer — iteration layer — contains implementation of an iteration of Cyclic Exchange, an iteration of Lloyd's algorithm, and generation of random initial solution. Finally, the top layer — experiment layer — uses the modules in the middle layer in a series of experiments: standalone Cyclic Exchange, Cyclic Exchange with preclustering, and two-stage algorithm. Listings of source codes are given in Appendix B.

The elementary layer consists of the following classes, which describe basic elements in the implementation of clustering algorithms.

- `Vector`

  A `Vector` represents a data point in d-dimensional space. Usual vector operations, such as addition, scaling, and dot product, are allowed.

- `VectorSet`

  A `VectorSet` is a collection of `Vectors`.

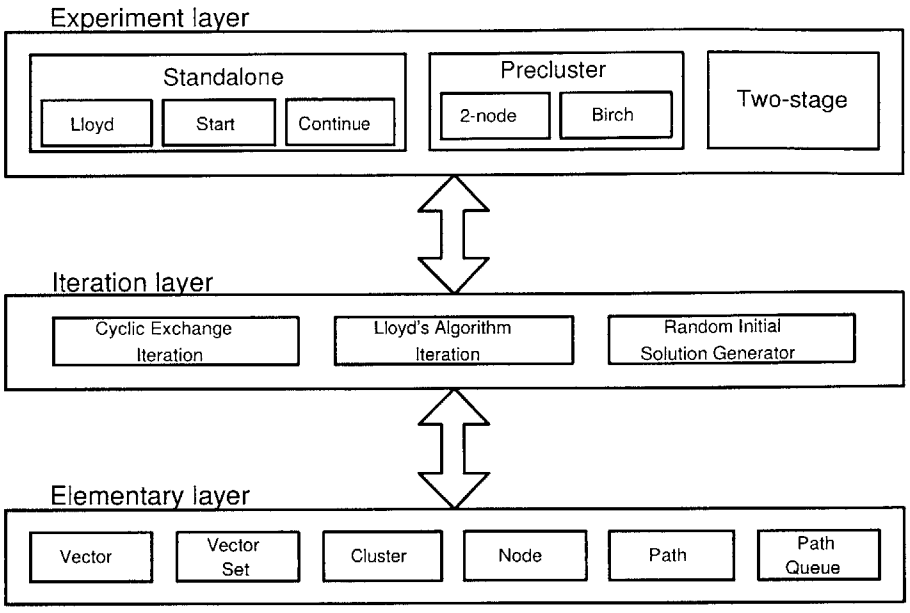


Figure 4-1: Layered Structure of the Software

- `Node`

  A `Node` is a wrapper class for `Vector`. It provides an interface with other graph-related classes.

- `Cluster`

  A `Cluster` is a collection of `Nodes` in a cluster. Its methods provide moving nodes between clusters and calculating the centroid of the cluster. This class is used for implementations of both Cyclic Exchange and Lloyd's algorithm, in the iteration layer.

- `Path`

  A `Path` represents a path in the improvement graph. This class is used exclusively for implementation of negative-cost cycle detection in Cyclic Exchange. A method for checking if two paths have the same key is provided.

- `PathQueue`

  A `PathQueue` is a data structure for storing negative-cost paths during the execution of negative-cost cycle detection. It provides methods for hashing

path keys and eliminating the dominated path.

The iteration layer has a single class called `Cyclic`. `Cyclic` holds the data points and has methods for performing an iteration of Cyclic Exchange, performing an iteration of Lloyd's algorithm, and generating a random solution. In addition, preclustering methods are added later for the preclustering experiment. The details of each methods are as followed.

- `Cyclic::cyclicExchange()`

  `cyclicExchange()` performs an iteration of Cyclic Exchange. An iteration starts with the current solution stored within the class. Then, it enters a loop for finding a negative-cost cycle and applying such cycle. The loop terminates when there is no negative-cost cycle or the accumulated improvement during the last three steps are less than 0.1%.

- `Cyclic::lloyd()`

  `lloyd()` performs an iterations of Lloyd's algorithm with a similar termination criteria as `cyclicExchange()`.

- `Cyclic::randomInit()`

  This method generates a random initial solution by choosing k centers uniformly from n data points.

- `Cyclic::preCluster()`

  In preclustering experiment, this methods is used to perform two-point preclustering. Any two points that are closer than a given threshold are grouped into miniclusters.

- `Cyclic::cfPreCluster()`

  In preclustering experiment, this methods is used to perform Birch preclustering (using CF Tree).

- `Cyclic::initByCenters(VectorSet centers)`

  For two-stage algorithm, this method initializes the current solution to the

solution described by the argument `centers`. These solutions are stored during the first stage.

Chapter 5

Experiments, Results, and Discussions

5.1 Standalone Cyclic Exchange

5.1.1 Methods

The first version of our program runs several iterations of Cyclic Exchange procedure in Figure 3-6. The Lloyd's algorithm is also run on the same datasets as a reference. As the first assessment of the Cyclic Exchange algorithm, we seek to answer the following questions.

- How are the initial solutions chosen?
- What is the quality of solutions obtained by Cyclic Exchange?
- How long does the Cyclic Exchange take?
- How does Cyclic Exchange reach its local optimum?

We try three versions of Cyclic Exchanges:

Start: Initial solutions are constructed from randomly choosing k data points as clusters centers and assign each data point to the nearest center.

Continue: Initial solutions are local optima from Lloyd's algorithm.

Hybrid: Initial solutions are random. A step of Lloyd's algorithm and a step of Cyclic Exchange are applied alternatively.

We run Lloyd's algorithm and three versions of Cyclic Exchanges for the same number of iterations, where an iteration start with a random initial solution and end when the algorithm does not produce more than 0.1% cost improvement in three loops. This terminating condition is chosen in order to prevent the program for running too long but does not get significant cost improvement. The cost of the best local optimum and the amount of time used are recorded. For each dataset, all methods use the same random seed for the random number generator and thus use the same sequence of random numbers. Since the only time that random numbers are used is when a random solution is constructed, random initial solutions for all four methods are the same in every iteration.

In addition to the final cost, the cost after each step that Cyclic Exchange applies negative-cost cycle to the solution and after each step that Lloyd's algorithm moves a single point are also recorded. These trails of costs will reveal the dynamics within an iteration of Cyclic Exchange and Lloyd's algorithm.

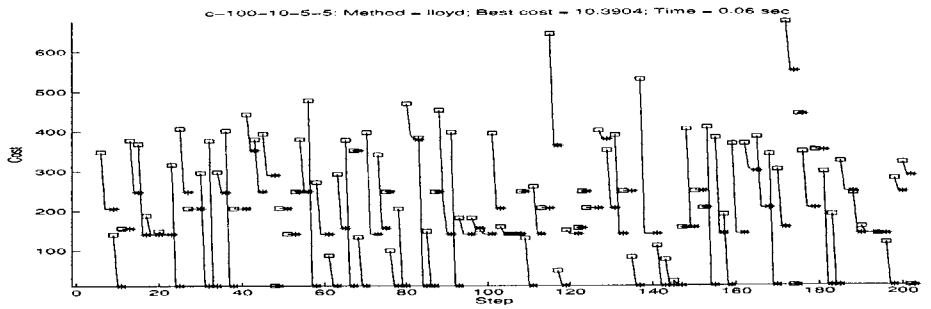
5.1.2 Results and Discussion

The final cost and time for Lloyd's algorithm and three versions of Cyclic Exchange are shown in Table 5.1. As a preliminary assessment, the program is run only on small-size datasets. In general, the time used by all versions of Cyclic Exchange is significantly more than the time used by Lloyd's algorithm, and the Cyclic Exchange get better costs in a few datasets.

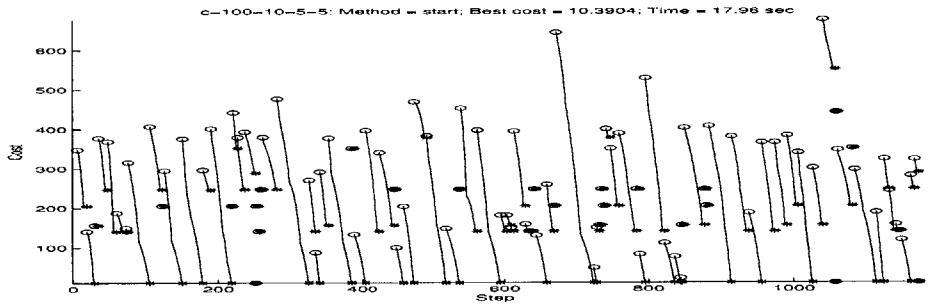
Figure 5-1 shows the costs over 100 iterations of Lloyd's algorthm and three versions of Cyclic Exchange for dataset `c-100-10-5-5`, for which all versions of Cyclic Exchange do not get better results. Figure 5-2 shows similar graphs for dataset `m-100-10-5`, for which all versions of Cyclic Exchange get better results. The graphs for other datasets are given in Appendix C. In the graphs, squares denote starting

Table 5.1: Comparison between Lloyd's algorihtm, Start Cyclic Exchange, Continue Cyclic Exchange, and Hybrid Cyclic Exchange. All methods run for 100 iterations.

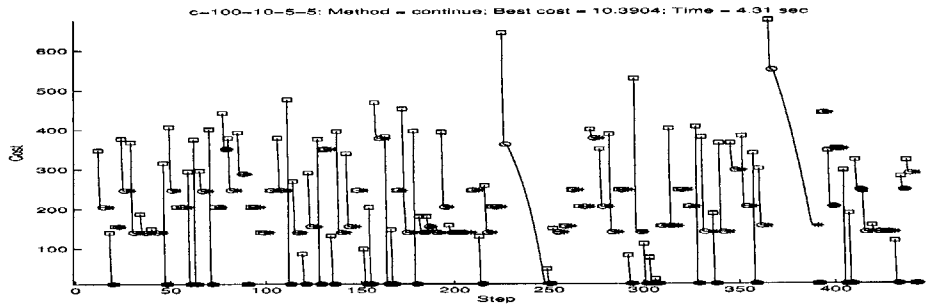
Dataset	k	Final Cost				Time (seconds)			
		Lloyd	Start	Continue	Hybrid	Lloyd	Start	Continue	Hybrid
c-100-3-5-5	53	$3.5151e+00$	$3.5151e+00$	$3.5151e+00$	$3.5151e+00$	0.05	28.37	5.63	5.83
c-100-5-5-5	53	$3.8248e+00$	$3.8248e+00$	$3.8248e+00$	$3.8248e+00$	0.05	24.68	4.94	5.95
c-100-10-5-5	53	$1.0390e+01$	$1.0390e+01$	$1.0390e+01$	$1.0390e+01$	0.06	17.96	4.31	3.94
c-100-20-5-5	53	$1.9527e+01$	$1.9527e+01$	$1.9527e+01$	$1.9527e+01$	0.09	20.43	4.28	5.11
c-200-3-5-5	53	$7.0512e+00$	$7.0512e+00$	$7.0512e+00$	$7.0512e+00$	0.09	331.49	94.27	40.37
c-200-3-10-5	53	$1.0279e+02$	$1.0279e+02$	$1.0279e+02$	$1.0279e+02$	0.09	370.01	37.98	35.94
c-200-5-5-5	53	$1.9782e+01$	$1.9782e+01$	$1.9782e+01$	$1.9782e+01$	0.10	164.68	18.24	24.28
c-200-5-10-5	53	$1.7848e+02$	$1.7848e+02$	$1.7848e+02$	$1.7848e+02$	0.09	179.20	14.06	12.50
c-200-10-5-5	53	$1.8564e+01$	$1.8564e+01$	$1.8564e+01$	$1.8564e+01$	0.12	218.18	16.98	20.70
c-200-10-10-5	53	$6.3048e+02$	$6.3048e+02$	$6.3048e+02$	$6.3048e+02$	0.10	130.80	16.31	13.18
c-200-20-5-5	53	$6.1182e+01$	$6.1182e+01$	$6.1182e+01$	$6.1182e+01$	0.17	148.14	30.63	38.02
c-200-20-10-5	53	$1.6607e+03$	$1.6607e+03$	$1.6607e+03$	$1.6607e+03$	0.15	123.89	8.53	11.76
m-100-3-5	53	$3.0179e+01$	$3.0179e+01$	$3.0179e+01$	$3.0179e+01$	0.05	56.05	7.58	4.45
m-100-5-5	53	$9.6927e+01$	$9.6927e+01$	$9.6927e+01$	$9.6927e+01$	0.05	37.54	8.32	5.72
m-100-10-5	53	$5.2319e+02$	$5.1164e+02$	$5.1200e+02$	$5.1200e+02$	0.10	26.64	8.51	6.00
m-100-20-5	53	$3.3160e+02$	$3.3160e+02$	$3.3160e+02$	$3.3160e+02$	0.10	200.83	6.24	5.20
m-200-3-5	53	$3.3730e-01$	$3.3730e-01$	$3.3730e-01$	$3.3730e-01$	0.11	290.85	29.12	36.23
m-200-5-5	53	$3.3783e+02$	$3.3783e+02$	$3.3783e+02$	$3.3783e+02$	0.11	324.85	33.08	24.18
m-200-10-5	53	$1.5087e+03$	$1.5082e+03$	$1.5076e+03$	$1.5084e+03$	0.31	774.08	81.74	120.61
m-200-20-5	53	$1.7223e+03$	$1.6912e+03$	$1.6912e+03$	$1.6912e+03$	0.19	372.21	47.36	25.78
covtype-100	53	$2.9006e+03$	$2.8365e+03$	$2.8366e+03$	$2.8366e+03$	0.29	121.28	49.15	39.17
glass	51	$1.2391e+03$	$1.2391e+03$	$1.2391e+03$	$1.2391e+03$	0.16	238.14	15.65	20.13
lenses	51	$6.0000e+01$	$6.0000e+01$	$6.0000e+01$	$6.0000e+01$	0.01	0.18	0.12	0.08
soybean	52	$3.6714e+02$	$3.6714e+02$	$3.6714e+02$	$3.6714e+02$	0.07	5.97	1.10	1.27
wine	51	$1.2779e+03$	$1.2779e+03$	$1.2779e+03$	$1.2779e+03$	0.12	833.00	17.16	19.14
zoo	51	$9.6725e+02$	$9.6725e+02$	$9.6725e+02$	$9.6725e+02$	0.06	21.48	3.45	2.61



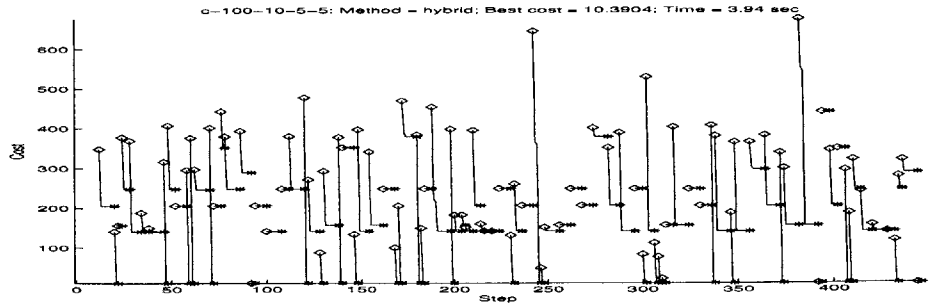
(a)



(b)

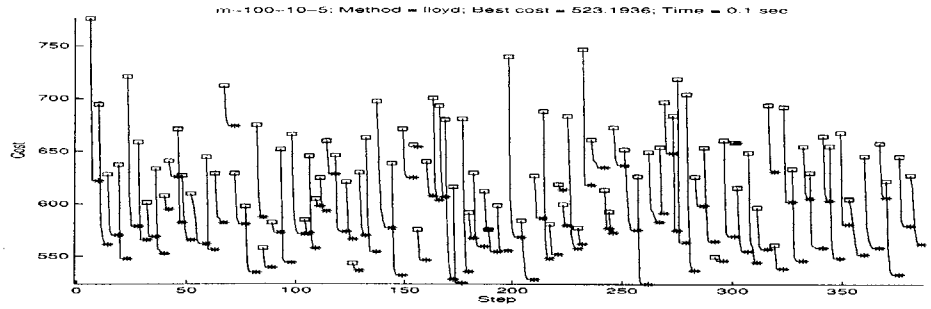


(c)

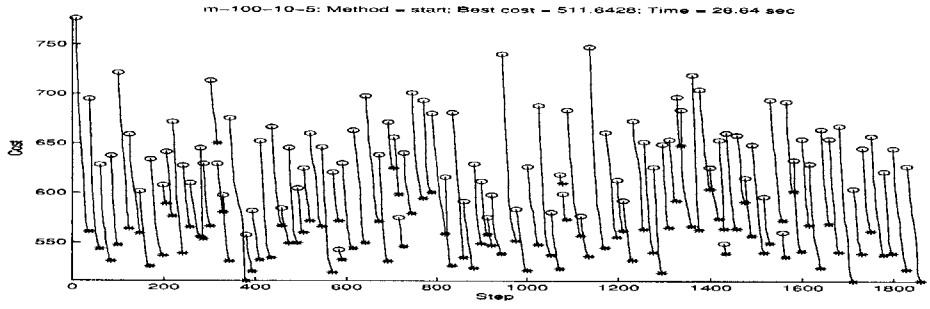


(d)

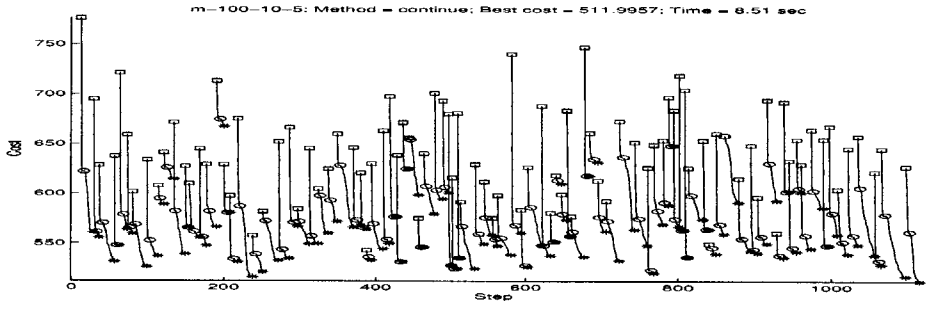
Figure 5-1: Dynamics During Iterations of Lloyd's Algorithm and Cyclic Exchange for Dataset `c-100-10-5-5`: (a) Lloyd's algorithm, (b) Start Cyclic Exchange, (c) Continue Cyclic Exchange, (d) Hybrid Cyclic Exchange.



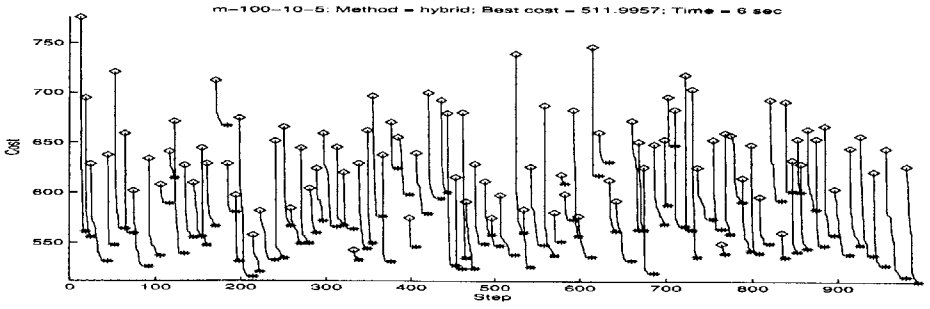
(a)



(b)



(c)



(d)

Figure 5-2: Dynamics During Iterations of Lloyd's Algorithm and Cyclic Exchange for Dataset `m-100-10-5`: (a) Lloyd's algorithm, (b) Start Cyclic Exchange, (c) Continue Cyclic Exchange, (d) Hybrid Cyclic Exchange.

points for Lloyd's algorithm, circles denote starting points for Cyclic Exchange, and asterisks denote the final costs for each iteration. For Hybrid Cyclic Exchange, diamonds denote the starting point where Lloyd's algorithm and Cyclic Exchange are applied alternatively.

For both datasets, Start Cyclic Exchange gradually decrease the cost over many steps in each iterations. Since each step of Cyclic Exchange takes considerable amount of time, Start Cyclic Exchange takes much more time than Continue Cyclic Exchange, as shown in Table 5.1. Hybrid Cyclic Exchange gets the same costs for almost datasets as Continue Cyclic Exchange, and it also takes comparable time to Continue Cyclic Exchange.

For the datasets that all versions of Cyclic Exchange cannot obtain better solutions (e.g. `c-100-10-5-5`, as shown in Figure 5-1), each iteration of Continue Cyclic Exchange consists of only one step — to verify that there is no negative-cost cycle in the improvement graph. On the other hand, for some datasets (e.g. `m-100-10-5`, as shown in Figure 5-2), Cyclic Exchange is capable of finding more improvement upon Lloyd's algorithm. In this case, Continue Cyclic Exchange utilizes fast Lloyd's algorithm to bring down the cost significantly and uses slower Cyclic Exchange steps to decrease the cost further.

In addition to the longer running time, the final costs for Start Cyclic Exchange are no smaller than those for Continue Cyclic Exchange with an exception of dataset `m-100-10-5` where the different is almost insignificant. This observation implies that using random starts for Cyclic Exchange is not preferable. Lloyd's algorithm can achieve the same amount of cost decreases in much less time. Applying Cyclic Exchange only after Lloyd's algorithm has reached a local optimum seems more reasonable.

An apparent problem seen from the results above is the running time of all versions of Cyclic Exchange. Cyclic Exchange uses 3 to 4 order of magnitudes more time than Lloyd's algorithm and yet obtains similar results. Although [2] sucessfully uses the subset-disjoing negative-cost cycle detection algorithm to perform neighborhood search on the capacitated minimum spaning tree problem (CMST), the same algo-

rithm seems inviable here. For CMST, [2] runs the program for 3600 seconds on the graph with 200 nodes. This degree of complexity is plausible for CMST, but it is not comparable to Lloyd's algorithm, which is very fast. Unless Cyclic Exchange can get a much lower cost than Lloyd's algorithm, it will not be a substitute for Lloyd's algorihtm. This problem leads us to try to increase the scalability of Cyclic Exchange via preclustering.

5.2 Cyclic Exchange with Preclustering

5.2.1 Methods

The results from the previous section suggest that Cyclic Exchange takes too long to be practical, no matter what type of initial solutions are used. In order to improve the scalability of cyclic exchange, we try to decrease the effective data size. [14] suggests that not all data are equally important. For example, data points that are very close together may be assumed to be in the same cluster. These data points can be preclustered and treated effectively as a single data point, which needs to be scaled appropriately when the cost is calculated.

In this section, we present our experiments on two kinds of preclustering: Two-point preclustering and Birch preclustering. For both experiments, we record final cost, time, and effective data size after preclustering.

Two-point Preclustering

The first try on decreasing the effective data size is to group any two data points that are very close into a minicluster. The threshold for how close the two points in the minicluster should be is to be determined. The preclustered points will never be taken apart and will essentially be in the same cluster in every solution. A minicluster can be sufficiently described by the number of points in the minicluster and the vector sum of all points in the minicluster since the cost function in Eq. 2.2 involves only these two quantities. This description of a minicluster is similar to the clustering feature

in [14], except that the total squared magnitude is not necessary in our context.

In this scheme of preclustering, the data are preprocessed by first calculating the distances between all pairs of points and then grouping any two points that are closer than the threshold into miniclusters. Once a point is preclustered with another point, it will not be precluster with other points. Hence, the minicluster is of size two. The resulting miniclusters and unpaired points are then used as data for Cyclic Exchange. Since miniclusters are limited to two points, the effective data size will be no less than half of the original data size.

Birch Preclustering

Two-point preclustering, although simple, is not flexible. It does not precluster more than two points that are very close together, and the threshold as to how close points in the minicluster should be is not obvious. We employ a highly scalable heirachical clustering algorithm called Birch [14]. Birch clustering algorithm is divided into several stages. We employ the first stage where Birch reads all the data and construct a clustering feature (CF) tree. The CF tree groups data points that are close together in subtrees. We extract miniclusters from the leaves of the CF tree. By adjusting the parameters in building the CF tree, we get different trees of different sizes and thus different degree of preclustering. If the CF tree is small, we have a small number of miniclusters, and Cyclic Exchange will run faster. However, the quality of solution may be deteriorated since some points that should belong to different clusters may fall into the same minicluster. In constrast, if the CF tree is large, the effective data size does not decrease much, but the quality of solution is less likely deteriorated.

5.2.2 Results and Discussion

Two-point Preclustering

The effective data size, final cost, and time for standalone Continue Cyclic Exchange and Continue Cyclic Exchange with two-point preclustering are shown in Table 5.2. These results include larger datasets that are not tested for the standalone method.

Table 5.2: Results for Continue Cyclic Exchange with Two-point Preclustering

Dataset	Data Size		Final Cost		Time (seconds)	
	Original	Effective	Continue	Two-point	Continue	Two-point
c-100-3-5-5	100	91	$3.5151e+00$	$3.5151e+00$	5.63	4.45
c-100-5-5-5	100	95	$3.8248e+00$	$3.8248e+00$	4.94	4.26
c-100-10-5-5	100	100	$1.0390e+01$	$1.0390e+01$	4.31	4.19
c-100-20-5-5	100	100	$1.9527e+01$	$1.9527e+01$	4.28	4.20
c-200-3-5-5	200	168	$7.0512e+00$	$7.0512e+00$	94.27	54.99
c-200-3-10-5	200	174	$1.0279e+02$	$1.0279e+02$	37.98	29.00
c-200-5-5-5	200	191	$1.9782e+01$	$1.9782e+01$	18.24	16.15
c-200-5-10-5	200	195	$1.7848e+02$	$1.7848e+02$	14.06	14.05
c-200-10-5-5	200	200	$1.8564e+01$	$1.8564e+01$	16.98	16.59
c-200-10-10-5	200	200	$6.3048e+02$	$6.3048e+02$	16.31	15.88
c-200-20-5-5	200	200	$6.1182e+01$	$6.1182e+01$	30.63	29.20
c-200-20-10-5	200	200	$1.6607e+03$	$1.6607e+03$	8.53	8.16
c-500-3-5-5	500	384	N/A	$1.9828e+01$	N/A	98.99
c-500-3-10-5	500	380	N/A	$2.3745e+02$	N/A	126.54
c-500-5-5-5	500	468	N/A	$2.4121e+01$	N/A	162.12
c-500-5-10-5	500	475	N/A	$6.3882e+02$	N/A	127.44
c-500-10-5-5	500	500	N/A	$1.0384e+02$	N/A	340.21
c-500-10-10-5	500	500	N/A	$1.3634e+03$	N/A	75.91
c-500-20-5-5	500	500	N/A	$1.2341e+02$	N/A	312.88
c-500-20-10-5	500	500	N/A	$3.9150e+03$	N/A	101.07
m-100-3-5	100	64	$3.0179e+01$	$3.0179e+01$	7.58	3.99
m-100-5-5	100	66	$9.6927e+01$	$9.6927e+01$	8.32	3.26
m-100-10-5	100	90	$5.1200e+02$	$5.1200e+02$	8.51	7.97
m-100-20-5	100	65	$3.3160e+02$	$3.3160e+02$	6.24	2.56
m-200-3-5	200	105	$3.3730e-01$	$3.3730e-01$	29.12	5.50
m-200-5-5	200	145	$3.3783e+02$	$3.3783e+02$	33.08	16.01
m-200-10-5	200	200	$1.5076e+03$	$1.5076e+03$	81.74	78.95
m-200-20-5	200	142	$1.6912e+03$	$1.6912e+03$	47.36	31.23
m-500-3-5	500	262	N/A	$2.1588e+01$	N/A	231.38
m-500-5-5	500	289	N/A	$6.1693e+02$	N/A	60.61
m-500-10-5	500	321	N/A	$1.8706e+03$	N/A	333.26
m-500-20-5	500	319	N/A	$4.2592e+03$	N/A	311.62
covtype-100	100	100	$2.8366e+03$	$2.8366e+03$	49.15	46.95
covtype-200	200	200	N/A	$7.1608e+03$	N/A	125.42
covtype-300	300	300	N/A	$1.0166e+04$	N/A	166.65
covtype-400	400	400	N/A	$1.4751e+04$	N/A	352.69
covtype-500	500	500	N/A	$1.8515e+04$	N/A	388.14
diabetes	768	768	N/A	$5.1287e+03$	N/A	50.80
glass	214	212	$1.2391e+03$	$1.2391e+03$	15.65	15.52
housing	506	506	N/A	$3.4527e+03$	N/A	841.39
ionosphere	351	350	N/A	$8.2617e+03$	N/A	340.94
lena32-22	256	253	N/A	$3.8644e+02$	N/A	659.36
lena32-44	64	64	N/A	$5.3126e+02$	N/A	17.52
lenses	24	24	$6.0000e+01$	$6.0000e+01$	0.12	0.11
machine	209	197	N/A	$7.7010e+02$	N/A	27.24
soybean	47	47	$3.6714e+02$	$3.6714e+02$	1.10	1.09
wine	178	178	$1.2779e+03$	$1.2779e+03$	17.16	16.29

The threshold for preclustering is chosen, emperically, such that it is the highest value that give the final costs no worse than the standalone Cyclic Exchange. This decision is made to ensure that, while decreasing the effective data sizes and speeding up the process, the qualities of solutions are not worsen by preclustering.

As shown in Table 5.2, two-point preclustering is able to speed up Continue Cyclic Exchange while maintaining the same final costs. It is uncertain if two-point preclustering would obtain the same final costs for the larger datasets that standalone Cyclic Exchange does not run on (shown as N/A in the table). In any case, standalone Cyclic Exchange runs for intractable amount of time for these datasets, so two-point preclustering is still advantageous to the standalone method.

A problem with this method is how to choose the threshold. Choosing the threshold emperically seems unattractive. A statistical approach may be applicable. For example, one can calculate a distribution of distances among all points and choose the preclustering threshold to be the tenth percentile of this distribution. Nevertheless, this calculation adds more preprocessing time to the overall process.

Birch Preclustering

The effective data size, final cost, and time for standalone Continue Cyclic Exchange and Continue Cyclic Exchange with Birch preclustering are shown in Table 5.3. The parameters for the construction of CF trees are chosen to be the same as those used in [14].

As shown in Table 5.3, Birch preclustering is able to speed up Continue Cyclic Exchange. For several datasets Birch preclustering brings down the running time to the same order of magnitude as Lloyd's algorithm. Unfortunately, the running time and the quality of solutions seem to be negatively correlated, as expected.

Implication from Preclustering

Speeding up Cyclic Exchange by decreasing effective data size is rather unattractive. The improvement on time is not enough to match Lloyd's algorithm, yet the qualities of solutions are negatively affected in most cases. In addition, this preprocessing takes

Table 5.3: Results for Continue Cyclic Exchange with Birch Preclustering

Dataset	Data Size		Final Cost		Time (seconds)	
	Original	Effective	Continue	Birch	Continue	Birch
c-100-3-5-5	100	21	$3.5151e+00$	$1.8121e+01$	5.63	0.05
c-100-5-5-5	100	22	$3.8248e+00$	$3.8248e+00$	4.94	0.11
c-100-10-5-5	100	26	$1.0390e+01$	$1.0390e+01$	4.31	0.24
c-100-20-5-5	100	27	$1.9527e+01$	$1.9527e+01$	4.28	0.41
c-200-3-5-5	200	44	$7.0512e+00$	$7.0333e+01$	94.27	0.73
c-200-3-10-5	200	44	$1.0279e+02$	$1.2551e+02$	37.98	0.36
c-200-5-5-5	200	49	$1.9782e+01$	$1.9782e+01$	18.24	0.68
c-200-5-10-5	200	53	$1.7848e+02$	$2.0685e+02$	14.06	0.61
c-200-10-5-5	200	47	$1.8564e+01$	$5.4075e+01$	16.98	0.56
c-200-10-10-5	200	50	$6.3048e+02$	$6.3048e+02$	16.31	0.64
c-200-20-5-5	200	50	$6.1182e+01$	$6.1182e+01$	30.63	1.03
c-200-20-10-5	200	54	$1.6607e+03$	$1.6929e+03$	8.53	1.05
c-500-3-5-5	500	122	N/A	$1.9828e+01$	N/A	4.83
c-500-3-10-5	500	110	N/A	$3.1384e+02$	N/A	11.88
c-500-5-5-5	500	117	N/A	$2.4121e+01$	N/A	6.80
c-500-5-10-5	500	126	N/A	$7.0968e+02$	N/A	4.87
c-500-10-5-5	500	114	N/A	$1.0384e+02$	N/A	8.70
c-500-10-10-5	500	113	N/A	$1.3634e+03$	N/A	3.53
c-500-20-5-5	500	121	N/A	$1.2341e+02$	N/A	8.89
c-500-20-10-5	500	130	N/A	$3.9620e+03$	N/A	5.92
m-100-3-5	100	17	$3.0179e+01$	$4.2431e+01$	7.58	0.04
m-100-5-5	100	24	$9.6927e+01$	$1.1834e+02$	8.32	0.18
m-100-10-5	100	37	$5.1200e+02$	$5.1164e+02$	8.51	1.86
m-100-20-5	100	18	$3.3160e+02$	$3.3160e+02$	6.24	0.13
m-200-3-5	200	31	$3.3730e-01$	$2.3787e+01$	29.12	0.14
m-200-5-5	200	53	$3.3783e+02$	$3.5757e+02$	33.08	1.27
m-200-10-5	200	164	$1.5076e+03$	$1.5096e+03$	81.74	78.33
m-200-20-5	200	58	$1.6912e+03$	$1.6912e+03$	47.36	6.22
m-500-3-5	500	64	N/A	$3.0284e+01$	N/A	0.52
m-500-5-5	500	107	N/A	$6.3402e+02$	N/A	6.63
m-500-10-5	500	149	N/A	$1.8591e+03$	N/A	60.48
m-500-20-5	500	130	N/A	$4.2539e+03$	N/A	72.43
covtype-100	100	42	$2.8366e+03$	$2.8589e+03$	49.15	10.20
covtype-200	200	68	N/A	$7.2609e+03$	N/A	31.41
covtype-300	300	91	N/A	$1.0348e+04$	N/A	62.57
covtype-400	400	114	N/A	$1.4967e+04$	N/A	77.96
covtype-500	500	130	N/A	$1.8898e+04$	N/A	64.14
diabetes	768	333	N/A	$5.1731e+03$	N/A	11.11
glass	214	53	$1.2391e+03$	$1.2721e+03$	15.65	1.27
housing	506	115	N/A	$3.6278e+03$	N/A	38.35
ionosphere	351	155	N/A	$8.3409e+03$	N/A	88.19
lena32-22	256	73	N/A	$4.3771e+02$	N/A	8.67
lena32-44	64	34	N/A	$5.7391e+02$	N/A	2.25
lenses	24	16	$6.0000e+01$	$6.0000e+01$	0.12	0.04
machine	209	34	N/A	$7.8528e+02$	N/A	0.85
soybean	47	10	$3.6714e+02$	$4.2975e+02$	1.10	0.08
wine	178	81	$1.2779e+03$	$1.2817e+03$	17.16	3.28

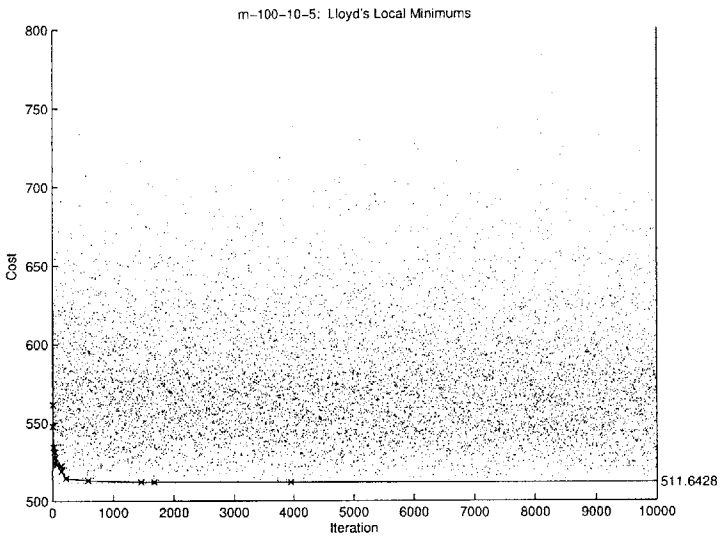


Figure 5-3: Costs of Local Minima from Lloyd's Algorithm in 10000 Iterations: The line at the bottom indicates the best local minimum found upto each iteration.

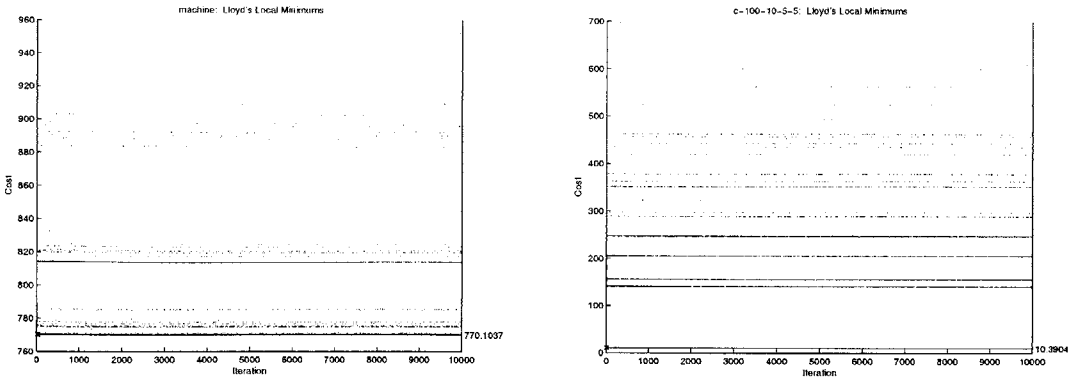
time and hence adds up to the overall complexity of Cyclic Exchange.

At this point, we are more interested to verify whether Cyclic Exchange can get any better solutions than Lloyd's algorithm. The time complexity of Cyclic Exchange is impractical. We will only try to get better solution quality from Cyclic Exchange. This leads us to the next experiment on the two-stage algorithm, where we study Lloyd's algorithm more closely and find a way to improve it.

5.3 Two-stage Algorithm

5.3.1 Methods

The unsuccessful attempts in speeding up Cyclic Exchange by decreasing the effective data size suggest that the time complexity of Cyclic Exchange is far to high to be comparable with Lloyd's algorithm. However, the fact that Cyclic Exchange is capable of finding a better solution still holds. In this section, we present a scheme for combining Lloyd's algorithm and Cyclic Exchange, to the so-called the two-stage algorithm. The two-stage algorithm is motivated by an observation that Lloyd's algorithm does not find a better solution after some number of iterations.



(a) (b)

Figure 5-4: Structures in the Costs of Local Minima from Lloyd's Algorithm in 10000 Iterations: (a) Dataset `machine` (from application). (b) Dataset `c-100-10-10-5` (synthetic).

An iteration of Lloyd's algorithm runs very quickly, and this advantage allows us to run Lloyd's algorithm for large number of iterations. However, the best local minimum that Lloyd's algorithm finds usually emerges at early iterations. We conduct an experiment whereby Lloyd's algorithm is run for 10000 iterations. Figure 5-3 shows the costs of the local minima found in 10000 iterations and a line indicating the best local minimum found upto each iteration, for the dataset `m-100-10-5`. After approximately 4000 iterations, Lloyd's algorithm does not find any better local minima. This observation suggests that running excessive iterations of Lloyd's algorithm is unlikely to be better than running only moderate number of iterations. In the extreme, running Lloyd's algorithm forever is unlikely to be useful. Table 5.4 shows cost decreases of the best local minima found in 10000 iterations, in 1000 iterations intervals. For most datasets, no improvement is found after 1000 iterations.

Our study on the local minima from Lloyd's algorithm also shows that, for several datasets, the costs of local minima from Lloyd's algorithm are group into levels. Figure 5-4 shows the results from an application dataset and a synthetic dataset. Appendix D gives the graphs for all datasets. This leveling of local minima costs suggests that local minima from Lloyd's algorithm have some kind of structures. Running more iterations of Lloyd's algorithm would essentially gives a local minimum in one of levels. Since all local minima in the same level have costs differed by only a

Table 5.4: The Cost Decreases for Lloyd's Algorithm in 10000 Iterations: The number of clusters for each dataset is the same as in Table 5.1. Cost decreases after 2000 iterations are compared with the best cost after 1000 iterations.

Dataset	Cost after 1000 Iter.	Percentage Cost Decreases from Previous Interval (1000 iterations each)									Time (seconds)
		2	3	4	5	6	7	8	9	10	
c-100-10-5-5	$1.0390e+01$	0	0	0	0	0	0	0	0	0	4.89
c-100-20-5-5	$1.9527e+01$	0	0	0	0	0	0	0	0	0	6.04
c-100-3-5-5	$3.5151e+00$	0	0	0	0	0	0	0	0	0	4.63
c-100-5-5-5	$3.8248e+00$	0	0	0	0	0	0	0	0	0	4.26
c-200-10-10-5	$6.3048e+02$	0	0	0	0	0	0	0	0	0	9.19
c-200-10-5-5	$1.8564e+01$	0	0	0	0	0	0	0	0	0	10.51
c-200-20-10-5	$1.6607e+03$	0	0	0	0	0	0	0	0	0	12.35
c-200-20-5-5	$6.1182e+01$	0	0	0	0	0	0	0	0	0	14.32
c-200-3-10-5	$1.0279e+02$	0	0	0	0	0	0	0	0	0	8.54
c-200-3-5-5	$7.0512e+00$	0	0	0	0	0	0	0	0	0	9.14
c-200-5-10-5	$1.7848e+02$	0	0	0	0	0	0	0	0	0	7.96
c-200-5-5-5	$1.9782e+01$	0	0	0	0	0	0	0	0	0	9.78
c-500-10-10-5	$1.3634e+03$	0	0	0	0	0	0	0	0	0	24.24
c-500-10-5-5	$1.0384e+02$	0	0	0	0	0	0	0	0	0	31.60
c-500-20-10-5	$3.9150e+03$	0	0	0	0	0	0	0	0	0	29.84
c-500-20-5-5	$1.2341e+02$	0	0	0	0	0	0	0	0	0	39.93
c-500-3-10-5	$2.3745e+02$	0	0	0	0	0	0	0	0	0	23.64
c-500-3-5-5	$1.9828e+01$	0	0	0	0	0	0	0	0	0	24.50
c-500-5-10-5	$6.3882e+02$	0	0	0	0	0	0	0	0	0	21.88
c-500-5-5-5	$2.4121e+01$	0	0	0	0	0	0	0	0	0	24.93
m-100-10-5	$5.1308e+02$	−0.2110	0	−0.0689	0	0	0	0	0	0	7.71
m-100-20-5	$3.3160e+02$	0	0	0	0	0	0	0	0	0	7.46
m-100-3-5	$3.0179e+01$	0	0	0	0	0	0	0	0	0	4.72
m-100-5-5	$9.6927e+01$	0	0	0	0	0	0	0	0	0	5.05
m-200-10-5	$1.5065e+03$	−0.1570	−0.0775	0	−0.1494	0	0	0	0	0	28.93
m-200-20-5	$1.6912e+03$	0	0	0	0	0	0	0	0	0	16.35
m-200-3-5	$3.3730e-01$	0	0	0	0	0	0	0	0	0	10.24
m-200-5-5	$3.3783e+02$	0	0	0	0	0	0	0	0	0	11.37
m-500-10-5	$1.8488e+03$	0	0	0	0	0	0	0	0	0	46.72
m-500-20-5	$4.2527e+03$	0	0	0	0	0	0	0	0	0	52.26
m-500-3-5	$2.1588e+01$	0	0	0	0	0	0	0	0	0	30.87
m-500-5-5	$6.1693e+02$	0	0	0	0	0	0	0	0	0	30.82
covtype-100	$2.8626e+03$	−0.3741	−0.2665	0	0	0	0	0	0	0	23.23
covtype-200	$7.1208e+03$	−0.0229	0	0	0	0	0	0	0	0	50.71
covtype-300	$1.0168e+04$	0	0	−0.0367	0	−0.0176	0	0	−0.0020	0	76.09
covtype-400	$1.4723e+04$	0	0	0	0	0	0	0	−0.0922	0	110.43
covtype-500	$1.8464e+04$	0	0	0	0	0	0	0	−0.0218	0	136.70
diabetes	$5.1287e+03$	0	0	0	0	0	0	0	0	0	41.46
glass	$1.2391e+03$	0	0	0	0	0	0	0	0	0	15.75
housing	$3.4527e+03$	0	0	0	0	0	0	0	0	0	66.10
ionosphere	$8.2617e+03$	0	0	0	0	0	0	0	0	0	50.29
lena32-22	$3.8644e+02$	0	0	0	0	0	0	0	0	0	20.09
lena32-44	$5.3126e+02$	0	0	0	0	0	0	0	0	0	6.21
lenses	$6.0000e+01$	0	0	0	0	0	0	0	0	0	1.20
machine	$7.7010e+02$	0	0	0	0	0	0	0	0	0	12.69
soybean	$3.6714e+02$	0	0	0	0	0	0	0	0	0	4.86
wine	$1.2779e+03$	0	0	0	0	0	0	0	0	0	11.82
zoo	$9.6725e+02$	0	0	0	0	0	0	0	0	0	5.85

small fraction, the solutions (center of clusters) themselves must differ by only small parts, and they may be obtained from one another via local search. Therefore, for these kinds of datasets, we are interested in applying Cyclic Exchange to local minima only in the lowest level.

The first stage of the two-stage algorithm runs Lloyd's algorithm for 1000 iterations and stores the 100 lowest-cost local minima. In the second stage, these 100 local minima are used as initial solutions for Continue Cyclic Exchange. We expect that two-stage algorithm, with 1000 iterations of Lloyd's algorithm, will be no worse than 10000 iterations of purely Lloyd's algorithm and will finish earlier or in comparable time.

5.3.2 Results and Discussion

The summary of results for two-stage algorithm is shown in Table 5.5. Stage 2 can decrease the costs from stage 1 in seven datasets, and the costs after stage 2 are better than 10000 iterations of Lloyd's algorithm in three datasets. Although the improvement between two stages and the improvement beyond 10000 iterations of Lloyd's algorithm are less than one percent, they reveal to us some interesting insights of these datasets.

In all other datasets that stage 2 does not improve stage 1, the costs after stage 1 also match the cost from 10000 iterations of Lloyd's algorithm. This fact is actually revealed in Table 5.4 as Lloyd's algorithm stops finding better solutions after 1000 iterations. On the other hands, for some `multiclus` synthetic data and `covtype` application data, Lloyd's algorithm does make improvement after 1000 iterations, and stage 2 also makes improvement. This result suggests that there are some characteristics of datasets that are hard, in some sense, for k-means clustering. We have tried the experiment on more applications datasets but were not able to find ones that give similar results. Further studies are necessary for identifying the underlying structure of these data, and the knowledge of such structures will be crucial in developing a better k-means clustering algorithm.

Table 5.5: Results for Two-stage Algorithm

Dataset	Two-stage Algorithm				10000 Iter. of Lloyd's	
	Stage 1	Stage 2	%Diff.	Time (s)	Cost	Time (s)
c-100-10-5-5	$1.0390e+01$	$1.0390e+01$	0.0000	5.13	$1.0390e+01$	4.89
c-100-20-5-5	$1.9527e+01$	$1.9527e+01$	0.0000	7.22	$1.9527e+01$	6.04
c-100-3-5-5	$3.5151e+00$	$3.5151e+00$	0.0000	8.41	$3.5151e+00$	4.63
c-100-5-5-5	$3.8248e+00$	$3.8248e+00$	0.0000	6.34	$3.8248e+00$	4.26
c-200-10-10-5	$6.3048e+02$	$6.3048e+02$	0.0000	5.61	$6.3048e+02$	9.19
c-200-10-5-5	$1.8564e+01$	$1.8564e+01$	0.0000	29.75	$1.8564e+01$	10.51
c-200-20-10-5	$1.6607e+03$	$1.6607e+03$	0.0000	4.63	$1.6607e+03$	12.35
c-200-20-5-5	$6.1182e+01$	$6.1182e+01$	0.0000	37.85	$6.1182e+01$	14.32
c-200-3-10-5	$1.0279e+02$	$1.0279e+02$	0.0000	53.46	$1.0279e+02$	8.54
c-200-3-5-5	$7.0512e+00$	$7.0512e+00$	0.0000	67.31	$7.0512e+00$	9.14
c-200-5-10-5	$1.7848e+02$	$1.7848e+02$	0.0000	20.42	$1.7848e+02$	7.96
c-200-5-5-5	$1.9782e+01$	$1.9782e+01$	0.0000	29.18	$1.9782e+01$	9.78
c-500-10-10-5	$1.3634e+03$	$1.3634e+03$	0.0000	51.83	$1.3634e+03$	24.24
c-500-10-5-5	$1.0384e+02$	$1.0384e+02$	0.0000	332.38	$1.0384e+02$	31.60
c-500-20-10-5	$3.9150e+03$	$3.9150e+03$	0.0000	23.55	$3.9150e+03$	29.84
c-500-20-5-5	$1.2341e+02$	$1.2341e+02$	0.0000	488.36	$1.2341e+02$	39.93
c-500-3-10-5	$2.3745e+02$	$2.3745e+02$	0.0000	431.21	$2.3745e+02$	23.64
c-500-3-5-5	$1.9828e+01$	$1.9828e+01$	0.0000	252.95	$1.9828e+01$	24.50
c-500-5-10-5	$6.3882e+02$	$6.3882e+02$	0.0000	92.75	$6.3882e+02$	21.88
c-500-5-5-5	$2.4121e+01$	$2.4121e+01$	0.0000	271.58	$2.4121e+01$	24.93
m-100-10-5	$5.1308e+02$	$5.1164e+02$	-0.2798	4.63	$5.1164e+02$	7.71
m-100-20-5	$3.3160e+02$	$3.3160e+02$	0.0000	3.75	$3.3160e+02$	7.46
m-100-3-5	$3.0179e+01$	$3.0179e+01$	0.0000	7.85	$3.0179e+01$	4.72
m-100-5-5	$9.6927e+01$	$9.6927e+01$	0.0000	8.66	$9.6927e+01$	5.05
m-200-10-5	$1.5065e+03$	$1.5046e+03$	-0.1296	44.14	$1.5008e+03$	28.93
m-200-20-5	$1.6912e+03$	$1.6912e+03$	0.0000	21.62	$1.6912e+03$	16.35
m-200-3-5	$3.3730e-01$	$3.3730e-01$	0.0000	45.79	$3.3730e-01$	10.24
m-200-5-5	$3.3783e+02$	$3.3783e+02$	0.0000	16.73	$3.3783e+02$	11.37
m-500-10-5	$1.8488e+03$	$1.8488e+03$	0.0000	360.88	$1.8488e+03$	46.72
m-500-20-5	$4.2527e+03$	$4.2527e+03$	0.0000	175.24	$4.2527e+03$	52.26
m-500-3-5	$2.1588e+01$	$2.1588e+01$	0.0000	759.62	$2.1588e+01$	30.87
m-500-5-5	$6.1693e+02$	$6.1693e+02$	0.0000	226.92	$6.1693e+02$	30.82
covtype-100	$2.8626e+03$	$2.8362e+03$	-0.9230	35.46	$2.8443e+03$	23.23
covtype-200	$7.1208e+03$	$7.1194e+03$	-0.0188	71.38	$7.1191e+03$	50.71
covtype-300	$1.0168e+04$	$1.0159e+04$	-0.0878	105.00	$1.0162e+04$	76.09
covtype-400	$1.4723e+04$	$1.4718e+04$	-0.0375	184.94	$1.4709e+04$	110.43
covtype-500	$1.8464e+04$	$1.8459e+04$	-0.0276	251.30	$1.8460e+04$	136.70
diabetes	$5.1287e+03$	$5.1287e+03$	0.0000	55.75	$5.1287e+03$	41.46
glass	$1.2391e+03$	$1.2391e+03$	0.0000	12.76	$1.2391e+03$	15.75
housing	$3.4527e+03$	$3.4527e+03$	0.0000	2143.81	$3.4527e+03$	66.10
ionosphere	$8.2617e+03$	$8.2617e+03$	0.0000	152.95	$8.2617e+03$	50.29
lena32-22	$3.8644e+02$	$3.8644e+02$	0.0000	405.01	$3.8644e+02$	20.09
lena32-44	$5.3126e+02$	$5.3126e+02$	0.0000	11.85	$5.3126e+02$	6.21
lenses	$6.0000e+01$	$6.0000e+01$	0.0000	0.17	$6.0000e+01$	1.20
machine	$7.7010e+02$	$7.7010e+02$	0.0000	60.64	$7.7010e+02$	12.69
soybean	$3.6714e+02$	$3.6714e+02$	0.0000	2.51	$3.6714e+02$	4.86
wine	$1.2779e+03$	$1.2779e+03$	0.0000	47.12	$1.2779e+03$	11.82
zoo	$9.6725e+02$	$9.6725e+02$	0.0000	8.50	$9.6725e+02$	5.85

Chapter 6

Conclusions

While Cyclic Exchange can reduce the cost of a local minimum for Lloyd's algorithm, so can running Lloyd's algorithm from multiple starting positions. Except in a few cases, the final costs for Cyclic Exchange are not lower than those from Lloyd's algorithm with multiple starts. Although the results did not establish usefulness of Cyclic Exchange for the k-means clustering, our experiment did reveal insights that may be useful in developing other algorithms.

The two-stage algorithm is a way to improve Lloyd's algorithm by first running Lloyd's algorithm for some iterations, e.g. 1000, then using a subset of local minima found by Lloyd's algorithm as initial solutions for some further search techniques. The results have shown that the neighborhoods generated by Cyclic Exchange do not lead to further improvement in general. We believe that for the new search algorithm to be successful, it will need to be able to move multiple points together. Preclustering may be used to allow multiple points to move as a single unit.

The results from the two-stage algorithm show a few datasets for which Cyclic Exchange performs better. These datasets are in high dimension, and Lloyd's algorithm finds better solutions even after a few thousand iterations. We were unable to identify features that lead the Cyclic Exchange to perform better.

Cyclic Exchange does not scale well because of its computational complexity. It involves subset-disjoint negative-cost cycle detection. In constrast, Lloyd's algorithm is very simple to implement and to understand, and each iteration is really fast. This

gives it an advantage of running a large number of iterations. In our datasets, Lloyd's algorithm converges to its best solution within about 1000 iterations.

Appendix A

Codes for Generating Synthetic Data

The two types of synthetic data described in Section 3.3.1 are generated using the following Matlab codes.

Generating Data from the Gaussian Distribution

```
% function [xy] = genGauss( means, sigma, num )
function [xy] = genGauss( means, sigma, num )

d = size(means,2);
xy = zeros(sum(num),d);
s = 0;

for i=1:length(num)
    n = num(i);
    m = ones(n,d) * diag(means(i,:));
    r = ones(n,d) * diag(sigma(i,:));
    xy(s+1:s+n,:) = m + r.*randn([n d]);
    s = s+n;
end

% shuffle the order
xy = xy(randperm(size(xy,1)),:);
```

ClusGauss Datasets

```
% function [data] = clusGauss(n,d,k,sigma)
function [data] = clusGauss(n,d,k,sigma)

% each cluster has roughly equal size
num = round(n/k)*ones(k,1);
num(end) = n-sum(num(1:end-1));

% center is uniformedly distributed in [-1,1]^d
centers = rand(k,d)*2 - 1;

data = genGauss(centers,sigma*ones(k,1),num);
```

MultiClus Datasets

```
% function [data] = multiClus(n,d,baseSigma)
function [data] = multiClus(n,d,baseSigma)

% cluster sizes are 2^b where Prob{b=b} = 1/2^b;
num = [];
k = 0;
remain = n;
while remain>0
  numk = 2^(random('geo',.5,1,1)+1);
  if numk>remain
      numk = remain;
  end
  num = [num; numk];
  k = k+1;
  remain = remain - numk;
end

% standard deviation of cluster of size m is baseSigma/sqrt(m)
sigma = ones(k,1)*baseSigma ./ sqrt(num);

% center is uniformedly distributed in [-1,1]^d
centers = rand(k,d)*2 - 1;

data = genGauss(centers,sigma,num);
```

Appendix B

Source Codes

The following source codes implement the software design in Chapter 4. The structure of the software is repeated below for reference.

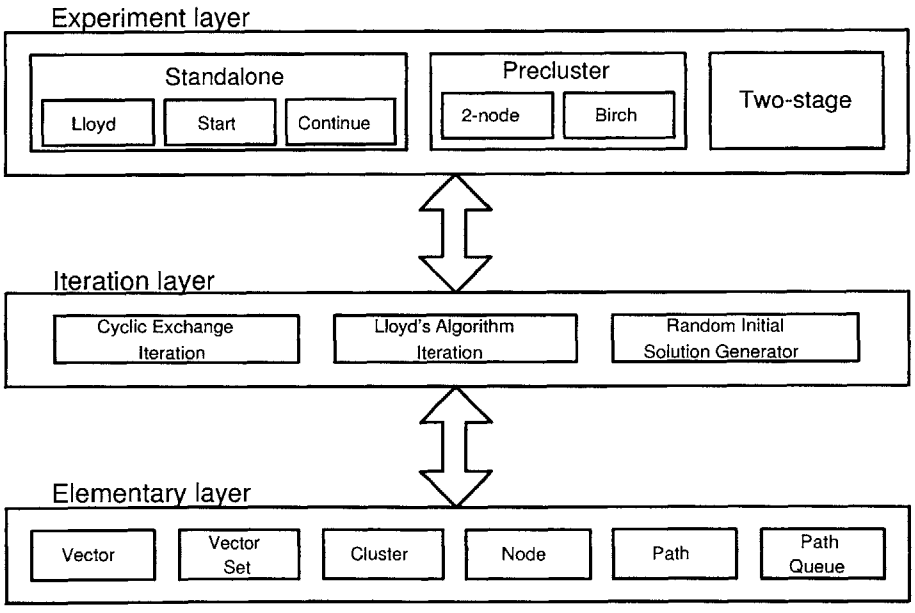


Figure B-1: Layered Structure of the Software

Elementary Layer

Class `Vector` and Class `VectorSet`

Vector.hh

```
#ifndef _VECTOR_H
#define _VECTOR_H

#include <iostream>
#include <string>
#include "global.hh"
using namespace std;

typedef double CoorType;

class Vector {
  friend class VectorSet;
private:
  int dim;
#ifndef TEST_VECTOR
  CoorType *component;
#endif
  double magSq;
  void calcMagSq();
public:
#ifdef TEST_VECTOR
  CoorType *component;
#endif
  Vector();
  Vector(int dimension);
  Vector(const Vector&);
  ~Vector();
  int dimension() const;
  double magnitudeSq() const;
  double distanceSqTo(const Vector&) const;
  unsigned int hashCode() const;
  friend istream &operator>>(istream &, Vector &);
  friend ostream &operator<<(ostream &, const Vector &);
  void add(const Vector&);
  void subtract(const Vector&);
  void scale(double);
  double dot(const Vector&) const;
  void nullify();
  int cluster;
  bool operator>(Vector &);
  VectorSet *chopUp(int k);
};

class VectorSet {
private:
  int n,dim,m;
  Vector **element;
  Vector *meanVec;
public:
  VectorSet(int n, int d);
  ~VectorSet();
  static VectorSet *readDenseMatrix(istream &);
  static VectorSet *readDenseMatrix(istream &,bool normalize);
  void writeDenseMatrix(ostream &);
  Vector *get(int index) const;
  Vector mean() const;
  int dimension() const;
  int size() const;
  friend ostream &operator<<(ostream &,const VectorSet&);
  bool full() const;
  void append(Vector v);
  void truncate(int i);
  void lexSort();
  Vector collapse();
};

inline double square(double a) { return a*a; }

#endif
```

Vector.cc

```
#include "Vector.hh"
#include <iomanip>
#include <cmath>

/********** VECTOR **********/

/***** Constructor / Destructor *****/

Vector::Vector() {
  dim = 0;
  component = NULL;
  magSq = 0;
  cluster = -1;
}

Vector::Vector(int d) {
  dim = d;
  component = new CoorType[d];
  memset(component,0,d*sizeof(CoorType));
  magSq = 0;
  cluster = -1;
}

Vector::Vector(Vector const &v) {
  dim = v.dim;
  if(dim!=0) {
    component = new CoorType[dim];
    memcpy(component, v.component, dim*sizeof(CoorType));
  }
  magSq = v.magSq;
  cluster = v.cluster;
}

Vector::~Vector() {
  if(component!=NULL) delete component;
#ifdef DEBUG_VECTOR_1
  _vector_desCount++;
#endif
}

/***** Helper functions *****/

void Vector::calcMagSq() {
  magSq = 0;
  for(int i=0; i<dim; i++)
    magSq += component[i] * component[i];
}

/***** Accessors *****/

int Vector::dimension() const { return dim; }

double Vector::magnitudeSq() const { return magSq; }

/***** Queries *****/

double Vector::distanceSqTo(const Vector &v) const {
  double sum = 0;
  double diff;
  for(int i=0; i<dim; i++) {
    diff = component[i] - v.component[i];
    sum += diff*diff;
  }
  return sum;
}

unsigned int Vector::hashCode() const {
  unsigned int sum = 0;
  for(int i=0; i<dim; i++) {
    sum = sum*17 + (int)(component[i]*component[i]/magSq*1000);
  }
  return sum;
}

/***** Mutators *****/

void Vector::add(const Vector &v) {
  if(component==NULL) {
    dim = v.dim;
    if(dim!=0) {
      component = new CoorType[dim];
      memset(component,0,dim*sizeof(CoorType));
    } else return;
  }
  for(int i=0; i<dim; i++)
    component[i] += v.component[i];
  calcMagSq();
}
```

```
void Vector::subtract(const Vector &v) {
  if(component==NULL) {
    dim = v.dim;
    if(dim!=0) {
      component = new CoorType[dim];
      memset(component,0,dim*sizeof(CoorType));
    } else return;
  }
  for(int i=0; i<dim; i++)
    component[i] -= v.component[i];
  calcMagSq();
}

void Vector::scale(double a) {
  for(int i=0; i<dim; i++)
    component[i] *= a;
  magSq *= a*a;
}

double Vector::dot(const Vector &v) const {
  double res = 0;
  for(int i=0; i<dim; i++) {
    res += component[i] * v.component[i];
  }
  return res;
}

void Vector::nullify() {
  for(int i=0; i<dim; i++)
    component[i] = 0;
  magSq = 0;
}

/***** Parser / Unparser *****/

istream &operator>>(istream &in, Vector &v) {
  CoorType x;
  for(int i=0; i<v.dim; i++) {
    in >> x;
    v.component[i] = x;
  }
  v.calcMagSq();
  return in;
}

ostream &operator<<(ostream &out, const Vector &v) {
  out << setprecision(VEC_PRINT_PRECISION) << fixed;
  out << '(' << v.component[0];
  for(int i=1; i<v.dim; i++)
    out << ',' << v.component[i];
  out << ')';
  return out;
}

bool Vector::operator>(Vector &v) {
  assert(dim==v.dim);
  double d;
  for(int i=0; i<dim; i++) {
    d = component[i]-v.component[i];
    if( d!=0 ) {
      if(d>0) return true;
      else return false;
    }
  }
  return false;
}

VectorSet *Vector::chopUp(int k) {
  assert(dim%k==0);
  VectorSet *res = new VectorSet(dim/k,k);
  Vector v(k);
  for(int i=0; i<dim/k; i++) {
    for(int j=0; j<k; j++) v.component[j] = component[i*k+j];
    res->append(v);
  }
  return res;
}


/********** VECTOR SET **********/

/***** Constructor / Destructor *****/

VectorSet::VectorSet(int n, int d) {
  element = new (Vector*)[n];
  this->n = n;
  dim = d;
  meanVec = new Vector(dim);
  m = 0;
}

VectorSet::~VectorSet() {
```

```
  delete meanVec;
  for(int i=0; i<m; i++)
    delete element[i];
  delete element;
}

/***** Accessors *****/

Vector *VectorSet::get(int i) const { return element[i]; }

Vector VectorSet::mean() const { return *meanVec; }

int VectorSet::dimension() const { return dim; }

int VectorSet::size() const { return m; }

bool VectorSet::full() const { return (m==n); }

/***** Mutator *****/

void VectorSet::append(Vector v) {
  meanVec->scale(m);
  element[m++] = new Vector(v);
  meanVec->add(v);
  meanVec->scale(1./m);
}

/***** Parser / Unparser *****/

ostream &operator<<(ostream &out, const VectorSet &vs) {
  out << "(VectorSet: size=" << vs.m << "; elements={";
  if(vs.n>0) out << *vs.element[0];
  for(int i=1; i<vs.m; i++)
    out << ',' << *vs.element[i];
  out << "})";
  return out;
}

VectorSet *VectorSet::readDenseMatrix(istream &in) {
  return readDenseMatrix(in,false);
}

void VectorSet::writeDenseMatrix(ostream &out) {
  out << m << ' ' << dim << endl;
  out << setprecision(15) << scientific;
  for(int i=0; i<m; i++) {
    for(int d=0; d<dim; d++) {
      out << element[i]->component[d] << ' ';
    }
    out << endl;
  }
}

VectorSet *VectorSet::readDenseMatrix(istream &in, bool normalize) {
  int n,d;
  in >> n >> d;
  assert(in.good());
  VectorSet *result = new VectorSet(n,d);
  Vector *v;
  for(int i=0; i<n; i++) {
    assert(in.good());
    v = new Vector(d);
    in >> *v;
    result->element[i] = v;
    result->meanVec->add(*v);
  }
  if(n>0) result->meanVec->scale(1./n);
  result->m = n;

  if(normalize) {
    Vector sd(d);
    // sigma x^2
    for(int j=0; j<d; j++) {
      for(int i=0; i<n; i++) {
        sd.component[j] += square(result->element[i]->component[j]);
      }
    }
    // sd = sqrt( sigma x^2 / N  - x_bar^2 )
    for(int j=0; j<d; j++) {
      sd.component[j] = sqrt(sd.component[j]/n
                             - square(result->meanVec->component[j]) );
    }

    for(int i=0; i<n; i++) {
      result->element[i]->subtract(*result->meanVec);
      for(int j=0; j<d; j++) {
        if(sd.component[j]>0) result->element[i]->component[j] /= sd.component[j];
      }
    }

    result->meanVec->nullify();
    // recalculate mean (redundant)
```

```
    //    for(int i=0; i<n; i++)  result->meanVec->add(*result->get(i));

  }

  return result;
}

void VectorSet::lexSort() {
  // lexicographical sort
  // by insertion sort
  Vector *temp;
  for(int i=m-2; i>=0; i--) {
    temp = element[i];
    int j = i+1;
    while( (j<m) && (*temp>*element[j]) ) {
      element[j-1] = element[j];
      j++;
    }
    element[j-1] = temp;
  }
}

Vector VectorSet::collapse() {
  Vector res(dim*m);
  for(int i=0; i<m; i++)
    for(int j=0; j<dim; j++)
      res.component[i*dim+j] = element[i]->component[j];
  return res;
}

void VectorSet::truncate(int i) {
  if(i<m) m=i;
}
```

Class `Cluster` and Class `Node`

Cluster.hh

```
#ifndef _CLUSTER_H
#define _CLUSTER_H

#include <iostream>
#include <string>
#include "global.hh"
#include "Vector.hh"
using namespace std;

class Node {
  friend class Cluster;
  friend class Path;
  friend class Cyclic;
private:
  unsigned int hCode;
  inline void computeHashCode();
protected:
  Vector *data; // data==NULL mean dummy
  Node *next,*prev;
  int cluster;
public:

  Node *nextMini;
  int countMini;
  Vector *tsMini;

  Node(Vector *dat);
  Node(int k);
  ~Node();
  unsigned int hashCode() const; // stored
  void removeFrom(Cluster *owner) const;
  friend ostream &operator<<(ostream &, const Node &);
  friend ostream &operator<<(ostream &, Cluster &);
  friend ostream &operator<<(ostream &, const Path &);
};

class Cluster {
  friend class Node;
protected:
  int id;
private:
  int count;
  Vector *ts;
  Node *current, *lastReturned; // for iterator
  bool lastRemoved; // for iterator
public:
```

```
  Node *member;
  Cluster(int k, int d);
  ~Cluster();
  int size() const;
  Vector totalSum() const;
  Vector mean() const;
  void add(Node*);
  void initIterator();
  bool hasNext();
  Node *next();
  void removeIt(); // remove the node last returned by next()
  friend ostream &operator<<(ostream &, Cluster &);
};

#endif
```

Cluster.cc

```
#include "Cluster.hh"
#include <cassert>

/********** NODE **********/

#ifdef DEBUG_NODE_1
int _node_conCount = 0;
int _node_desCount = 0;
#endif

Node::Node(int k) {
  // dummy node for cluster k
  data = NULL;
  prev = this;
  next = this;
  cluster = k;
  computeHashCode();
#ifdef DEBUG_NODE_1
  _node_conCount++;
#endif
  nextMini = NULL;
  countMini = 1;
  tsMini = NULL;
}

Node::Node(Vector *dat) {
  data = dat;
  prev = this;
  next = this;
  cluster = data->cluster;
  computeHashCode();
#ifdef DEBUG_NODE_1
  _node_conCount++;
#endif
  nextMini = NULL;
  countMini = 1;
  tsMini = new Vector(*dat);
}

Node::~Node() {
#ifdef DEBUG_NODE_1
  _node_desCount++;
#endif
  delete tsMini;
}

void Node::removeFrom(Cluster *owner) const {
  // require this belongs to owner
  prev->next = next;
  next->prev = prev;
  owner->count -= countMini;
  if(owner->count==0) owner->ts->nullify();
  else owner->ts->subtract(*tsMini);
}

/***** Hash *****/
inline void Node::computeHashCode() {
  if(data==NULL) {
    hCode = (unsigned int)cluster*11;
  } else {
    hCode = data->hashCode();
  }
}

unsigned int Node::hashCode() const {
  return hCode;
}

/***** Unparser *****/
```

```
// print the vector
ostream &operator<<(ostream &out, const Node &node) {
  out << *node.data;
  return out;
}

/********** CLUSTER **********/

/***** Constructor / Destructor *****/

Cluster::Cluster(int k, int d) {
  member = new Node(k); // sentinel
  id = k;
  count = 0;
  ts = new Vector(d);
}

Cluster::~Cluster() {
  delete ts;
  Node *node,*next;
  node = member->next;
  for(int i=0; i<count; i++) {
    next = node->next;
    delete node;
    node = next;
  }
  delete node; // sentinel
}

/***** Accessor *****/

int Cluster::size() const { return count; }

Vector Cluster::totalSum() const { return *ts; }

Vector Cluster::mean() const {
  Vector a = *ts;
  a.scale(1./count);
  return a;
}

/***** Mutator *****/

void Cluster::add(Node *node) {
  // add to the end
  node->prev = member->prev;
  node->prev->next = node;
  node->next = member;
  member->prev = node;
  node->cluster = id;
  node->data->cluster = id;

  count += node->countMini;
  ts->add(*node->tsMini);
}

/***** Iterator *****/

void Cluster::initIterator() {
  current = member->next;
  lastReturned = NULL;
  lastRemoved = false;
}

bool Cluster::hasNext() {
  return current!=member;
}

Node *Cluster::next() {
  lastReturned = current;
  current = current->next;
  lastRemoved = false;
  return lastReturned;
}

void Cluster::removeIt() {
  assert(lastReturned!=NULL);
  assert(!lastRemoved);
  lastReturned->prev->next = lastReturned->next;
  lastReturned->next->prev = lastReturned->prev;
  count -= lastReturned->countMini;
  if(count==0) ts->nullify();
  else ts->subtract(*lastReturned->tsMini);
  lastRemoved = true;
}

/***** Unparser *****/
/**
   output format:
   *****************
   (Cluster $id
   size = $size;
```

```
   mean = $mean;
   members = {
   $v_1
   $v_2
   ...
   $v_size
   })*****************
 */
ostream &operator<<(ostream &out, Cluster &c) {
  Vector mean = c.totalSum();
  if(c.count>0) mean.scale(1./c.count);
  out << "(Cluster " << c.id << ":" << endl;
  out << "size = " << c.count << endl;
  out << "mean = " << mean << endl;
  out << "members = {" << endl;
  Node *node = c.member->next;
  while(node!=c.member) {
    out << *node << endl;
    node = node->next;
  }
  out << "})";
  return out;
}
```

Class Path

Path.hh

```
#ifndef _PATH_H
#define _PATH_H

#include <iostream>
#include "global.hh"
#include "Cluster.hh"
using namespace std;

class Path {
  friend class Cyclic;
protected:
  class Element {
    friend class Path;
    friend class Cyclic;
  protected:
    Node *node;
    Element *next;
  public:
    Element(Node *n, Element *nx);
    Element(Element &);
    ~Element();
    friend ostream &operator<<(ostream &, const Path&);
  };
  Element *tail, *head; // path goes from tail to head

private:
  int numCluster;
  int len;
  double co; // cost
  int labelWidth;
  unsigned int *label;
  inline void setLabel(int k);

public:
  Path(int k, Node *n1, Node *n2, double initCost);
  Path(Path &);
  ~Path();
  int length() const;
  double cost() const;
  bool noCluster(int k) const;
  unsigned int hashCode() const;
  bool sameKey(Path &p) const;
  void extend(Node *n, double incCost);
  friend ostream &operator<<(ostream &, const Path&);
  void checkRepCycle();
};

#endif
```

Path.cc

```
#include "Path.hh"
#include <cassert>
#include <iomanip>

/********** PATH ELEMENT **********/

Path::Element::Element(Node *n, Element*nx) {
  node = n;
  next = nx;
}

Path::Element::Element(Element &p) {
  node = p.node;
  next = p.next;
}

Path::Element::~Element() {
  // recursively destruct all the elements in this path
  if(next!=NULL) delete next;
}

/********** PATH **********/

/***** Constructor / Destructor *****/

Path::Path(int k, Node *n1, Node *n2, double initCost) {
  assert(n1->cluster!=n2->cluster); // XXX
  numCluster = k;
  // n1->n2
  head = new Element(n2,NULL);
  tail = new Element(n1,head);
  len = 1;
  co = initCost;

  // init label
  labelWidth = 1 + (numCluster>>5);  // div 32
  label = new (unsigned int)[labelWidth];
  memset(label,0,labelWidth*sizeof(unsigned int));
  setLabel(n1->cluster);
  setLabel(n2->cluster);
}

Path::Path(Path &p) {
  numCluster = p.numCluster;
  head = new Element(p.tail->next->node,NULL);
  tail = new Element(p.tail->node,head);
  len = 1;
  labelWidth = 1 + (numCluster>>5);  // div 32
  label = new (unsigned int)[labelWidth];

  Element *x = p.tail->next->next;
  while(x!=NULL) {
    extend(x->node,0.);
    x = x->next;
  }

  memcpy(label,p.label,labelWidth*sizeof(unsigned int));
  co = p.co;
}

Path::~Path() {
  delete label;
  delete tail; // recursive through all elements
}

/***** Helper *****/

inline void Path::setLabel(int k) {
  // set label[k div 32] at bit (k mod 32) to 1
  label[k>>5] = label[k>>5] | (1<<(k&0x1F));
}

/***** Accessor / Query *****/

int Path::length() const { return len; }

double Path::cost() const { return co; }

bool Path::noCluster(int k) const {
  return ((label[k>>5] & (1<<(k&0x1F))) == 0);
}

unsigned int Path::hashCode() const {
  assert(tail!=NULL);
  assert(head!=NULL);
  assert(tail->node!=NULL);
  assert(head->node!=NULL);
  unsigned int sum = tail->node->hashCode() + head->node->hashCode()*31;
  for(int i=0; i<labelWidth; i++)
```

```
    sum = sum*31 + label[i];
  return sum;
}

bool Path::sameKey(Path &p) const {
  if(tail->node != p.tail->node) return false;
  if(head->node != p.head->node) return false;
  assert(labelWidth == p.labelWidth);
  for(int i=0; i<labelWidth; i++)
    if(label[i] != p.label[i]) return false;
  return true;
}

/***** Mutator *****/

void Path::extend(Node *n, double incCost) {
  Element *p = new Element(n,NULL);
  head->next = p;
  head = p;
  len++;
  co += incCost;
  setLabel(n->cluster);
}

/***** Unparser *****/

ostream &operator<<(ostream &out, const Path &p) {
  out << "(Path:" << endl;
  out << "Length = " << p.len << endl;
  out << "Cost = " << setprecision(F_PRECISION) << scientific << p.co << endl;
  out << "Elements = {" << endl;
  Path::Element *x = p.tail;
  while(x!=NULL) {
    if(x->node->data==NULL) {
      out << "dummy(" << x->node->cluster << ')' << endl;
    } else {
      out << *x->node->data << '(' << x->node->cluster << ')' << endl;
      assert(x->node->cluster==x->node->data->cluster);
    }
    x = x->next;
  }
  out << "})";
  return out;
}

void Path::checkRepCycle() {
  assert(tail->node->cluster == head->node->cluster);
  int countLen = -1;
  bool *has = new bool[numCluster];
  memset(has,0,numCluster*sizeof(bool));
  Path::Element *x = tail;
  while(x!=NULL) {
    countLen ++;
    if(x->node->data==NULL) {
    } else {
      assert(x->node->cluster == x->node->data->cluster);
    }
    if(x!=head) assert(! has[x->node->cluster] );
    has[x->node->cluster] = true;
    x = x->next;
  }
  assert(countLen==len);
}
```

Class PathQueue

PathQueue.hh

```
#ifndef _PATHQUEUE_H
#define _PATHQUEUE_H

#include <iostream>
#include "global.hh"
#include "Path.hh"

class PathQueue {

protected:
  class Element {
  public:
    Path *p;
    Element *prev, *next;
    Element(Path* p, Element *pv, Element *nx);
    ~Element();
```

```
  };

  class HashNode {
  public:
    HashNode();
    ~HashNode();
    Element *elem;
    HashNode *next;
  };

  class HashTable {
    friend class PathQueue;
  private:
    int tabSize;
    HashNode **tab;
    int *count;
  public:
    HashTable(int n, int k);
    ~HashTable();
    Element *locate(Path *p); // locate q element that has the same key as p
    void add(Element *e);
    void remove(Element *e); // remove e from hash table. does not free e
  };

private:
  unsigned int count;
  Element *first, *last;
  HashTable *table;

public:
  PathQueue(int n, int k);
  ~PathQueue(); // also free all path in the queue
  int size() const;
  void enQueue(Path*); // substitute and free the dominated path (if exists)
  Path *deQueue();
  void clear(); // free all path and empty q and hash table

};

#endif
```

PathQueue.cc

```
#include "PathQueue.hh"
#include <cassert>

/********** PATH QUEUE ELEMENT **********/

PathQueue::Element::Element(Path* p, Element *pv, Element *nx) {
  this->p = p;
  prev = pv;
  next = nx;
}

PathQueue::Element::~Element() {
}

/********** HASH NODE **********/

PathQueue::HashNode::HashNode() {
}

PathQueue::HashNode::~HashNode() {
}

/********** HASH TABLE **********/

/***** Constructor / Destructor *****/

int max_count = 0;

PathQueue::HashTable::HashTable(int n, int k) {
  tabSize = n*n*k;
  tab = new (HashNode*)[tabSize];
  for(int i=0; i<tabSize; i++) tab[i] = NULL;
  count = new int[tabSize];
  memset(count,0,tabSize*sizeof(int));
}

PathQueue::HashTable::~HashTable() {
  HashNode *a,*b;
  for(int i=0; i<tabSize; i++) {
    b = tab[i];
    while(b!=NULL) {
      a = b;
      b = b->next;
      delete a;
```

```
    }
  }
  delete tab;
  delete count;
}

/***** Accessor / Query / Mutator *****/

PathQueue::Element *PathQueue::HashTable::locate(Path *p) {
  HashNode *a = tab[p->hashCode() % tabSize];
  while(a!=NULL) {
    if(a->elem->p->sameKey(*p))  {
      return a->elem;
    }
    a = a->next;
  }
  return NULL;
}

void PathQueue::HashTable::add(Element *e) {
  unsigned int h = e->p->hashCode() % tabSize;
  HashNode *a = new HashNode();
  a->elem = e;
  a->next = tab[h];
  tab[h] = a;
  count[h]++;
  if(count[h]>max_count) max_count = count[h];
}

void PathQueue::HashTable::remove(Element *e) {
  assert(e!=NULL);
  assert(e->p!=NULL);
  unsigned int h = e->p->hashCode() % tabSize;
  count[h]--;
  HashNode *a = NULL;
  HashNode *b = tab[h];
  while(b!=NULL) {
    if(b->elem == e) {
      if(a==NULL) { // the beginning of list
        tab[h] = b->next;
      } else {
        a->next = b->next;
      }
      delete b;
      return;
    }
    a = b;
    b = b->next;
  }
  assert(false);
}

/********** PATH QUEUE **********/

/***** Constructor / Destructor *****/

int max_queue_size = 0;

PathQueue::PathQueue(int n, int k) {
  count = 0;
  first = NULL;
  last = NULL;
  table = new HashTable(n,k);
}

PathQueue::~PathQueue() {
  clear();
  delete table;
}

/***** Accessor / Query *****/

int PathQueue::size() const { return count; }

/***** Mutator *****/

void PathQueue::enQueue(Path *p) {
  Element *el = table->locate(p);
  if( el==NULL ) {
    // no entry with the samekey, just add
    el = new Element(p,last,NULL);
    if( count==0 )
      first = el;
    else
      last->next = el;
    last = el;
    count++;
    table->add(el);

  } else {
    // compare the cost. more negative cost is better
    if( p->cost() < el->p->cost() ) { // p is better -> substitute
```

```
      Path *worse = el->p;
      el->p = p;
      delete worse;
    } else { // if p is worse, discard p
      delete p;
    }
  }

  if(count>max_queue_size) max_queue_size=count;
}

Path *PathQueue::deQueue() {
  assert(count>0);

  Path *result = first->p;
  Element *el = first;
  first = first->next;
  count--;

  table->remove(el);
  delete el;
  return result;
}

void PathQueue::clear() {
  Path *p;
  while( count>0 ) {
    p = deQueue();
    delete p;
  }
}
```

Birch Preclustering

CFTree.hh

```
#ifndef _CFTREE_H
#define _CFTREE_H

#include <iostream>
#include "global.hh"
#include "Vector.hh"
#include "Cyclic.hh"
using namespace std;

class CFEntry;
class CFNode;
class CFInternal;
class CFLeaf;
class CFTree;

class CFEntry {
 public:
  int n;
  Vector linearSum;
  double squaredSum;
  CFEntry();
  CFEntry(CFEntry const &e);
  CFEntry(Vector &v);
  ~CFEntry();
  void nullify();
  // get a sum of sq dist from v to all point in this entry
  double distanceSqTo(const Vector &v) const;
  double distanceSqTo(const CFEntry &e) const; // distance between centers
  double combineCost(const CFEntry &e) const; // the less, the better
  void add(const CFEntry &e);
  friend ostream &operator<<(ostream &,const CFEntry&);
};

class SplitNode {
  friend class CFNode;
  friend class CFInternal;
  friend class CFLeaf;
  friend class CFTree;
private:
  CFNode *child1, *child2;
  CFEntry *ent1, *ent2;
public:
  SplitNode() { child1=NULL; child2=NULL; };
  SplitNode(CFNode *c1, CFNode *c2, CFEntry *e1, CFEntry *e2) {
    child1=c1; child2=c2; ent1=e1; ent2=e2;
  };
};
```

```
class CFNode {
  friend class CFTree;
private:
protected:
  CFEntry **entry;
  int nEntry,maxEntry;
public:
  CFNode(int maxEntry);
  CFNode();
  virtual ~CFNode();
  int numEntry() const { return nEntry; };
  virtual SplitNode addEntry(CFEntry *e)=0;
  virtual void print(ostream&,int depth)=0;
  virtual void recursiveDelete();
};

class CFInternal : public CFNode {
  friend class CFTree;
private:
  CFNode **child;
public:
  CFInternal(int maxChild);
  ~CFInternal();
  virtual void recursiveDelete();
  virtual SplitNode addEntry(CFEntry *e);
  virtual void print(ostream&,int depth);
};

class CFLeaf : public CFNode {
  friend class CFTree;
  friend class Cyclic;
private:
  CFLeaf *prev, *next;
  double threshold;
public:
  CFLeaf(int maxEntry, double t, CFLeaf *inFront);
  CFLeaf();
  ~CFLeaf();
  virtual void recursiveDelete();
  virtual SplitNode addEntry(CFEntry *e);
  virtual void print(ostream&,int depth);
};

class CFTree {
  friend class Cyclic;
private:
  CFNode *root;
  CFLeaf *leafList;
  int maxInternalChild, maxLeafEntry;
  double leafThreshold;
  int nPoint;
public:
  CFTree(int maxInternalChild,int maxLeafEntry, double leafThreshold);
  ~CFTree();
  void addPoint(Vector v);
  //  void prepare(Cyclic *cyclic, int level);
  friend ostream &operator<<(ostream &,const CFTree&);
  int numLeaves();
  VectorSet *getLeavesCenters();
};

#endif
```

CFTree.cc

```
#include "CFTree.hh"

extern int _cf_maxInternalChild;
extern int _cf_maxLeafEntry;
extern double _cf_leafThreshold;

/***** CF Entry *****/
CFEntry::CFEntry() {
  n = 0;
  squaredSum = 0;
#ifdef DEBUG_CFENTRY_1
  _cfentry_conCount++;
#endif
}

CFEntry::CFEntry(CFEntry const &e) {
  n = e.n;
  //  linearSum = e.linearSum;
  linearSum.nullify();
  linearSum.add(e.linearSum);
  squaredSum = e.squaredSum;
#ifdef DEBUG_CFENTRY_1
```

```
  _cfentry_conCount++;
#endif
}

CFEntry::CFEntry(Vector &v) {
  n = 1;
  linearSum.nullify();
  linearSum.add(v);
  squaredSum = v.magnitudeSq();
#ifdef DEBUG_CFENTRY_1
  _cfentry_conCount++;
#endif
}

CFEntry::~CFEntry() {
#ifdef DEBUG_CFENTRY_1
  _cfentry_desCount++;
#endif
}

void CFEntry::nullify() {
  n = 0;
  linearSum.nullify();
  squaredSum = 0;
}

double CFEntry::distanceSqTo(const Vector &v) const {
  return squaredSum - 2*linearSum.dot(v) + n*v.magnitudeSq();
}

double CFEntry::distanceSqTo(const CFEntry &e) const {
  double res;
  Vector v = linearSum;
  v.scale(1.0/(n*n));
  Vector w = e.linearSum;
  w.scale(1.0/(e.n*e.n));
  res = v.distanceSqTo(w);
  if(res<0) {
    cerr << "Error in CFEntry::distanceSqTo(" << endl;
    cerr << *this << ',' << endl;
    cerr << e << ')' << endl;
    cerr << '=' << res << endl;
  }
  return res;
}

double CFEntry::combineCost(const CFEntry &e) const {
  return distanceSqTo(e);
}

void CFEntry::add(const CFEntry &e) {
  n += e.n;
  linearSum.add(e.linearSum);
  squaredSum += e.squaredSum;
}

ostream &operator<<(ostream &out, const CFEntry &e) {
  out << "Entry(" << e.n << ',' << e.linearSum << ',' << e.squaredSum << ')';
  return out;
}

/***** CF Node *****/

CFNode::CFNode(int maxEntry) {
  entry = new CFEntry*[maxEntry];
  memset(entry,0,maxEntry*sizeof(CFEntry*));
  nEntry = 0;
  this->maxEntry = maxEntry;
}

CFNode::CFNode() {
  entry = NULL;
  maxEntry = nEntry = 0;
}

CFNode::~CFNode() {
  if(entry!=NULL) {
    delete entry;
  }
}

void CFNode::recursiveDelete() {
  // intentionally left blank
}

/***** CF Internal Node *****/

CFInternal::CFInternal(int maxChild) : CFNode(maxChild) {
  child = new CFNode*[maxChild];
  memset(child,0,maxEntry*sizeof(CFNode*));
}
```

```
CFInternal::~CFInternal() {
  // intentionally left blank
}

void CFInternal::recursiveDelete() {
  for(int i=0; i<nEntry; i++) {
    child[i]->recursiveDelete();
    delete child[i];
  }
  for(int i=0; i<nEntry; i++)
    delete entry[i];
  delete entry;
}

void CFInternal::print(ostream &out, int depth) {
  for(int i=0; i<nEntry; i++) {
    for(int j=0; j<depth<<1; j++) out << ' ';
    out << *entry[i] << endl;
    child[i]->print(out,depth+1);
  }
}

SplitNode CFInternal::addEntry(CFEntry *e) {
  if(nEntry==0) {
    cerr << "internal node can't have 0 child." << endl;
    abort();
  }

  // find closest CF
  int minIndex = -1;
  double minVal = HUGE_VAL, val;
  for(int i=0; i<nEntry; i++) {
    val = entry[i]->combineCost(*e);
    if(val < minVal) {
      minVal = val;
      minIndex = i;
    }
  }

  SplitNode res = child[minIndex]->addEntry(e);

  if(res.child2==NULL) {
    entry[minIndex]->add(*e);
    return SplitNode(this,NULL,entry[minIndex],NULL);

  } else {
    // delete node that is splited
    delete entry[minIndex];
    entry[minIndex] = res.ent1;
    delete child[minIndex];
    child[minIndex] = res.child1;

    if(nEntry == maxEntry) {
      //      cerr << "must continue breaking up internal node" << endl;
      CFInternal *child1, *child2;
      CFEntry *ent1, *ent2;

      // partition entry[0..maxEntry-1] to 2 internal nodes (child1 and child2)
      child1 = new CFInternal(maxEntry);
      child2 = new CFInternal(maxEntry);
      ent1 = new CFEntry();
      ent2 = new CFEntry();

      // find two seeds
      CFEntry *seed1, *seed2;
      double farthest = -HUGE_VAL;
      double d;
      for(int i=0; i<nEntry; i++) {
        for(int j=i+1; j<nEntry; j++) {
          //      d = entry[i]->distanceSqTo(*entry[j]);
          d = entry[i]->combineCost(*entry[j]);
          if(d>farthest) {
            farthest = d;
            seed1 = entry[i];
            seed2 = entry[j];
          }
        }
        //      d = entry[i]->distanceSqTo(*res.ent2);
        d = entry[i]->combineCost(*res.ent2);
        if(d>farthest) {
          farthest = d;
          seed1 = entry[i];
          seed2 = res.ent2;
        }
      }

      // split entry[] to child1 or child2 depend on which one is closest
      double d2;
      for(int i=0; i<nEntry; i++) {
        d = seed1->combineCost(*entry[i]);
        d2 = seed2->combineCost(*entry[i]);
        if(d<d2) { // seed 1 is closer
```

```
        child1->child[child1->nEntry] = child[i];
        child1->entry[child1->nEntry++] = entry[i];
        ent1->add(*entry[i]);
      } else { // seed 2 is closer
        child2->child[child2->nEntry] = child[i];
        child2->entry[child2->nEntry++] = entry[i];
        ent2->add(*entry[i]);
      }
    }

    d = seed1->combineCost(*res.ent2);
    d2 = seed2->combineCost(*res.ent2);
    // XXX
    if(child1->nEntry == 0) d=-1;
    if(child2->nEntry == 0) d2=-1;

    if(d<d2) { // seed 1 is closer
      child1->child[child1->nEntry] = res.child2;
      child1->entry[child1->nEntry++] = res.ent2;
      ent1->add(*res.ent2);
    } else { // seed 2 is closer
      child2->child[child2->nEntry] = res.child2;
      child2->entry[child2->nEntry++] = res.ent2;
      ent2->add(*res.ent2);
    }

    return SplitNode(child1,child2,ent1,ent2);

  } else {
    entry[nEntry] = res.ent2;
    child[nEntry] = res.child2;
    nEntry++;
    return SplitNode(this,NULL,entry[minIndex],NULL);
  }
 } // end spliting in child node
}

/***** CF Leaf Node *****/

CFLeaf::CFLeaf(int maxEntry, double t, CFLeaf *inFront) : CFNode(maxEntry) {
 prev = inFront;
 next = inFront->next;
 inFront->next = this;
 next->prev = this;
 threshold = t;
}

CFLeaf::CFLeaf() {
 prev = this;
 next = this;
 threshold = 0;
}

CFLeaf::~CFLeaf() {
 // intentionally blank
}

void CFLeaf::recursiveDelete() {
 for(int i=0; i<nEntry; i++)
  delete entry[i];
 delete entry;
}

void CFLeaf::print(ostream &out, int depth) {
 for(int i=0; i<nEntry; i++) {
  for(int j=0; j<depth<<1; j++) out << ' ';
  out << *entry[i] << endl;
 }
}

SplitNode CFLeaf::addEntry(CFEntry *e) {

 // find closest CF
 int minIndex = -1;
 double minVal = HUGE_VAL, val;
 for(int i=0; i<nEntry; i++) {
  val = entry[i]->combineCost(*e);
  if(val < minVal) {
   minVal = val;
   minIndex = i;
  }
 }

 // check if the closest CF can absorb
 if(minVal < threshold) {
  entry[minIndex]->add(*e);
  return SplitNode(this,NULL,entry[minIndex],NULL);

 } else {
  if(nEntry == maxEntry) {
   CFLeaf *child1, *child2;
   CFEntry *ent1, *ent2;
```

```
      // partition entry[0..maxEntry-1] to 2 leaf nodes (child1 and child2
      child1 = new CFLeaf(maxEntry,threshold,prev);
      prev->next = next;
      next->prev = prev;
      child2 = new CFLeaf(maxEntry,threshold,child1);
      ent1 = new CFEntry();
      ent2 = new CFEntry();

      // find two seeds
      CFEntry *seed1, *seed2;
      double farthest = -HUGE_VAL;
      double d;
      for(int i=0; i<nEntry; i++) {
        for(int j=i+1; j<nEntry; j++) {
          //     d = entry[i]->distanceSqTo(*entry[j]);
          d = entry[i]->combineCost(*entry[j]);
          if(d>farthest) {
            farthest = d;
            seed1 = entry[i];
            seed2 = entry[j];
          }
        }
        //     d = entry[i]->distanceSqTo(*e);
        d = entry[i]->combineCost(*e);
        if(d>farthest) {
          farthest = d;
          seed1 = entry[i];
          seed2 = e;
        }
      }

      // split entry[] to child1 or child2 depend on which one is closest
      double d2;
      for(int i=0; i<nEntry; i++) {
        d = seed1->combineCost(*entry[i]);
        d2 = seed2->combineCost(*entry[i]);
        if(d<d2) {
          child1->entry[child1->nEntry++] = entry[i];
          ent1->add(*entry[i]);
        } else {
          child2->entry[child2->nEntry++] = entry[i];
          ent2->add(*entry[i]);
        }
      }

      d = seed1->combineCost(*e);
      d2 = seed2->combineCost(*e);
      if(child1->nEntry == 0) d=-1;
      if(child2->nEntry == 0) d2=-1;

      if(d<d2) {
        child1->entry[child1->nEntry++] = e;
        ent1->add(*e);
      } else {
        child2->entry[child2->nEntry++] = e;
        ent2->add(*e);
      }
      return SplitNode(child1,child2,ent1,ent2);

    } else {
      entry[nEntry++] = e;
      return SplitNode(this,NULL,entry[nEntry-1],NULL);
    }
  }
}

/***************************** CF Tree *********************************/

CFTree::CFTree(int maxInternalChild,int maxLeafEntry, double leafThreshold) {
  this->maxInternalChild = maxInternalChild;
  this->maxLeafEntry = maxLeafEntry;
  this->leafThreshold = leafThreshold;
  leafList = new CFLeaf();
  root = new CFLeaf(maxLeafEntry,leafThreshold,leafList);
  nPoint = 0;
}

CFTree::~CFTree() {
  root->recursiveDelete();
  delete root;
  delete leafList;
}

void CFTree::addPoint(Vector v) {
  CFEntry *ent = new CFEntry(v);
  nPoint++;
  SplitNode res = root->addEntry(ent);
  if(res.child2!=NULL) {
    delete root;
    root = new CFInternal(maxInternalChild);
    ((CFInternal*)root)->child[0] = res.child1;
```

```
    root->entry[0] = res.ent1;
    ((CFInternal*)root)->child[1] = res.child2;
    root->entry[1] = res.ent2;
    root->nEntry = 2;
  }
}

ostream &operator<<(ostream &out, const CFTree& cft) {
  cft.root->print(out,0);
  return out;
}

int CFTree::numLeaves() {
  int n=0;
  CFLeaf *leaf = leafList->next;
  while(leaf!=leafList) {
    n+=leaf->nEntry;
    leaf = leaf->next;
  }
  return n;
}

VectorSet *CFTree::getLeavesCenters() {
  VectorSet *res = new VectorSet(numLeaves(),
                                 root->entry[0]->linearSum.dimension());
  CFLeaf *leaf = leafList->next;
  Vector v(root->entry[0]->linearSum.dimension());
  while(leaf!=leafList) {
    for(int i=0; i < leaf->nEntry; i++) {
      v.nullify();
      v.add(leaf->entry[i]->linearSum);
      v.scale(1./leaf->entry[i]->n);
      res->append(v);
    }
    leaf = leaf->next;
  }
  return res;
}
```

Iteration Layer

Cyclic.hh

```
#ifndef _CYCLIC_H
#define _CYCLIC_H

#include <iostream>
#include "global.hh"
#include "Vector.hh"
#include "Cluster.hh"
#include "Path.hh"
#include "PathQueue.hh"

class Cluster;
class Cyclic;
class Node;

class Solution {
private:
  int n;
  int *result;
  double kMeanCost;
public:
  Solution(int n);
  ~Solution();
  void record(VectorSet *data, double cost); // record data[].cluster's into result
  int get(int i) const; // get the result for data[i]
  double cost() const; // return k-means cost of this solution
  bool isBetter(Cyclic *cylic); // if solution in Cyclic is better, record it
};

class Cyclic {
private:
  VectorSet *data;
  int numData, numCluster, effN;
  Cluster **cluster;
  Node **dummy;
  Node **node;
  Node *miniClusterSentinel;

  // ** Cost **
  //
```

```
  // x' : grand mean
  // TSS = sum,k:sum,i:( (x_i-x')^2 )   -> constant
  //     = kMeanCost + sum,k:(n_k*m_k^2) - n*x'^2
  //
  // kMeanCost = n*x'^2 + TSS -  sum,k:(n_k*m_k^2)
  //           --constant--
  //
  // want to minimize kMeanCost
  //  => maximize weighted sum of squared cluster means
  //
  // costOffset = n*x'^2 + TSS
  // co = sum,k:(n_k*m_k^2)   ->  want to ***MAXIMIZE CO***
  //    = sum,k:( (1/n_k) * totalSum^2 )
  // kMeanCost i.e. cost = costOffset - co
  //
  double co, costOffset;

  // ** Arc Cost **
  // an arc (n1,n2) has a cost equals to
  //   the cost difference in inserting n1 into n2.cluster
  //   and eject n2 from n2.cluster
  // this cost difference is in term of delta_co
  //   i.e. change in weighted squared cluster mean of n2.cluster
  // negative difference in cost is good for cycle finding,
  //   but we're maximizing cost
  //  => arcCost = oldCost - newCost
  //
  // dummy node represents "no node"
  //
  // cost of applying a cycle:
  //   newCost = oldCost - cycleCost
  //
  double arcCost(Node *n1, Node *n2) const;

  void computeCost();

public:
  Cyclic(VectorSet *data, int numCluster);
  ~Cyclic();
  double cost() const; // return k-means cost of current solution
  int where(int i) const; // return cluster that data[i] is in for current solution
  void checkRep(); // check rep invariant
  int getEffN() const { return effN; }

  /**
   * initialize the solution by
   * picking random clusters centers from k data points
   * then assign all data to the closest center
   * (like doing 1 iteration of Lloyd's)
   *
   * MUST DO THIS BEFORE CALLING lloyd(t) or cyclicExchange()
   */
  void randomInit();

  bool initByCenters(VectorSet *);
  VectorSet *getCenters() const;

  /**
   * perform iterations of lloyd.
   * if iterations==0 -> repeat until improvement <= stopThreshold
   * return the k-means-cost change
   * iterations is changed to the actual iterations performed
   */
  double lloyd(int &iterations);

  /**
   * perform 1 iteration of cyclic exchange
   * return the k-means-cost change
   */
  double cyclicExchange();


  /**
   * preclustering
   */
  void preCluster();
  void cfPreCluster();

  /**
   * write out solution in the specified format in Document/output.format
   */
  void writeSolution(ostream&, int format);

};

inline int randomRange(int lowest_number, int highest_number_p1);

#endif
```

Cyclic.cc

```
#include "Cyclic.hh"
#include <cstdlib>
#include <ctime>
#include <cassert>
#include <cmath>
#include <iomanip>
#include <fstream>
#include "CFTree.hh"

/********** SOLUTION **********/

Solution::Solution(int n) {
  this->n = n;
  result = new int[n];
  memset(result,0xFF,n*sizeof(int));
  kMeanCost = HUGE_VAL;
}

Solution::~Solution() {
  delete result;
}

void Solution::record(VectorSet *data, double cost) {
  for(int i=0; i<n; i++)
    result[i] = data->get(i)->cluster;
  kMeanCost = cost;
}

int Solution::get(int i) const {
  return result[i];
}

double Solution::cost() const {
  return kMeanCost;
}

bool Solution::isBetter(Cyclic *cyclic) {
  if( cyclic->cost() < kMeanCost ) {
#ifdef DEBUG_COST
    cyclic->checkRep();
#endif
    for(int i=0; i<n; i++)
      result[i] = cyclic->where(i);
    kMeanCost = cyclic->cost();
    return true;
  } else {
    return false;
  }
}

/********** CYCLIC **********/

/***** Constructor / Destructor *****/

Cyclic::Cyclic(VectorSet *data, int numCluster) {
  this->data = data;
  numData = data->size();
  effN = numData;
  this->numCluster = numCluster;

  // init clusters and dummy nodes
  cluster = new (Cluster*)[numCluster];
  dummy = new (Node*)[numCluster];
  for(int k=0; k<numCluster; k++) {
    cluster[k] = new Cluster(k,data->dimension());
    dummy[k] = new Node(k);
  }

  // init nodes
  node = new (Node*)[numData];
  for(int i=0; i<numData; i++)
    node[i] = new Node(data->get(i));

  // compute cost offset
  // costOffset = TotalSumOfSquare + n*grandMeanSq
  Vector grandMean = data->mean();
  // costOffset = n*x'^2 + TSS
  costOffset = numData * grandMean.magnitudeSq();
  for(int i=0; i<numData; i++)
    costOffset += grandMean.distanceSqTo(*data->get(i));
  co = -HUGE_VAL; // k-mean-cost = inf

  miniClusterSentinel = new Node(-1);
}

Cyclic::~Cyclic() {
  for(int k=0; k<numCluster; k++) {
    delete cluster[k];
```

```
    delete dummy[k];
  }
  delete cluster;
  delete dummy;

  delete miniClusterSentinel;
}

/***** Query *****/

double Cyclic::cost() const { return costOffset - co; }

int Cyclic::where(int i) const { return data->get(i)->cluster; }

/***** Algorithm *****/

/*** Search Graph Definition ***/
double Cyclic::arcCost(Node *n1, Node *n2) const {
  assert(n1->cluster != n2->cluster);

  int k = n2->cluster; // cluster of interest
  Vector ts = cluster[k]->totalSum();
  int oldSize = cluster[k]->size();
  double old = ts.magnitudeSq() / oldSize;

  if(n1->data!=NULL) { // insert n1
    ts.add(*n1->tsMini);
    oldSize += n1->countMini;
  }
  if(n2->data!=NULL) { // eject n2
    ts.subtract(*n2->tsMini);
    oldSize -= n2->countMini;
  }
  if(oldSize<=0) return HUGE_VAL;
  else return old - ( ts.magnitudeSq() / oldSize );

}

/*** Helper ***/

void Cyclic::computeCost() {
  co = 0.;
  for(int k=0; k<numCluster; k++) {
    if(cluster[k]->size()<=0) {
      co = -HUGE_VAL;
      return;
    }
    co += cluster[k]->totalSum().magnitudeSq() / cluster[k]->size();
  }
}

inline int randomRange(int lowest_number, int highest_number_p1) {
  // return random number in range [lowest_number, highest_number_p1-1]
  return lowest_number +
    int( (highest_number_p1 - lowest_number) * (rand()/(RAND_MAX + 1.0)) );
}

/*** Randomize ***/
void Cyclic::randomInit() {
  // remove all data from clusters
  for(int k=0; k<numCluster; k++) {
    cluster[k]->initIterator();
    while(cluster[k]->hasNext()) {
      cluster[k]->next();
      cluster[k]->removeIt();
    }
   }

  // choose centers randomly
  Vector **center = new (Vector*)[numCluster]; // ref
  int randomInterval = numData / numCluster;
  for(int k=0; k<numCluster-1; k++) {
    center[k] = data->get(randomRange(0,numData));
  }
  center[numCluster-1] = data->get(randomRange(0,numData));

  // assign all data to the closest center
  double minDist,dist;
  int minIndex;

  for(int i=0; i<numData; i++) {
    // if node[i] is already in a minicluster, don't add again
    if(node[i]->nextMini == miniClusterSentinel) continue;

    minIndex = -1;
    minDist = HUGE_VAL;
    // v is mean of minicluster
    Vector v(*node[i]->tsMini);
    v.scale(1./node[i]->countMini);
    for(int k=0; k<numCluster; k++) {
      dist = center[k]->distanceSqTo(v);
      if( dist<minDist) {
```

```
        minDist = dist;
        minIndex = k;
      }
    }
    assert(minIndex>=0);
    cluster[minIndex]->add(node[i]);
  }

  // center can be delete. just array of pointers to existing objects
  delete center;

  computeCost();
}

bool Cyclic::initByCenters(VectorSet *centers) {
  assert((centers->size()==numCluster) &&
         (centers->dimension()==data->dimension()));
  // remove all data from clusters
  for(int k=0; k<numCluster; k++) {
    cluster[k]->initIterator();
    while(cluster[k]->hasNext()) {
      cluster[k]->next();
      cluster[k]->removeIt();
    }
  }

  // assign all data to the closest center
  double minDist,dist;
  int minIndex;
  Vector *v;
  for(int i=0; i<numData; i++) {
    minIndex = -1;
    minDist = HUGE_VAL;
    v = new Vector(*node[i]->tsMini);
    v->scale(1./node[i]->countMini);
    for(int k=0; k<numCluster; k++) {
      dist = centers->get(k)->distanceSqTo(*v);
      if( dist<minDist) {
        minDist = dist;
        minIndex = k;
      }
    }
    cluster[minIndex]->add(node[i]);
    delete v;
  }

  computeCost();

  return true;
}

VectorSet *Cyclic::getCenters() const {
  VectorSet *centers = new VectorSet(numCluster,data->dimension());
  for(int k=0; k<numCluster; k++) {
    centers->append(cluster[k]->mean());
  }
  centers->lexSort();
  return centers;
}

/*************** Lloyd's *****************/
double Cyclic::lloyd(int &iterations) {
  double costBefore = cost();
  if(iterations==0) iterations=INT_MAX;

  double old1 = HUGE_VAL;
  double old2 = HUGE_VAL;
  double old3;

  for(int iter=0; iter<iterations; iter++) {
    old3 = old2;
    old2 = old1;
    old1 = cost();

    Node *n;
    Vector *nCenter;
    double minDist,dist;
    int minIndex;

    for(int k=0; k<numCluster; k++) {
      cluster[k]->initIterator();
      while(cluster[k]->hasNext()) {
        n = cluster[k]->next();
        if(n->cluster >= 0) { // not yet considered (change a data point only once)
          nCenter = new Vector(*n->tsMini);
          nCenter->scale(1./n->countMini);
          // find closest center
          minIndex = -1;
          minDist = HUGE_VAL;
          for(int j=0; j<numCluster; j++) {
            Vector c = cluster[j]->totalSum(); // <---mean of cluster[j]
            c.scale(1./cluster[j]->size()); // <-------|
```

```
        dist = c.distanceSqTo(*nCenter);
        if( dist<minDist) {
          minDist = dist;
          minIndex = j;
        }
      }
      if(minIndex!=k) {
        // **** MOVE **** //
        cluster[k]->removeIt();
        cluster[minIndex]->add(n);
      }
      n->cluster -= numCluster; // mark as considered
      delete nCenter;
    }
  }
}

// undo marking
for(int k=0; k<numCluster; k++) {
  cluster[k]->initIterator();
  while(cluster[k]->hasNext()) {
    n = cluster[k]->next();
    assert(n->cluster<0);
    n->cluster += numCluster;
  }
}

computeCost();
cout << cost()-old1 << ' ';

// if not improving so much, terminate. i.e. terminate when
// - deplete some clusters (cost blows up)
// - reach local optimum (cost doesn't change)
// - decrease in cost over 3 stages is less than STOP_THRESHOLD_RATIO3
if( (isinf(co)) ||
    (cost()-old1 == 0.) ||
    (cost()-old3 > -old3*STOP_THRESHOLD_RATIO3 )
    ){
  iterations = iter+1;
  break;
}

} // iter loop

return cost()-costBefore;
}

/*************** Cyclic Exchange ***************/
double Cyclic::cyclicExchange() {
  double costBefore = cost();
  if(isinf(costBefore)) return NAN;

  PathQueue *PK[2];
  PathQueue *pk, *pk1; // ref
  Path *bestCycle;
  Node *ni, *nj, *nh; // ref
  double w; // weight (arc cost)

  PK[0] = new PathQueue(numData,numCluster);
  PK[1] = new PathQueue(numData,numCluster);

  //*** init P1 = { (i,j) : c_ij < 0 } ***
  pk = PK[1];
  for(int i=0; i<numCluster; i++) for(int j=0; j<numCluster; j++) if(i!=j) {
    // there are edge from a point to all points in other clusters

    cluster[i]->initIterator();
    while(cluster[i]->hasNext()) {
      ni = cluster[i]->next();
      cluster[j]->initIterator();
      while(cluster[j]->hasNext()) {
        nj = cluster[j]->next();
        w = arcCost(ni,nj);
        if(w<0) pk->enQueue( new Path(numCluster,ni,nj,w) );
      }
      // nis -> nj'
      w = arcCost(ni,dummy[j]);
      if(w<0) pk->enQueue( new Path(numCluster,ni,dummy[j],w) );
    }

    // ni' -> njs
    cluster[j]->initIterator();
    while(cluster[j]->hasNext()) {
      nj = cluster[j]->next();
      w = arcCost(dummy[i],nj);
      if(w<0) pk->enQueue( new Path(numCluster,dummy[i],nj,w) );
    }

  } // for all possible edges
  //*** end init P1 ***

  Path *p; // ref
```

```
 bestCycle = new Path(numCluster,dummy[0],dummy[1],HUGE_VAL); // sentinel
 //*** loop up the path length ***
 for(int pathLen=1; pathLen<numCluster; pathLen++) {
   pk = PK[pathLen&1];
   pk1 = PK[(pathLen+1)&1];

   while(pk->size() > 0) {
     p = pk->deQueue();
     ni = p->head->node;
     nh = p->tail->node;

     // try to close the cycle
     if( ni->cluster != nh->cluster ) {
       w = arcCost(ni,nh);
       if( w!=HUGE_VAL && p->cost()+w < bestCycle->cost() ) { // find better cycle
         delete bestCycle;
         bestCycle = new Path(*p);
         bestCycle->extend(nh,w);
       }
     }

     // try to extend ni->nj
     for(int j=0; j<numCluster; j++) if(p->noCluster(j)) {
       cluster[j]->initIterator();
       while(cluster[j]->hasNext()) {
         nj = cluster[j]->next();
         w = arcCost(ni,nj);
         if( w!=HUGE_VAL && p->cost()+w < 0) {
           Path *xp = new Path(*p);
           xp->extend(nj,w);
           pk1->enQueue(xp);
         }
       }
     }

     delete p;
   }

   // found good-enough cycle
   if(bestCycle->cost() < -costBefore*CYCLE_OKAY_RATIO) break;

 } // for pathLen
 //*** end loop up the path length ***

 //*** apply the improvement ***
 if( !isinf(bestCycle->cost()) && bestCycle->cost()<0 ) {
   // found a negative-cost cycle
   Path::Element *insert, *eject; // ref
   insert = bestCycle->tail;
   int firstK = insert->node->cluster;
   eject = insert->next;

   // handle the last-first edge
   if(insert->node->data!=NULL) { // if not dummy node, eject it
     insert->node->removeFrom(cluster[insert->node->cluster]);
   }

   // go around the improvement cycle
   int k; // cluster of interest
   while(eject!=bestCycle->head) {
     k = eject->node->cluster;
     if(insert->node->data!=NULL) {
       cluster[k]->add(insert->node);
     }

     if(eject->node->data!=NULL) {
       eject->node->removeFrom(cluster[k]);
     }

     insert = eject;
     eject = insert->next;
   }
   if(insert->node->data!=NULL) {
     cluster[firstK]->add(insert->node);
   }

   co -= bestCycle->cost();
 }
 //*** end apply the improvement ***

 delete PK[0];
 delete PK[1];
 delete bestCycle;

 return cost()-costBefore;
}

/***** Preclutering *****/

// 2-node minicluster only
const double PRECLUSTER_THRESHOLD_PER_DIM = .001;
void Cyclic::preCluster() {
```

```
  double d;
  double threshold = PRECLUSTER_THRESHOLD_PER_DIM * data->dimension();
  for(int i=0; i<numData-1; i++) {
    if(node[i]->nextMini!=NULL) continue;
    for(int j=i+1; j<numData; j++) {
      if(node[j]->nextMini!=NULL) continue;
      d = node[i]->data->distanceSqTo(*node[j]->data);
      if(d < threshold) {
        effN--;
        node[j]->nextMini = miniClusterSentinel;
        node[i]->nextMini = node[j];
        node[i]->countMini += node[j]->countMini;
        node[i]->tsMini->add(*node[j]->tsMini);
        break; // node[i] can't combine with other
      }
    }
  }
}

void Cyclic::cfPreCluster() {
  int B,L;
  double T;

  // calculate parameters
  B = 8;
  L = 10;
  T = .5 * data->dimension();

  // build cf tree
  CFTree *cft = new CFTree(B,L,T);
  for(int i=0; i<data->size(); i++)
    cft->addPoint(*data->get(i));

  // take all leaves to be mini clusters
  numData = 0;
  Node *node;
  CFLeaf *cfleaf = cft->leafList->next;
  while(cfleaf!=cft->leafList) {
    for(int i=0; i<cfleaf->nEntry; i++) {
      node = new Node(data->get(0));
      node->countMini = cfleaf->entry[i]->n;
      node->tsMini->nullify();
      node->tsMini->add(cfleaf->entry[i]->linearSum);
      this->node[numData++] = node;
    }
    cfleaf = cfleaf->next;
  }
  effN = numData;
}

/***** Output *****/
void Cyclic::writeSolution(ostream &out, int format) {
  if(format == 1) {
    //     format 1: cluster oriented
    //       n d k cost
    //       c_1_size c_1_mean
    //       x_1
    //       ...
    //       x_c_1_size
    //       c_2_size c_2_mean
    //       x_2
    //       ...
    //       x_c_2_size
    //       ...
    //       ...
    //       c_k_size c_k_mean
    //       x_1
    //       ...
    //       x_c_k_size

    out << setprecision(F_PRECISION) << scientific;

    out << numData << " "
        << data->dimension() << " "
        << numCluster << " "
        << cost() << endl;
    for(int k=0; k<numCluster; k++) {
      Vector mean = cluster[k]->totalSum();
      mean.scale(1./cluster[k]->size());
      out << cluster[k]->size() << " "
          << mean << endl;
      cluster[k]->initIterator();
      while(cluster[k]->hasNext())
        out << *cluster[k]->next() << endl;
    }

  }

  else if(format == 2) {
    //     format 2: data oriented
    //               (data are sorted in the same order as in input file)
    //       n d k cost
```

```
    //      c_1
    //      c_2
    //      ...
    //      c_n

    out << setprecision(F_PRECISION) << scientific;

    out << numData << " "
        << data->dimension() << " "
        << numCluster << " "
        << cost() << endl;

    for(int i=0; i<numData; i++) {
      out << where(i) << endl;
    }

  }
}

/***** Representation Invariant *****/
void Cyclic::checkRep() {
  int countNode = 0;
  Node *v;
  Vector ts(data->dimension());
  Vector *cMean;
  double countCost = 0.;

  for(int k=0; k<numCluster; k++) {
    assert(dummy[k]->cluster==k);
    ts.nullify();
    cMean = new Vector(cluster[k]->totalSum());
    cMean->scale(1./cluster[k]->size());
    cluster[k]->initIterator();
    int countClusterNode = 0;

    while(cluster[k]->hasNext()) {
      v = cluster[k]->next();
      if(v==NULL) break;
      assert(v->cluster==k);
      assert(v->data->cluster==k);
      countNode += v->countMini;
      countClusterNode += v->countMini;
      ts.add(*v->tsMini);
      while(v!=NULL && v!=miniClusterSentinel) {
        countCost += cMean->distanceSqTo(*v->data);
        v = v->nextMini;
      }
    }

    assert(countClusterNode > 0);
    assert(countClusterNode == cluster[k]->size());

    ts.scale(1./countClusterNode);
    if(ts.distanceSqTo(*cMean) > FP_TOL*ts.magnitudeSq()) {
      cerr << "cluster mean differences " << setprecision(20)
           << scientific << ts.distanceSqTo(*cMean) << endl;
      cerr << "freshly calculated = " << ts << endl;
      cerr << "stored = " << *cMean << endl;
      assert(ts.distanceSqTo(*cMean) < FP_TOL*ts.magnitudeSq());
    }
    delete cMean;
  }
  assert(countNode == numData);
  if(abs(countCost-cost()) >= FP_TOL*countCost) {
    cerr << "cost difference = " << setprecision(20)
         << scientific << abs(countCost-cost()) << endl;
    cerr << "freshly calculated = " << countCost << endl;
    cerr << "stored = " << cost() << endl;
    assert(abs(countCost-cost()) < FP_TOL*countCost);
  }
}
```

Experiment Layer

Standalone and Precluster

main.cc

```
#include "testdriver.hh"
#include "global.hh"
```

```
#include "Cyclic.hh"
#include <iostream>
#include <cassert>
#include <string>
using namespace std;

typedef enum {NONE,LLOYD,CYCLIC_START,CYCLIC_CONTINUE,HYBRID} AlgType;
typedef enum {PURE,TWONODES,BIRCH} PreType;

int numCluster = 0;
int numInit = 0;
VectorSet *data;
Cyclic *cyclic;
Solution *bestSol;

void runLloyd() {
  int lloydIter;
  for(int initi=0; initi<numInit; initi++) {
    cout << initi << ' ';
    cyclic->randomInit();
    cout << cyclic->cost() << ' ';
    lloydIter = 0; // let run until local optimum
    cyclic->lloyd(lloydIter);

    bestSol->isBetter(cyclic);
    cout << cyclic->cost() << ' ' << lloydIter << " 0" << endl;
  }
}

void runCyclicStart() {
  int lloydIter,cycIter;
  double change;
  double old1,old2,old3;
  for(int initi=0; initi<numInit; initi++) {
    cout << initi << ' ';
    cyclic->randomInit();
    cout << cyclic->cost() << ' ';
    lloydIter = 0;
    // do not run lloyd before cyclic->lloyd(lloydIter);

    old1 = HUGE_VAL;
    old2 = HUGE_VAL;
    cycIter = 0;
    while(true) {
      old3 = old2;
      old2 = old1;
      old1 = cyclic->cost();

      cycIter++;
      change = cyclic->cyclicExchange();
      cout << change << ' ';
      if( (isinf(change)) || (isnan(change)) ||
          (change==0.) ||
          (cyclic->cost()-old3 > -old3*STOP_THRESHOLD_RATIO3)
          ) break;
    }

    bestSol->isBetter(cyclic);
    cout << cyclic->cost() << ' ' << lloydIter << ' ' << cycIter << endl;
  }
}

void runCyclicContinue() {
  int lloydIter,cycIter;
  double change;
  double old1,old2,old3;
  for(int initi=0; initi<numInit; initi++) {
    cout << initi << ' ';
    cyclic->randomInit();
    cout << cyclic->cost() << ' ';
    lloydIter = 0;
    cyclic->lloyd(lloydIter);

    // now lloyd has reached local optimum.
    // try to use cyclic exchange to produce better result
    old1 = HUGE_VAL;
    old2 = HUGE_VAL;
    cycIter = 0;
    while(true) {
      old3 = old2;
      old2 = old1;
      old1 = cyclic->cost();

      cycIter++;
      change = cyclic->cyclicExchange();
      cout << change << ' ';
      if( (isinf(change)) || (isnan(change)) ||
          (change==0.) ||
          (cyclic->cost()-old3 > -old3*STOP_THRESHOLD_RATIO3)
          ) break;
    }
```

```
    bestSol->isBetter(cyclic);
    cout << cyclic->cost() << ' ' << lloydIter << ' ' << cycIter << endl;
  }
}

void runHybrid() {
  int lloydIter,cycIter,iter;
  double change;
  double old1,old2,old3;
  for(int initi=0; initi<numInit; initi++) {
    cout << initi << ' ';
    cyclic->randomInit();
    cout << cyclic->cost() << ' ';
    lloydIter = 0;

    old1 = HUGE_VAL;
    old2 = HUGE_VAL;
    cycIter = 0;
    while(true) {
      old3 = old2;
      old2 = old1;
      old1 = cyclic->cost();

      iter = HYBRID_NUM_LLOYD;
      lloydIter += iter;
      cyclic->lloyd(iter);

      cycIter++;
      change = cyclic->cyclicExchange();
      cout << change << ' ';

      if( (isinf(change)) || (isnan(change)) ||
          (change==0.) ||
          (cyclic->cost()-old3 > -old3*STOP_THRESHOLD_RATIO3)
          ) break;
    }

    bestSol->isBetter(cyclic);
    cout << cyclic->cost() << ' ' << lloydIter << ' ' << cycIter << endl;
  }
}

bool notNumber(const char* s) {
  for(int i=0; i<strlen(s); i++) {
    if(!isdigit(s[i])) return true;
  }
  return false;
}

void commandLineHelp() {
  clog << "command line:" << endl
       << "% CylicExchange {lloyd,cyclic,continue,hybrid} "
       << "numCluster numInit [-r randomseed] [-p {pure,two,birch}] [-z]" << endl;
  abort();
}

//////////////////////////////////////////////////////////////////////
/**
 * command line:
 * % CylicExchange {lloyd,cyclic,continue,hybrid} numCluster numInit [-r randomseed] [-p {pure,two,birch}] [-z]
 * if randomseed is not given, seed from system time
 * -z : z-score normalization
 *
 * takes input from stdin
 */

int main(int argc, char *argv[]) {

  /*** parse arguments ***/
  if(argc<4) commandLineHelp();

  // default values
  AlgType alg = NONE;
  unsigned randomSeed = static_cast<unsigned>(time(0));;
  PreType pre = PURE;
  bool zScore = false;

  // parse
  try {
    if(strcmp(argv[1],"lloyd")==0) alg = LLOYD;
    else if(strcmp(argv[1],"cyclic")==0) alg = CYCLIC_START;
    else if(strcmp(argv[1],"continue")==0) alg = CYCLIC_CONTINUE;
    else if(strcmp(argv[1],"hybrid")==0) alg = HYBRID;
    else commandLineHelp();

    if(notNumber(argv[2])) commandLineHelp();
    numCluster = atoi(argv[2]);
    if(notNumber(argv[3])) commandLineHelp();
    numInit = atoi(argv[3]);

    if(argc>4) {
      int i = 4;
```

```
    while(i<argc) {
      if(strcmp(argv[i],"-r")==0) {
        i++;
        if(notNumber(argv[i])) commandLineHelp();
        randomSeed = atoi(argv[i]);
        i++;
      }
      else if(strcmp(argv[i],"-p")==0) {
        i++;
        if(strcmp(argv[i],"two")==0) pre = TWONODES;
        else if(strcmp(argv[i],"birch")==0) pre = BIRCH;
        else if(strcmp(argv[i],"pure")==0) pre = PURE;
        else commandLineHelp();
        i++;
      }
      else if(strcmp(argv[i],"-z")==0) {
        zScore = true;
        i++;
      }
      else commandLineHelp();
    }
  }

} catch (int error) {
  commandLineHelp();
}

/*** get input data from stdin ***/
data = VectorSet::readDenseMatrix(cin,zScore);
cyclic = new Cyclic(data, numCluster);
bestSol = new Solution(data->size());

//  initTestDriver();
srand(randomSeed);

/*** run the algorithm ***/
startTimer();

switch(pre) {
case PURE: break;
case TWONODES : cyclic->preCluster(); break;
case BIRCH: cyclic->cfPreCluster(); break;
}

cout << numInit << ' ' << cyclic->getEffN() << endl;
cout << setprecision(F_PRECISION) << scientific;

switch(alg) {
case LLOYD : runLloyd(); break;
case CYCLIC_START: runCyclicStart(); break;
case CYCLIC_CONTINUE: runCyclicContinue(); break;
case HYBRID: runHybrid(); break;
}

cout << bestSol->cost() << ' ' // << numInit << ' '
     << setprecision(TIME_PRECISION) << fixed << ellapseTime() << endl;

clog << "Time = " << setprecision(TIME_PRECISION) << fixed << ellapseTime() << endl;

delete bestSol;
delete cyclic;
delete data;
return 0;
}
```

Two-stage Algorithm

A Structure for storing Lloyd's Locals

LloydTree.hh

```
#ifndef _LLOYD_TREE_H
#define _LLOYD_TREE_H

#include <iostream>
#include "global.hh"
#include "Vector.hh"
#include "Cluster.hh"
#include "Cyclic.hh"

class LloydTree {
private:
```

```
  class Node {
  public:
    Node(VectorSet *x, double c);
    VectorSet *data;
    double cost;
    Node *left,*right;
  };
  int n;
  Node *root;
  void infixPrint(ostream &, Node *n,int &i) const;
  void infixCollapse(VectorSet *res, Node *node);
  void infixGetFirst(VectorSet **res, int &count, int total, Node *node);
public:
  LloydTree();
  ~LloydTree();
  int size() const;
  bool insert(VectorSet *x, double c);
  VectorSet *makeCollapsedPoints();
  VectorSet **getFirstCenters(int &um);
  friend ostream &operator<<(ostream &, const LloydTree);
  static double centerDistance(VectorSet *a, VectorSet *b);
};

#endif
```

LloydTree.cc

```
#include "LloydTree.hh"
#include <iomanip>

LloydTree::Node::Node(VectorSet *x, double c) {
  data = x;
  cost = c;
  left = NULL;
  right = NULL;
}

LloydTree::LloydTree() {
  root = NULL;
  n = 0;
}

LloydTree::~LloydTree() {
}

int LloydTree::size() const {
  return n;
}

bool LloydTree::insert(VectorSet *x, double c) {
  if(n==0) {
    root = new Node(x,c);
    n = 1;
    return true;
  } else {
    double thres = c*FP_TOL;
    Node *p;
    Node *a = root;
    while(a!=NULL) {
      p = a;
      //      if(abs(c-a->cost)<thres) return false;
      if(centerDistance(x,a->data)<thres) return false;
      if(c>a->cost) a = a->right;
      else a = a->left;
    }
    a = new Node(x,c);
    if(c>p->cost) p->right = a;
    else p->left = a;
    n++;
    return true;
  }
}

void LloydTree::infixPrint(ostream &out, Node *n, int &i) const {
  if(n!=NULL) {
    infixPrint(out,n->left,i);
    out << fixed << i++ << ' ' << setprecision(F_PRECISION)
        << scientific << n->cost << ':' << *n->data << endl;
    infixPrint(out,n->right,i);
  }
}

ostream &operator<<(ostream &out, const LloydTree tree) {
  int i=0;
  tree.infixPrint(out,tree.root,i);
}
```

```
void LloydTree::infixCollapse(VectorSet *res, Node *node) {
  if(node!=NULL) {
    infixCollapse(res,node->left);
    res->append(node->data->collapse());
    infixCollapse(res,node->right);
  }
}

VectorSet *LloydTree::makeCollapsedPoints() {
  if(n==0) return NULL;
  VectorSet *res = new VectorSet(n,root->data->dimension()*root->data->size());
  infixCollapse(res,root);
  return res;
}

void LloydTree::infixGetFirst(VectorSet **res, int &count, int total, Node *node) {
  if(node!=NULL) {
    infixGetFirst(res,count,total,node->left);
    if(count<total) {
      res[count++] = node->data;
      if(count<total) infixGetFirst(res,count,total,node->right);
    }
  }
}

VectorSet **LloydTree::getFirstCenters(int &num) {
  VectorSet **res = new (VectorSet*)[num];
  int count = 0;
  infixGetFirst(res,count,num,root);
  num = count;
  return res;
}

double LloydTree::centerDistance(VectorSet *a, VectorSet *b) {
  int match[b->size()];
  for(int i=0; i<b->size(); i++) match[i] = -1;

  for(int i=0; i<a->size(); i++) {
    int minj,j;
    double minDist,dist;
    minj = -1;
    minDist = HUGE_VAL;
    for(j=0; j<b->size(); j++) {
      if(match[j]==-1) {
        dist = a->get(i)->distanceSqTo(*b->get(j));
        if(dist<minDist) {
          minDist = dist;
          minj = j;
        }
      }
    }
    assert((minj>=0) && (minj<b->size()));
    match[minj] = i;
  }

  double res = 0;
  for(int j=0; j<b->size(); j++) {
    res += b->get(j)->distanceSqTo(*a->get(match[j]));
  }
  return res;
}
```

Running Two-stage Algorithm

selective.cc

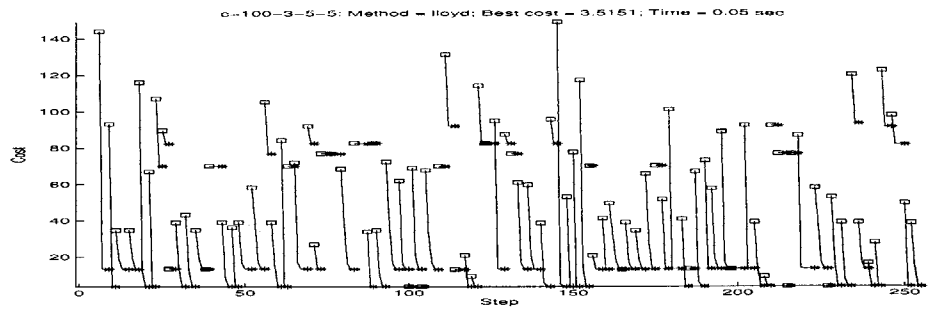
```
#include "testdriver.hh"
#include "global.hh"
#include "Cyclic.hh"
#include "LloydHeap.hh"
#include "LloydTree.hh"
#include <iostream>
#include <cassert>
#include <string>
#include <fstream>
#include "CFTree.hh"
using namespace std;

#define PERCENT_RUN_CYCLIC_LOWCOST .1
#define PERCENT_RUN_CYCLIC_SEPARATED .2

int numCluster = 0;
int numInitLloyd = 0;
VectorSet *data;
Cyclic *cyclic;
```

```
LloydTree *tree;
Solution *bestSol;

bool notNumber(const char* s) {
  for(int i=0; i<strlen(s); i++) {
    if(!isdigit(s[i])) return true;
  }
  return false;
}

void commandLineHelp() {
  cerr << "command line:" << endl
       << "% SelectCyclic numCluster numInitLloyd [-r randomseed] [-z]" << endl;
  abort();
}

void removeTooSimilar(VectorSet **cset, int k, int currentSize, int expectedSize) {
  double dist[currentSize][currentSize];
  for(int i=0; i<currentSize; i++) {
    dist[i][i]=0;
    for(int j=i+1; j<currentSize; j++) {
      dist[i][j] = LloydTree::centerDistance(cset[i],cset[j]);
      dist[j][i] = dist[i][j];
    }
  }
}

////////////////////////////////////////////////////////////////
/**
 * command line:
 * % SelectCyclic numCluster numInitLloyd [-r randomseed] [-z]
 *         0          1          2          3
 * if randomseed is not given, seed from system time
 * -z : z-score normalization
 *
 * takes input from stdin
 */

int main(int argc, char *argv[]) {

  /*** parse arguments ***/
  if(argc<3) commandLineHelp();

  // default values
  unsigned randomSeed = static_cast<unsigned>(time(0));;
  bool zScore = false;

  // parse
  try {
    if(notNumber(argv[1])) commandLineHelp();
    numCluster = atoi(argv[1]);
    if(notNumber(argv[2])) commandLineHelp();
    numInitLloyd = atoi(argv[2]);

    if(argc>3) {
      int i = 3;
      while(i<argc) {
        if(strcmp(argv[i],"-r")==0) {
          i++;
          if(notNumber(argv[i])) commandLineHelp();
          randomSeed = atoi(argv[i]);
          i++;
        }
        else if(strcmp(argv[i],"-z")==0) {
          zScore = true;
          i++;
        }
        else commandLineHelp();
      }
    }

  } catch (int error) {
    commandLineHelp();
  }

  /*** get input data from stdin ***/
  data = VectorSet::readDenseMatrix(cin,zScore);
  cyclic = new Cyclic(data,numCluster);
  //  heap = new LloydHeap(numInitLloyd / 10); //XXX 10% ?
  tree = new LloydTree();
  clog << "Read data" << endl;

  //  initTestDriver();
  srand(randomSeed);
  startTimer();

  /*** run the algorithm ***/

  clog << "Start Lloyd" << endl;
  // run lloyd for "numInitLloyd" iteration, keep distinct solutions
  cout << fixed << numInitLloyd << endl;
  cout << setprecision(F_PRECISION) << scientific;
```

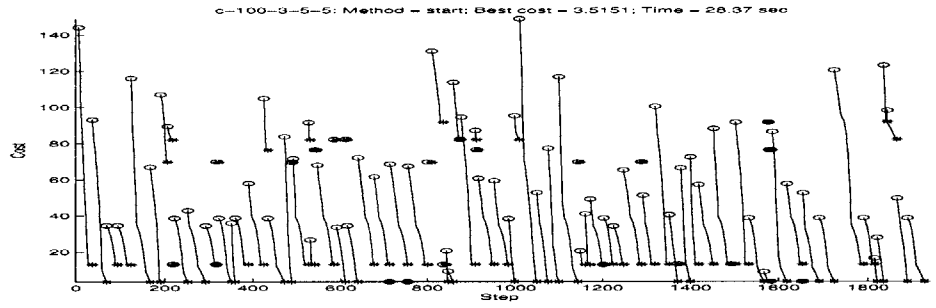
```
  for(int lit=0; lit<numInitLloyd; lit++) {
    cout << lit << ' ';
    cyclic->randomInit();
    cout << cyclic->cost() << ' ';
    int lloydIter = 0; // let run until local optimum
    cyclic->lloyd(lloydIter);
    cout << cyclic->cost() << ' ' << lloydIter << " 0" << endl;

    if(finite(cyclic->cost())) {
      VectorSet *x = cyclic->getCenters();
      if(!tree->insert(x,cyclic->cost())) {
        // clog << "Repeated solution !!!" << endl;
        delete x;
      } else {
      }
    }
  }

  clog << "Total distinct Lloyd's locals = " << tree->size() << endl;

  int numLowCostLocals = (int)round(numInitLloyd*PERCENT_RUN_CYCLIC_LOWCOST);
  VectorSet **cset = tree->getFirstCenters(numLowCostLocals);

  clog << "Consider " << numLowCostLocals << " low cost locals" << endl;

  int numSeparatedLocals = numLowCostLocals;
  removeTooSimilar(cset,numCluster,numLowCostLocals,numSeparatedLocals);

  clog << "Run cyclic exchange on " << numSeparatedLocals
       << " most separated low-cost locals" << endl;

  // among the solutions kept, run cyclic exchange.
  bestSol = new Solution(data->size());
  int initi = 0, cycIter;
  double old1,old2,old3,change;
  clog << "Run Cyclic Exchange on some Lloyd's locals" << endl;

  cout << fixed << numSeparatedLocals << endl;
  cout << setprecision(F_PRECISION) << scientific;

  for(int i=numSeparatedLocals-1; i>=0; i--) {
    cyclic->initByCenters(cset[i]);

    // run cylic exchange
    cout << fixed << initi++ << ' ';
    cout << setprecision(F_PRECISION) << scientific;
    cout << cyclic->cost() << ' ';
    old1 = HUGE_VAL;
    old2 = HUGE_VAL;
    cycIter = 0;
    while(true) {
      old3 = old2;
      old2 = old1;
      old1 = cyclic->cost();

      cycIter++;
      change = cyclic->cyclicExchange();
      cout << change << ' ';
      if( (isinf(change)) || (isnan(change)) ||
          (change==0.) ||
          (cyclic->cost()-old3 > -old3*STOP_THRESHOLD_RATIO3)
          ) break;
    }

    bestSol->isBetter(cyclic);
    cout << cyclic->cost() << ' ' << 0 << ' ' << cycIter << endl;
  }

  cout << setprecision(F_PRECISION) << scientific
       << bestSol->cost() << ' '
       << setprecision(TIME_PRECISION) << fixed
       << ellapseTime() << endl;

  clog << "Time = " << setprecision(TIME_PRECISION)
       << fixed << ellapseTime() << endl;

  delete data;
  delete cyclic;
  delete bestSol;
  return 0;
}
```

Appendix C

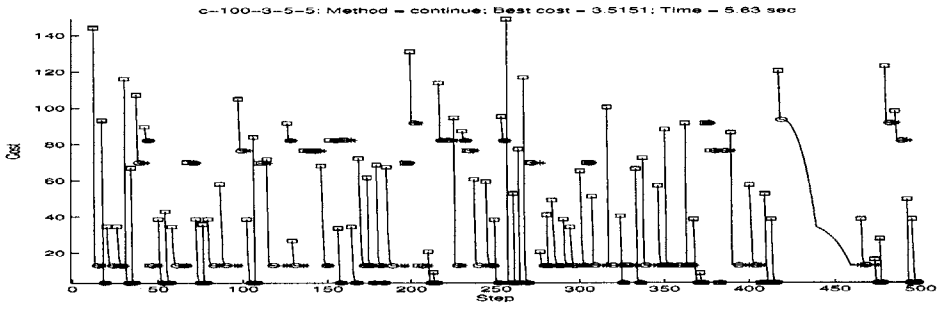
Dynamics During Iterations of Lloyd's Algorithm and Two Versions of Cyclic Exchanges



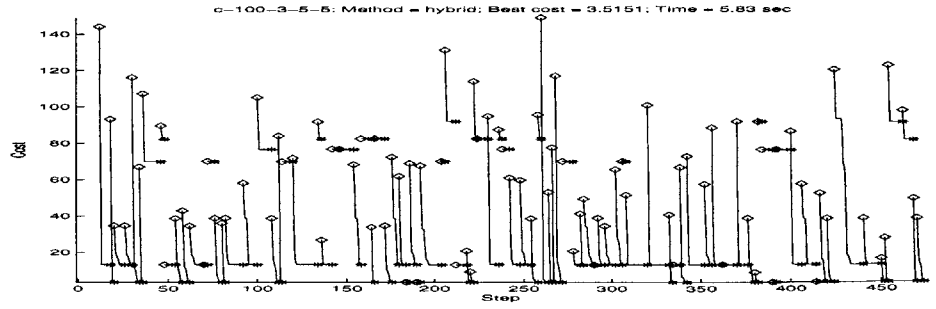
(a)



(b)

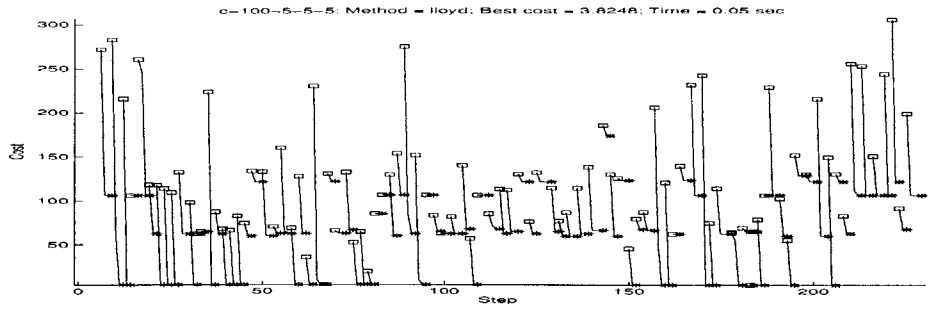


(c)

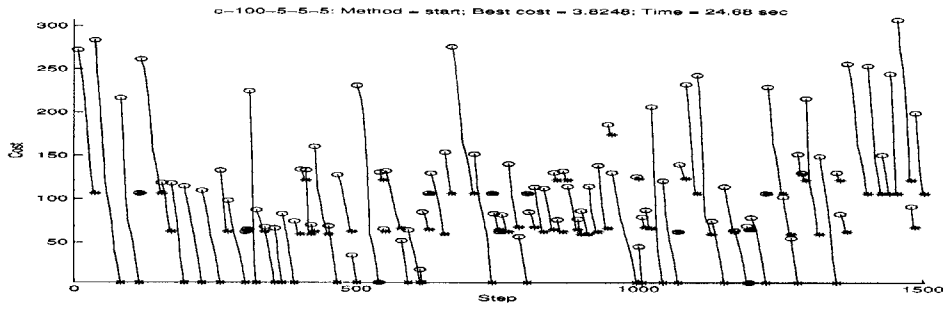


(d)

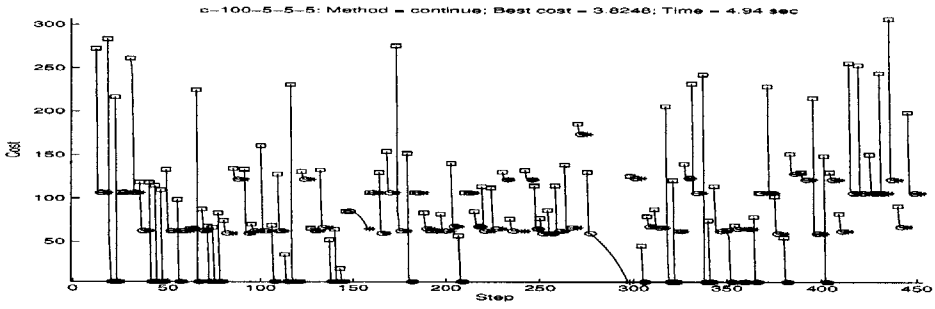
Figure C-1: Dynamics During Iterations of Lloyd's algorithm and Cyclic Exchanges for Dataset c-100-3-5-5: (a) Lloyd's algorithm, (b) Start Cyclic Exchange, (c) Continue Cyclic Exchange, (d) Hybrid Cyclic Exchange.



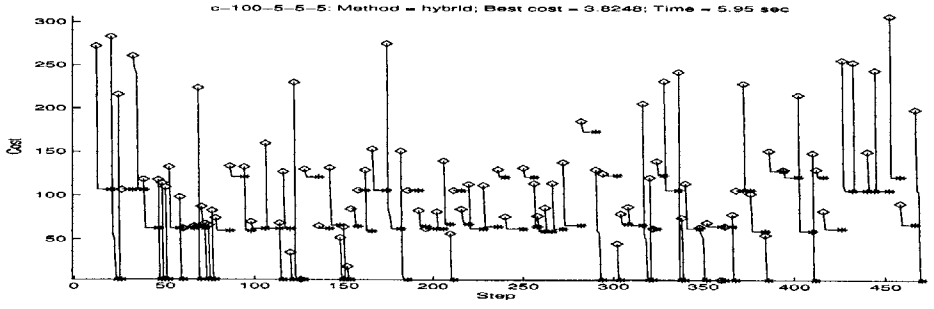
(a)



(b)

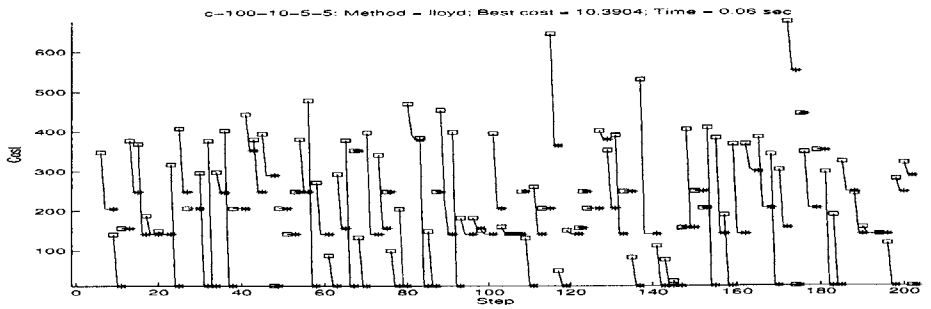


(c)

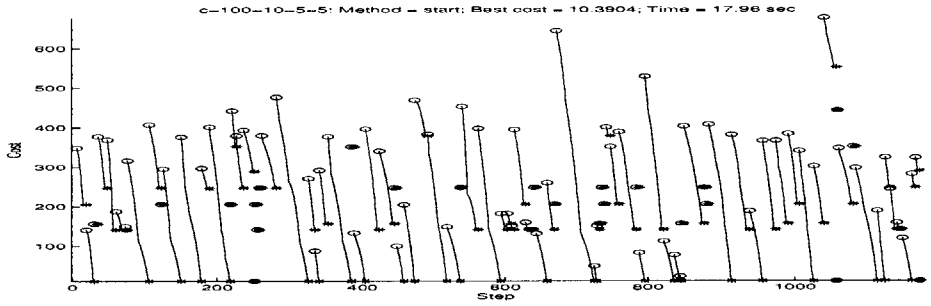


(d)

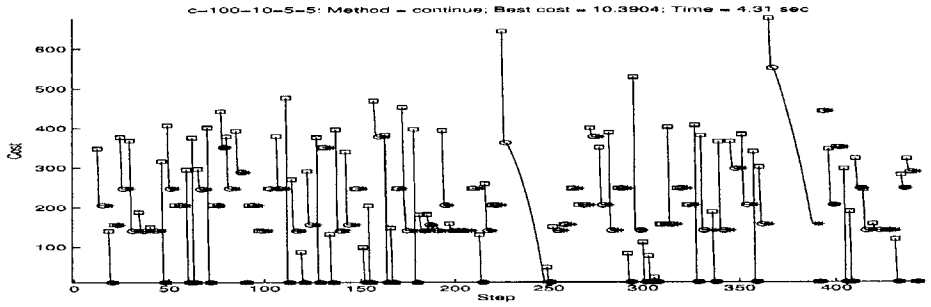
Figure C-2: Dynamics During Iterations of Lloyd's algorithm and Cyclic Exchanges for Dataset c-100-5-5-5: (a) Lloyd's algorithm, (b) Start Cyclic Exchange, (c) Continue Cyclic Exchange, (d) Hybrid Cyclic Exchange.



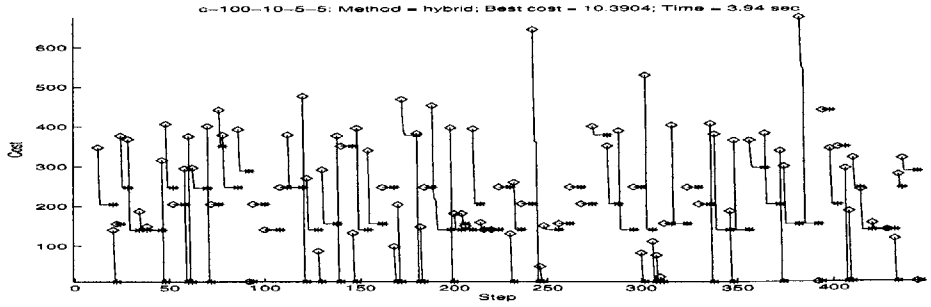
(a)



(b)

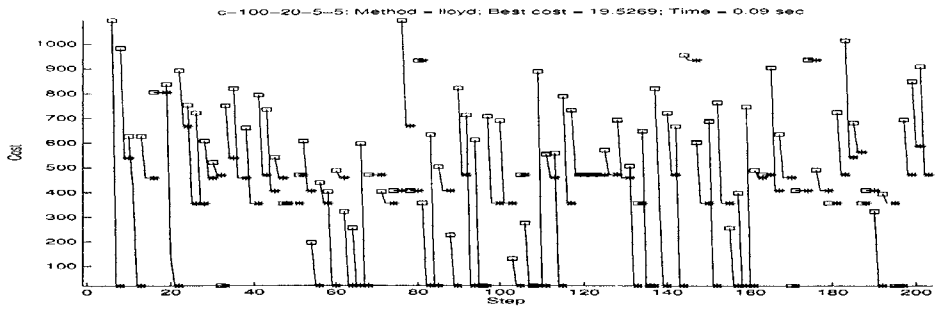


(c)

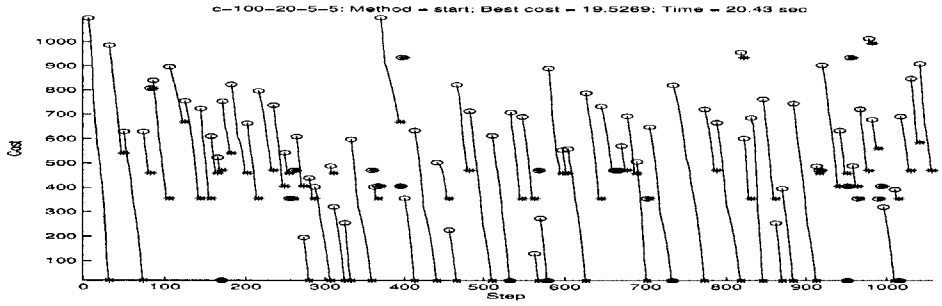


(d)

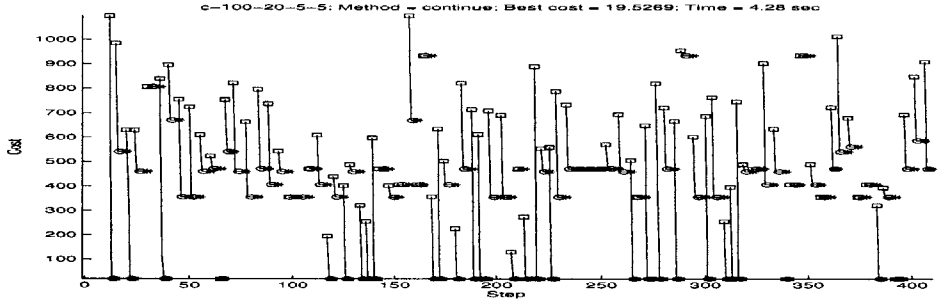
Figure C-3: Dynamics During Iterations of Lloyd's algorithm and Cyclic Exchanges for Dataset c-100-10-5-5: (a) Lloyd's algorithm, (b) Start Cyclic Exchange, (c) Continue Cyclic Exchange, (d) Hybrid Cyclic Exchange.



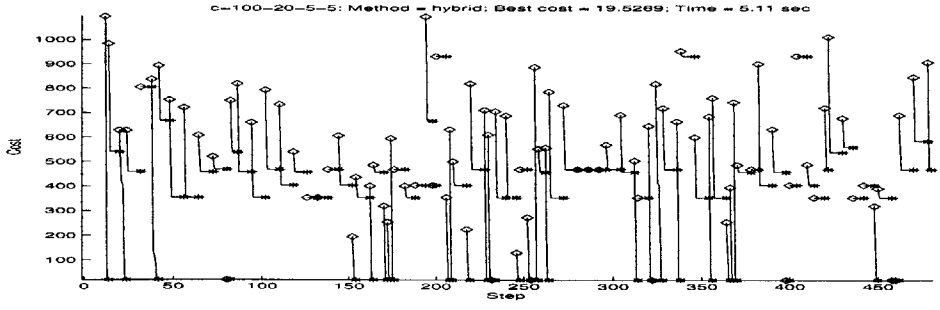
(a)



(b)

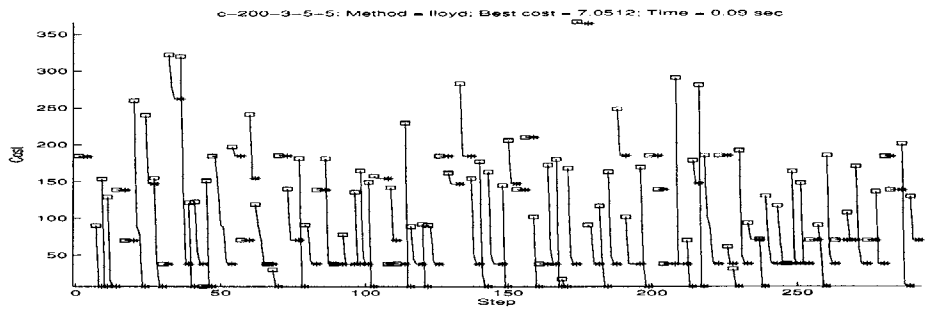


(c)

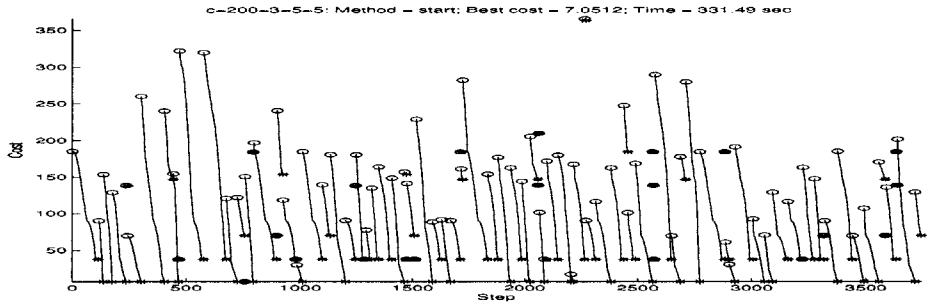


(d)

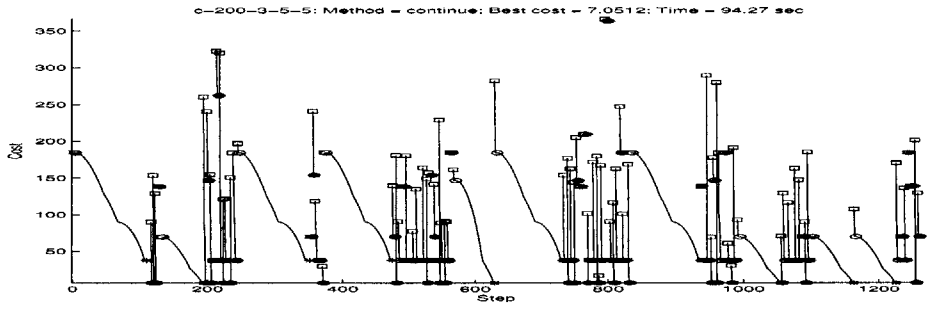
Figure C-4: Dynamics During Iterations of Lloyd's algorithm and Cyclic Exchanges for Dataset c-100-20-5-5: (a) Lloyd's algorithm, (b) Start Cyclic Exchange, (c) Continue Cyclic Exchange, (d) Hybrid Cyclic Exchange.



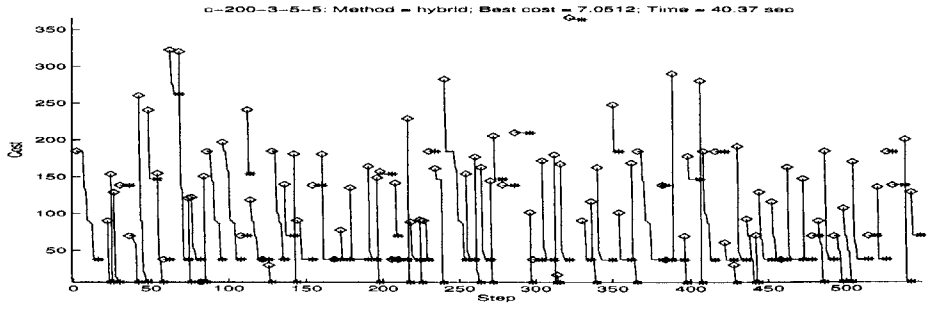
(a)



(b)

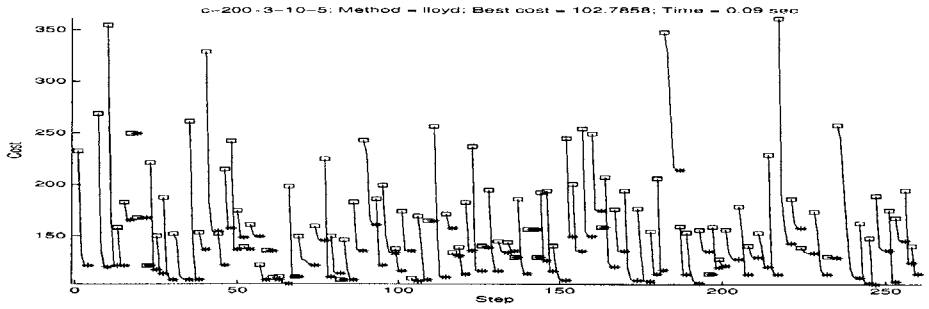


(c)

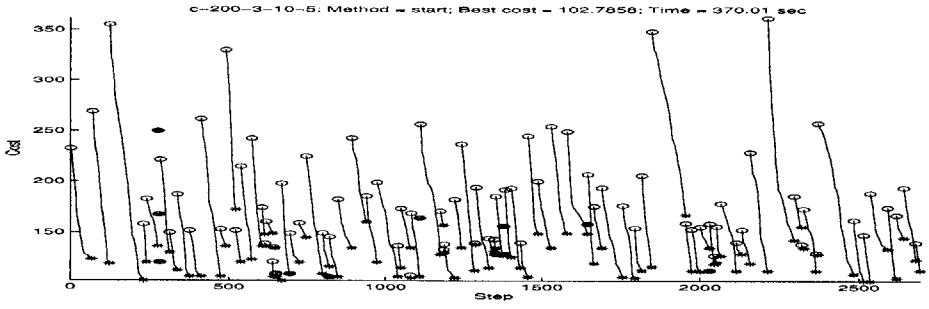


(d)

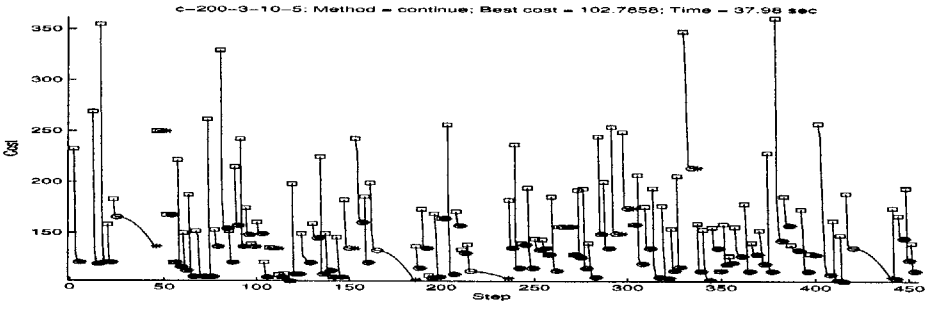
Figure C-5: Dynamics During Iterations of Lloyd's algorithm and Cyclic Exchanges for Dataset c-200-3-5-5: (a) Lloyd's algorithm, (b) Start Cyclic Exchange, (c) Continue Cyclic Exchange, (d) Hybrid Cyclic Exchange.



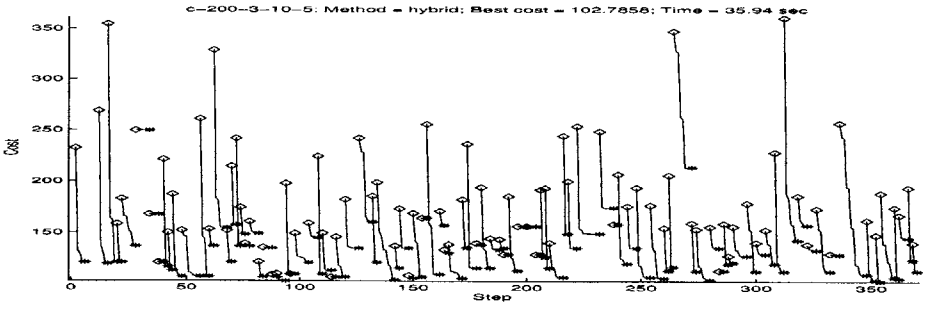
(a)



(b)

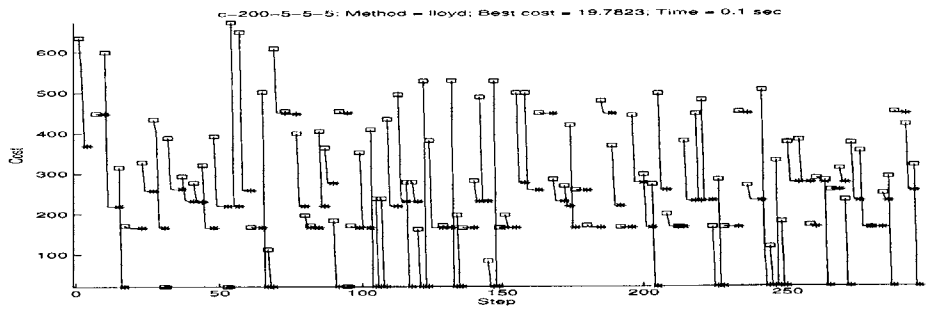


(c)

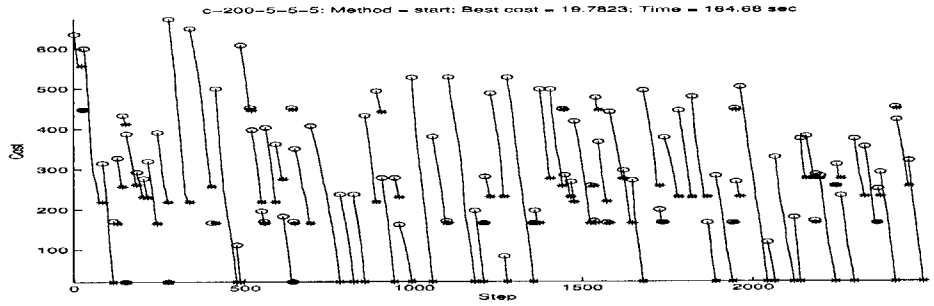


(d)

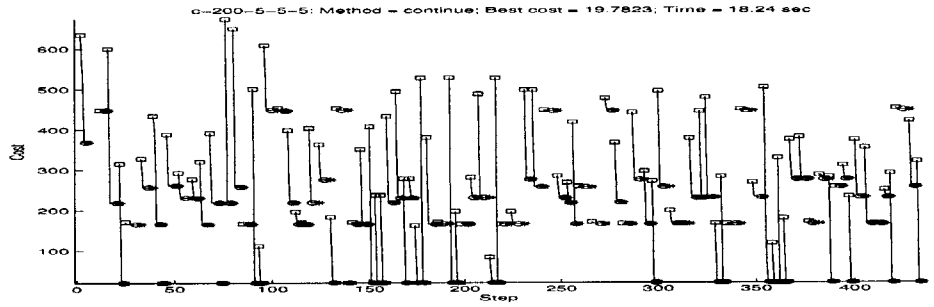
Figure C-6: Dynamics During Iterations of Lloyd's algorithm and Cyclic Exchanges for Dataset c-200-3-10-5: (a) Lloyd's algorithm, (b) Start Cyclic Exchange, (c) Continue Cyclic Exchange, (d) Hybrid Cyclic Exchange.



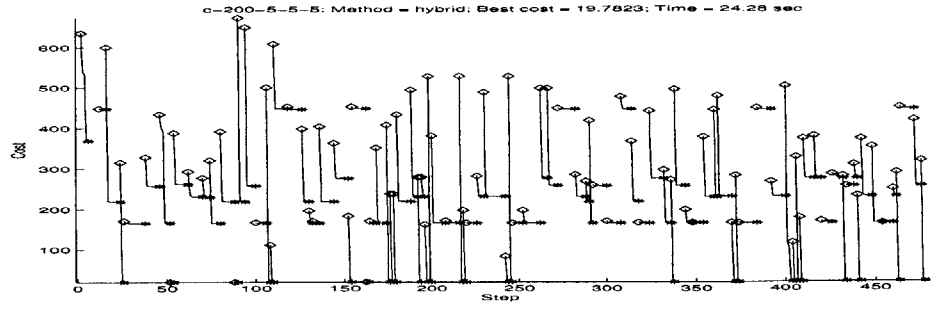
(a)



(b)

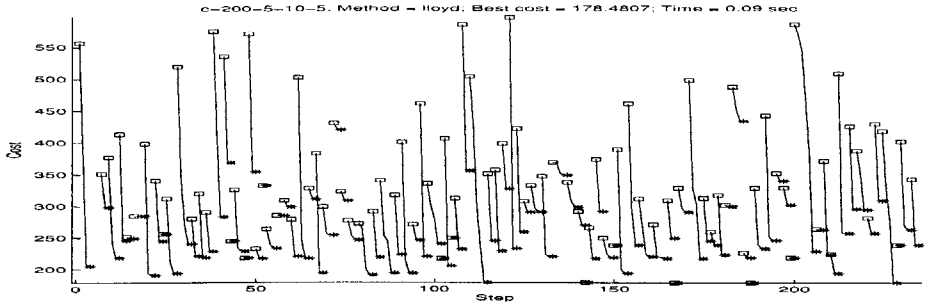


(c)

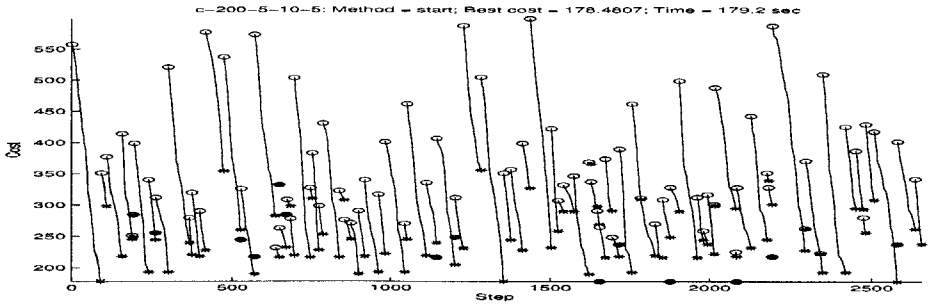


(d)

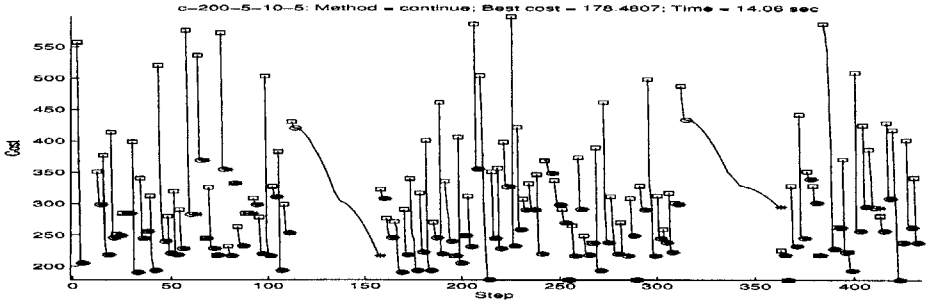
Figure C-7: Dynamics During Iterations of Lloyd's algorithm and Cyclic Exchanges for Dataset c-200-5-5-5: (a) Lloyd's algorithm, (b) Start Cyclic Exchange, (c) Continue Cyclic Exchange, (d) Hybrid Cyclic Exchange.



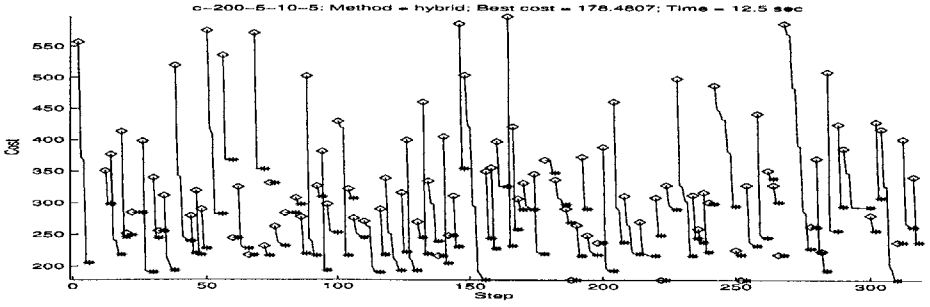
(a)



(b)

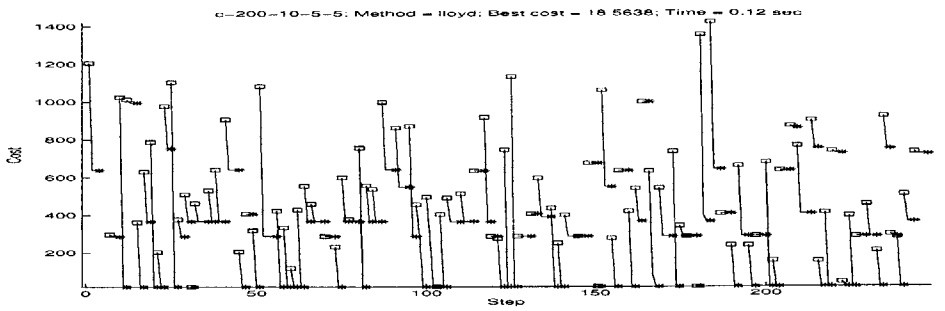


(c)

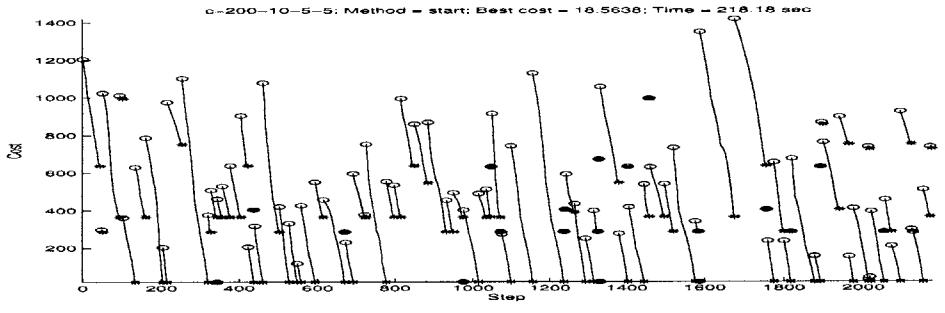


(d)

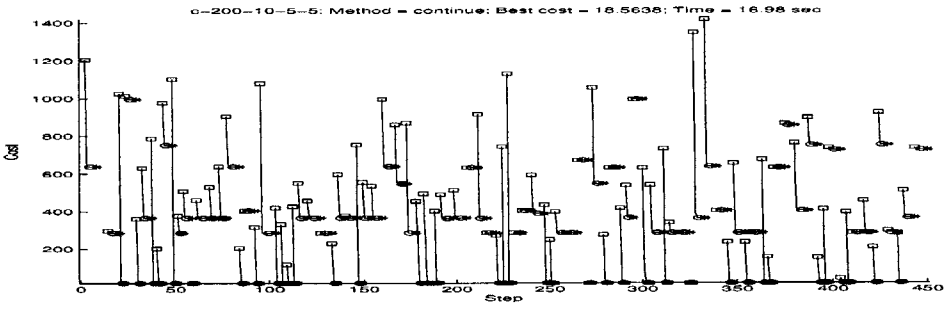
Figure C-8: Dynamics During Iterations of Lloyd's algorithm and Cyclic Exchanges for Dataset c-200-5-10-5: (a) Lloyd's algorithm, (b) Start Cyclic Exchange, (c) Continue Cyclic Exchange, (d) Hybrid Cyclic Exchange.



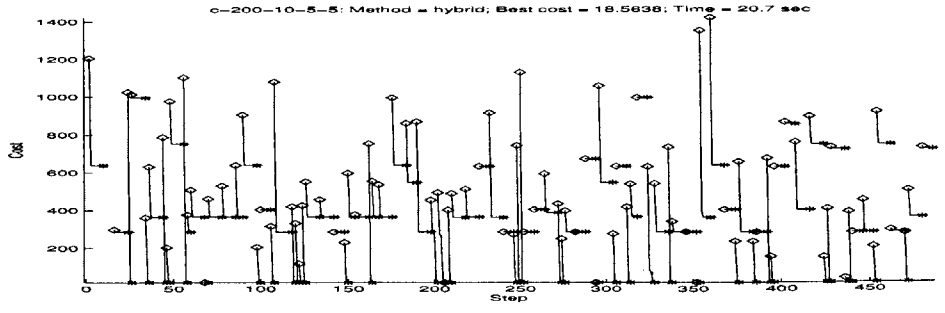
(a)



(b)

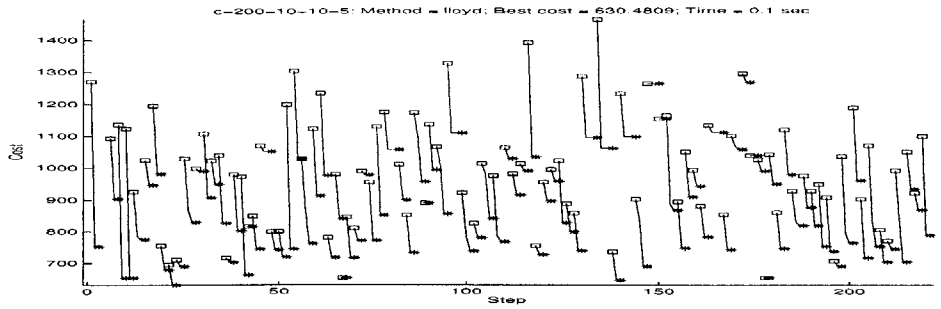


(c)

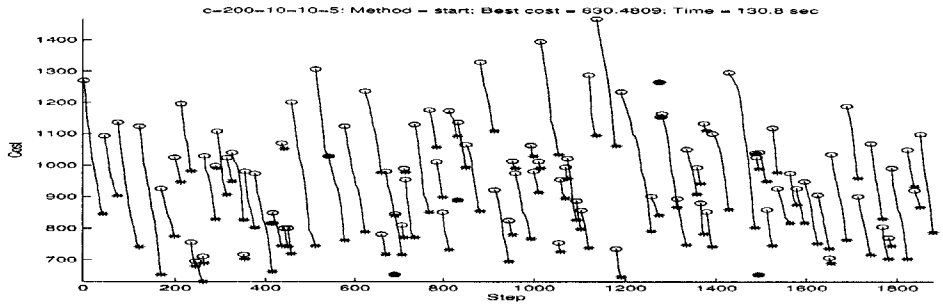


(d)

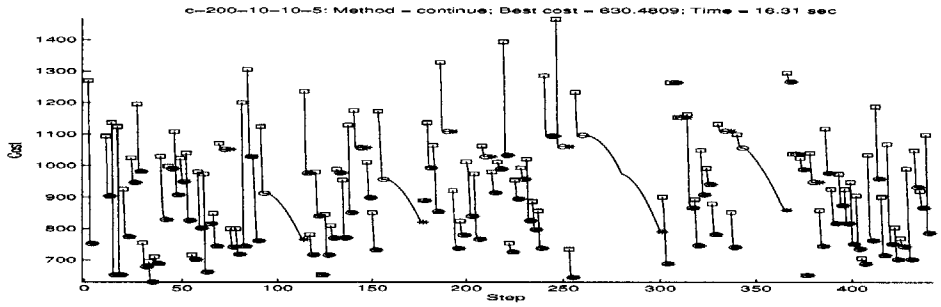
Figure C-9: Dynamics During Iterations of Lloyd's algorithm and Cyclic Exchanges for Dataset c-200-10-5-5: (a) Lloyd's algorithm, (b) Start Cyclic Exchange, (c) Continue Cyclic Exchange, (d) Hybrid Cyclic Exchange.



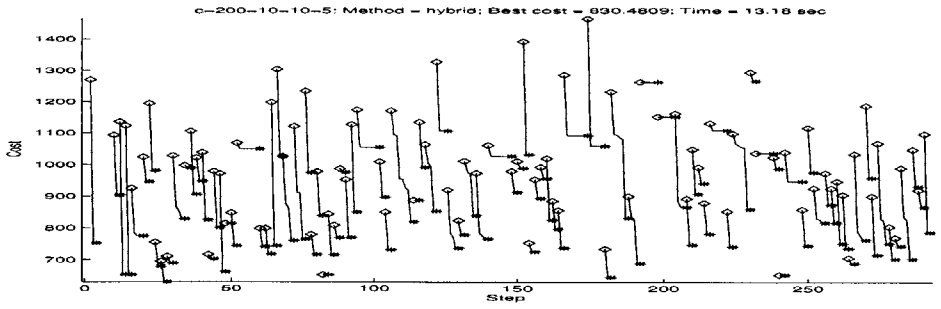
(a)



(b)

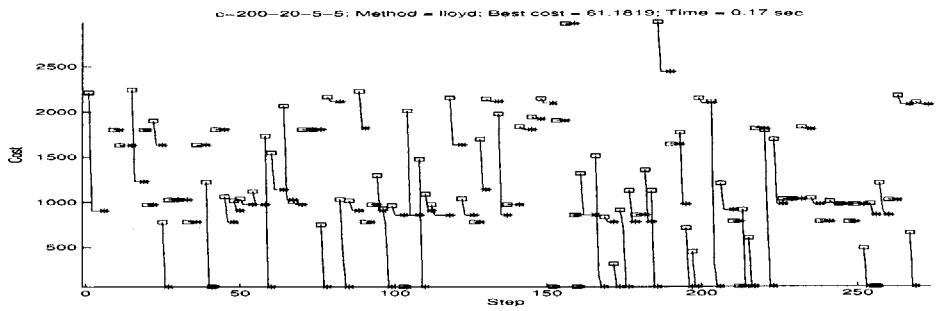


(c)

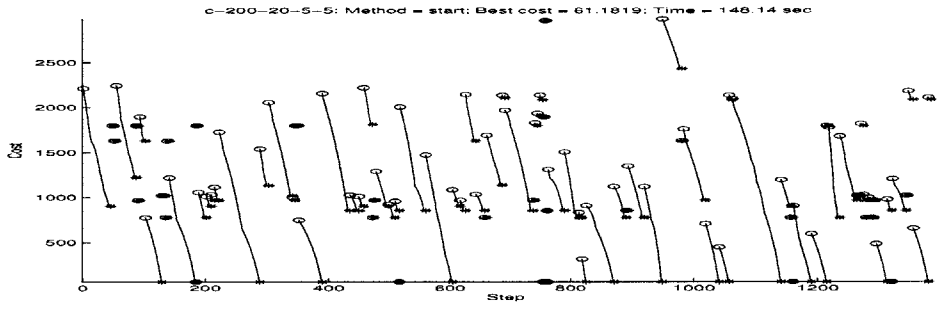


(d)

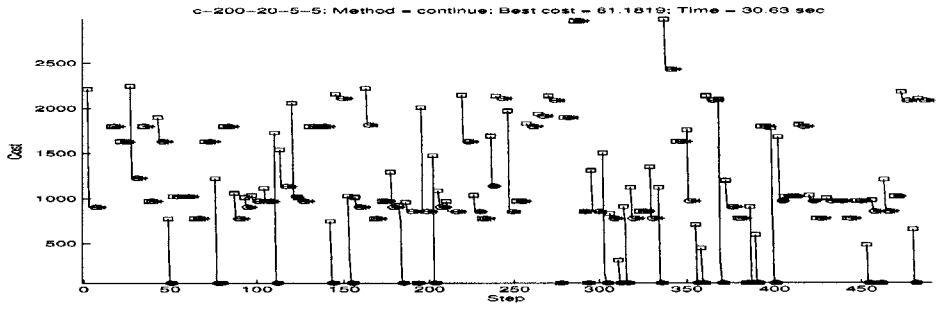
Figure C-10: Dynamics During Iterations of Lloyd's algorithm and Cyclic Exchanges for Dataset c-200-10-10-5: (a) Lloyd's algorithm, (b) Start Cyclic Exchange, (c) Continue Cyclic Exchange, (d) Hybrid Cyclic Exchange.



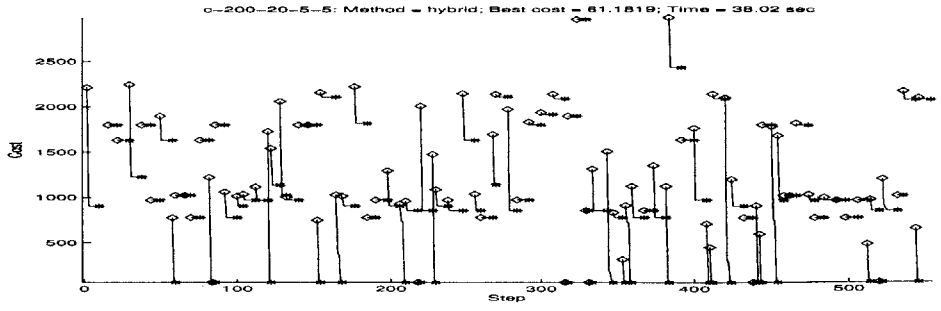
(a)



(b)

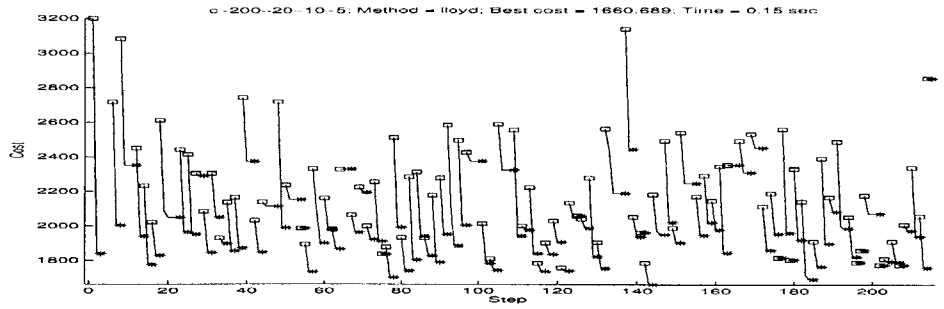


(c)

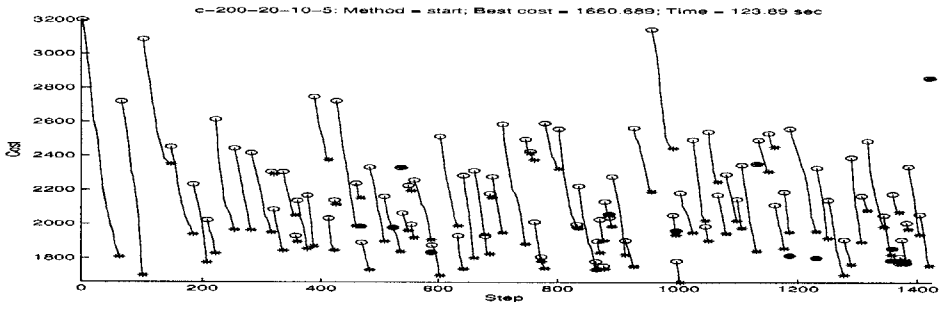


(d)

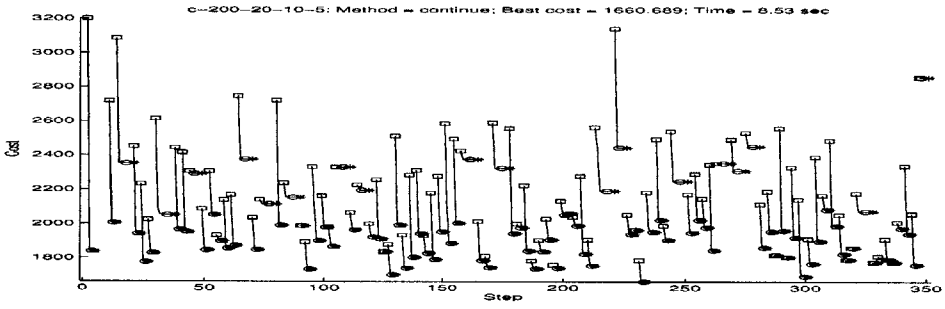
Figure C-11: Dynamics During Iterations of Lloyd's algorithm and Cyclic Exchanges for Dataset c-200-20-5-5: (a) Lloyd's algorithm, (b) Start Cyclic Exchange, (c) Continue Cyclic Exchange, (d) Hybrid Cyclic Exchange.



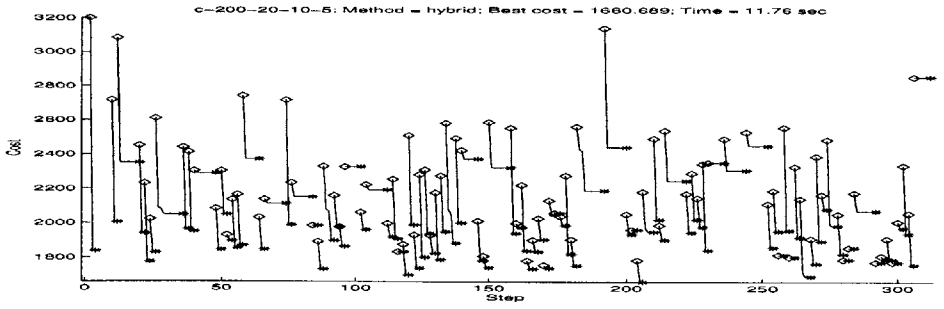
(a)



(b)

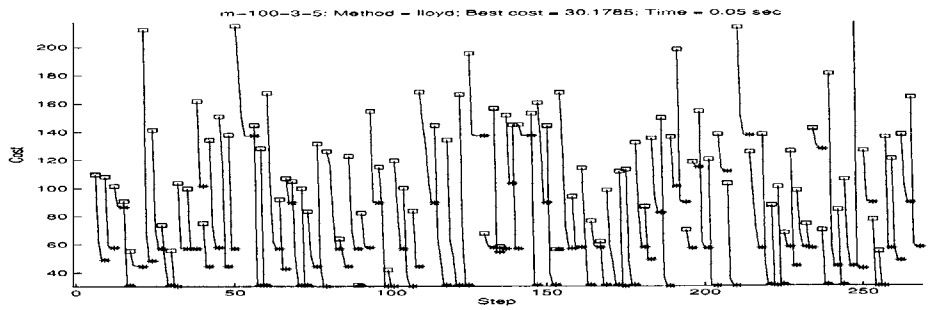


(c)

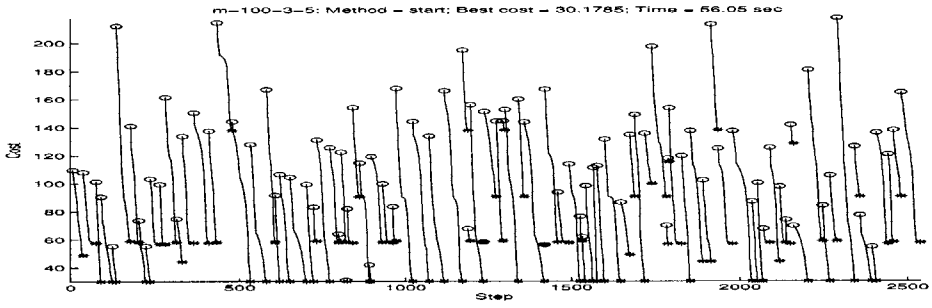


(d)

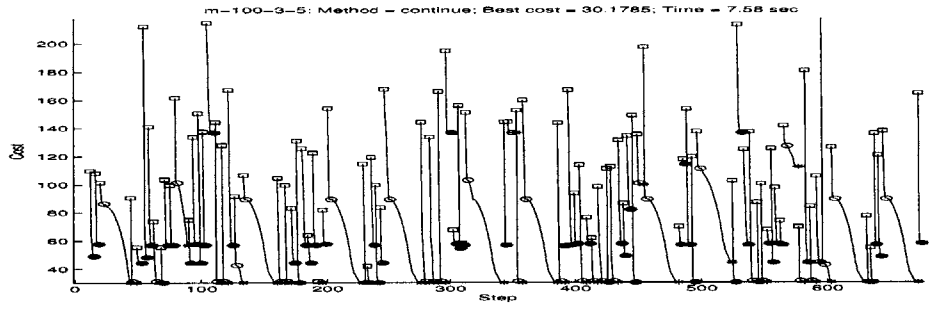
Figure C-12: Dynamics During Iterations of Lloyd's algorithm and Cyclic Exchanges for Dataset c-200-20-10-5: (a) Lloyd's algorithm, (b) Start Cyclic Exchange, (c) Continue Cyclic Exchange, (d) Hybrid Cyclic Exchange.



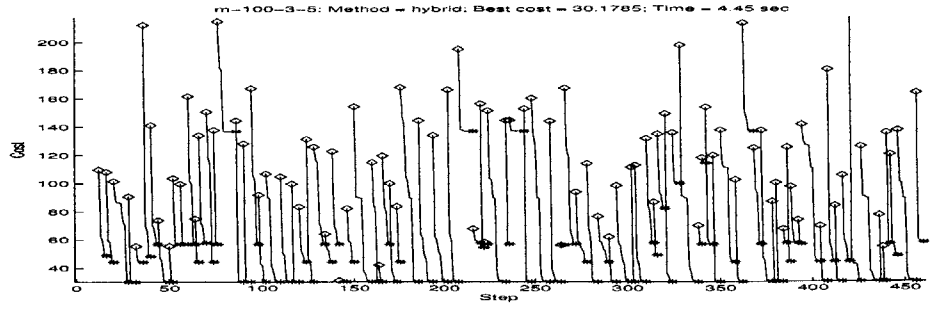
(a)



(b)



(c)



(d)

Figure C-13: Dynamics During Iterations of Lloyd's algorithm and Cyclic Exchanges for Dataset `m-100-3-5`: (a) Lloyd's algorithm, (b) Start Cyclic Exchange, (c) Continue Cyclic Exchange, (d) Hybrid Cyclic Exchange.

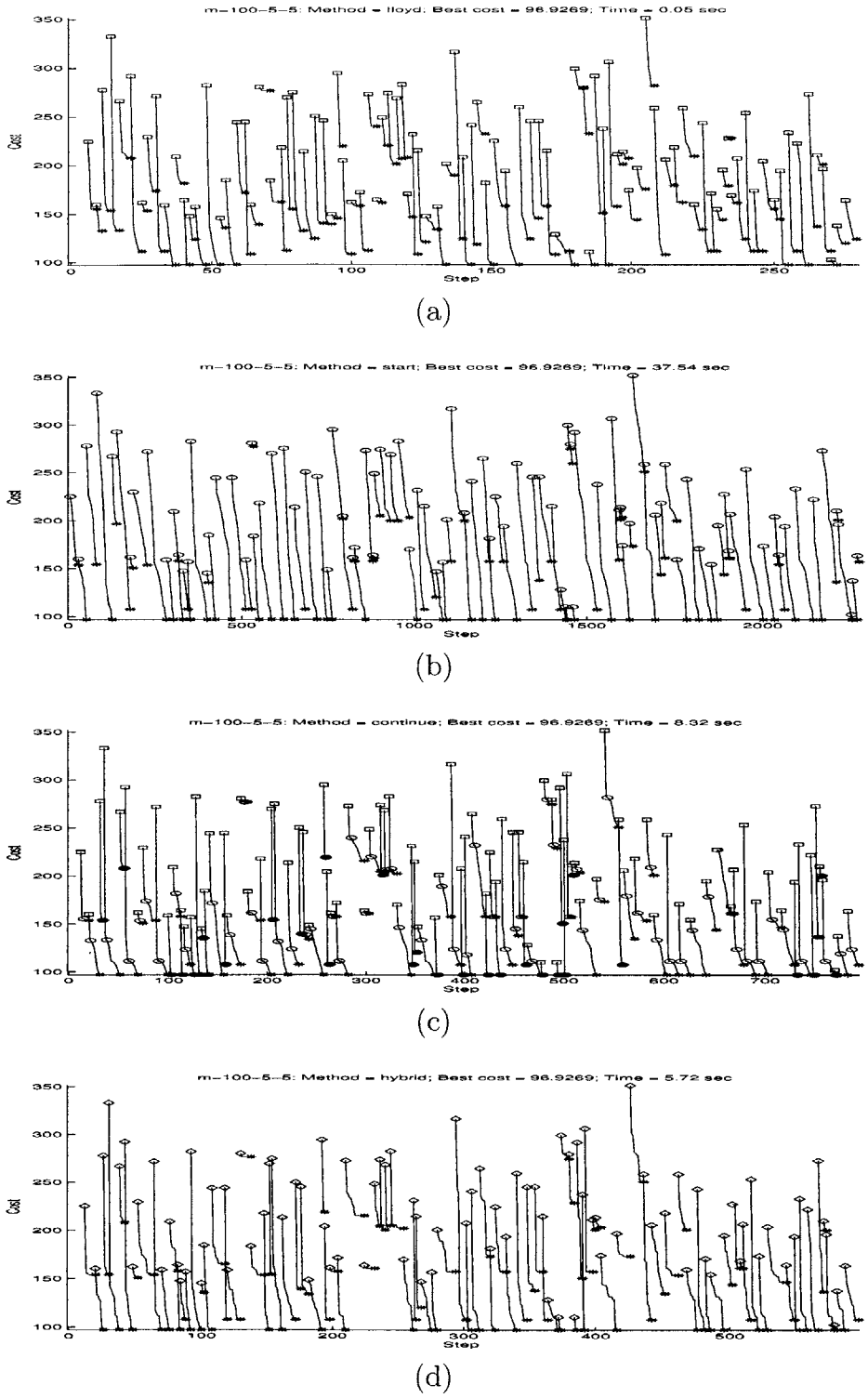
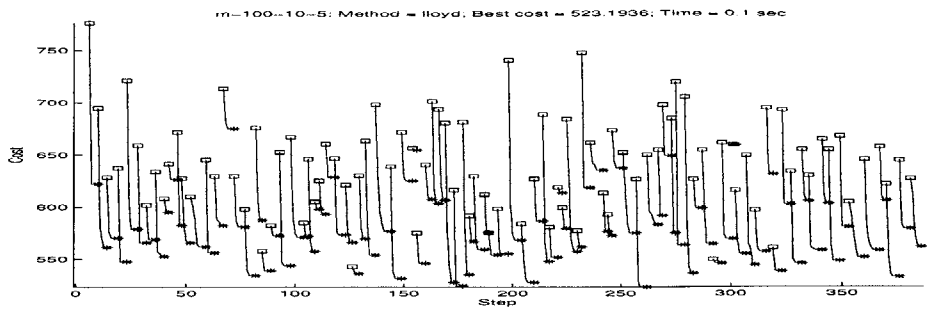
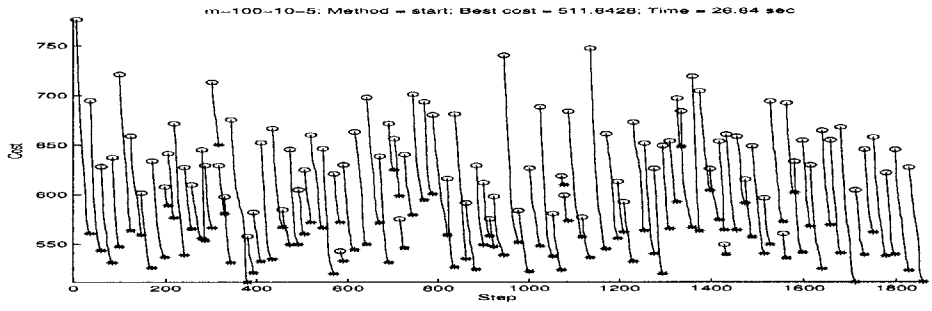


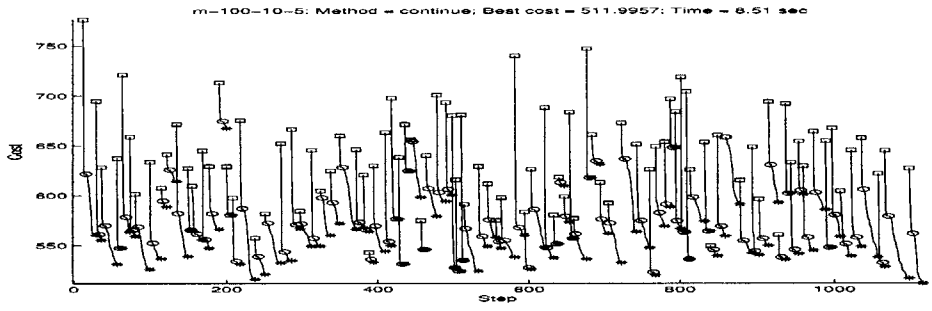
Figure C-14: Dynamics During Iterations of Lloyd's algorithm and Cyclic Exchanges for Dataset `m-100-5-5`: (a) Lloyd's algorithm, (b) Start Cyclic Exchange, (c) Continue Cyclic Exchange, (d) Hybrid Cyclic Exchange.



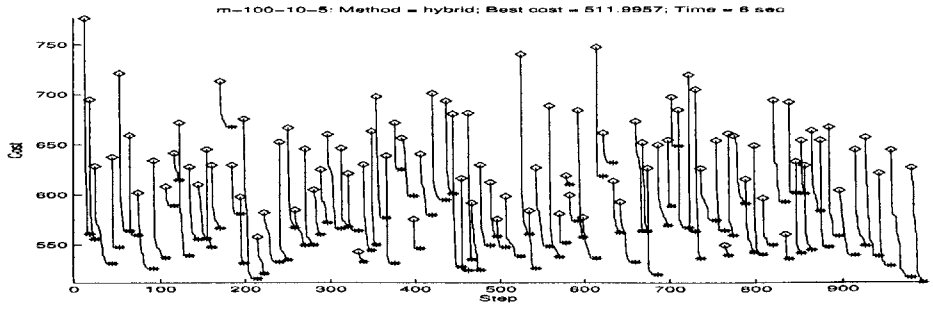
(a)



(b)

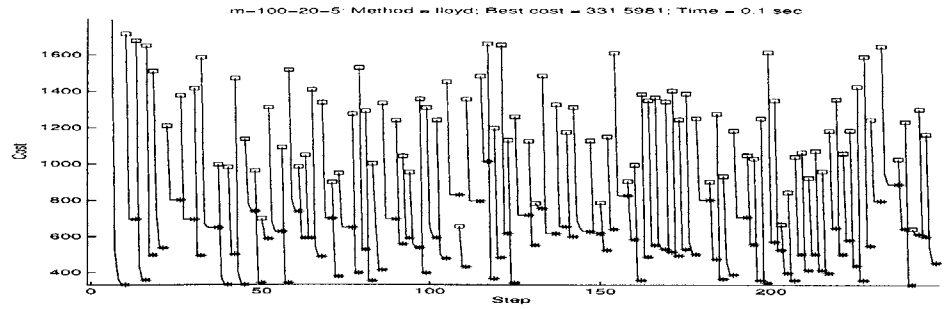


(c)

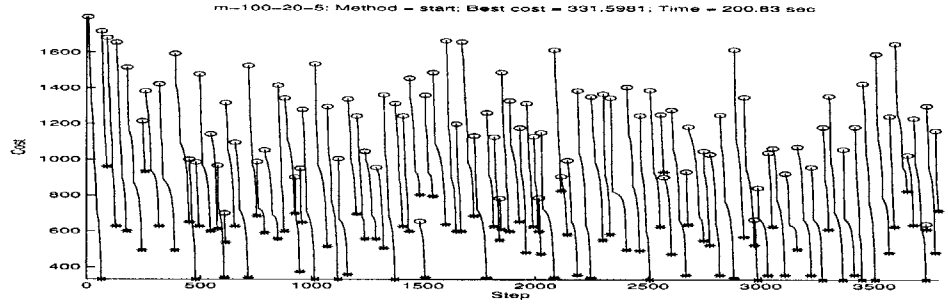


(d)

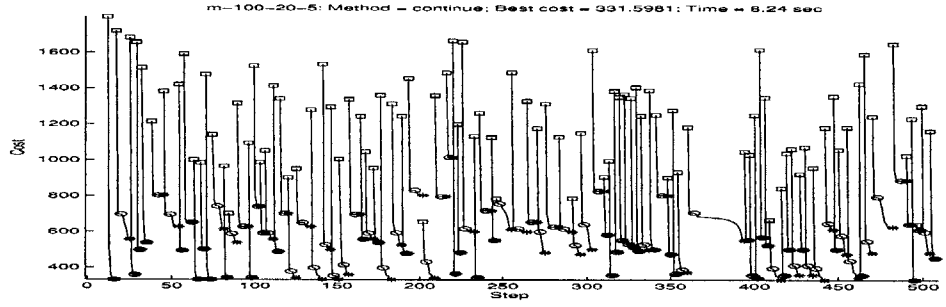
Figure C-15: Dynamics During Iterations of Lloyd's algorithm and Cyclic Exchanges for Dataset `m-100-10-5`: (a) Lloyd's algorithm, (b) Start Cyclic Exchange, (c) Continue Cyclic Exchange, (d) Hybrid Cyclic Exchange.



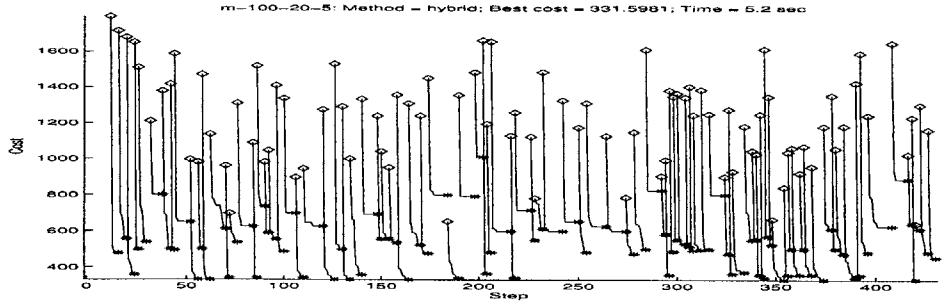
(a)



(b)

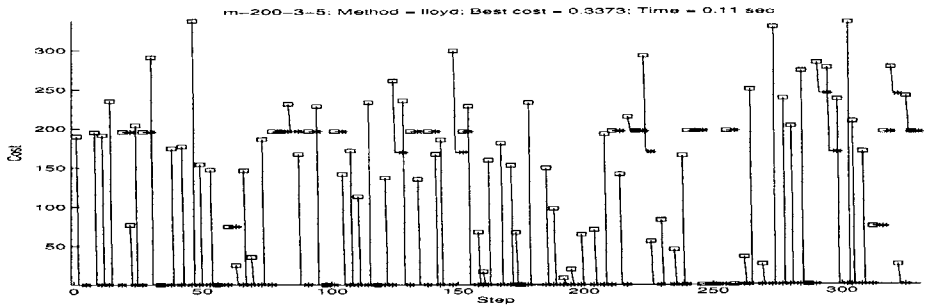


(c)

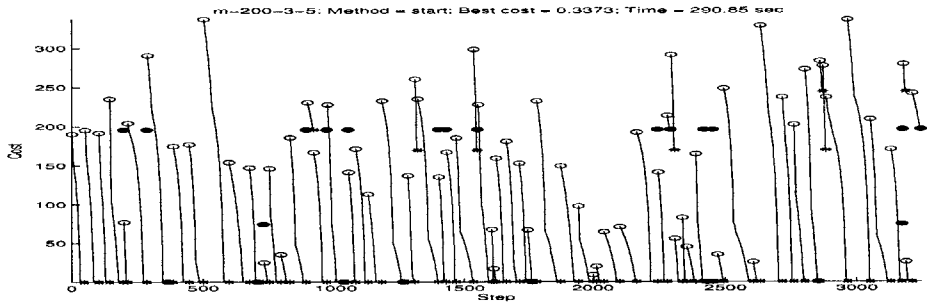


(d)

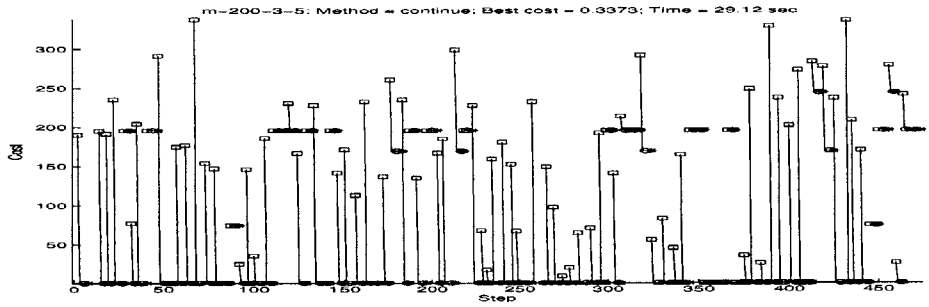
Figure C-16: Dynamics During Iterations of Lloyd's algorithm and Cyclic Exchanges for Dataset `m-100-20-5`: (a) Lloyd's algorithm, (b) Start Cyclic Exchange, (c) Continue Cyclic Exchange, (d) Hybrid Cyclic Exchange.



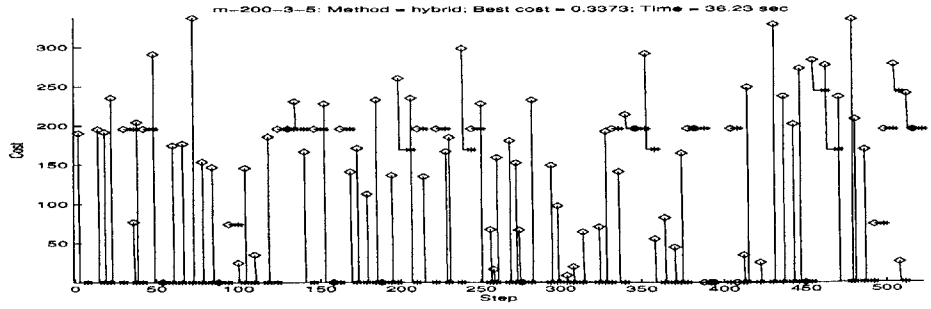
(a)



(b)

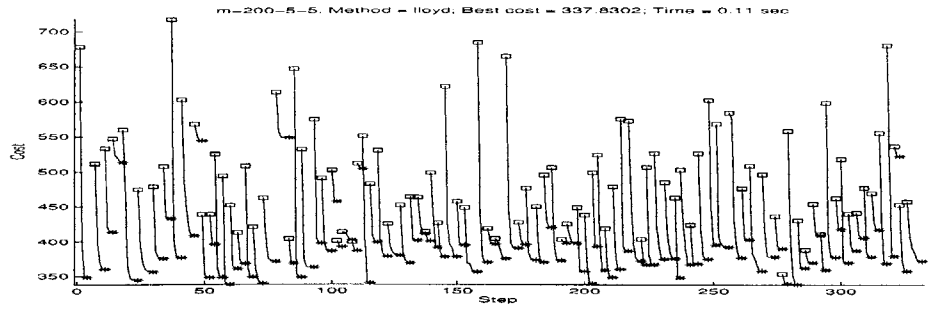


(c)

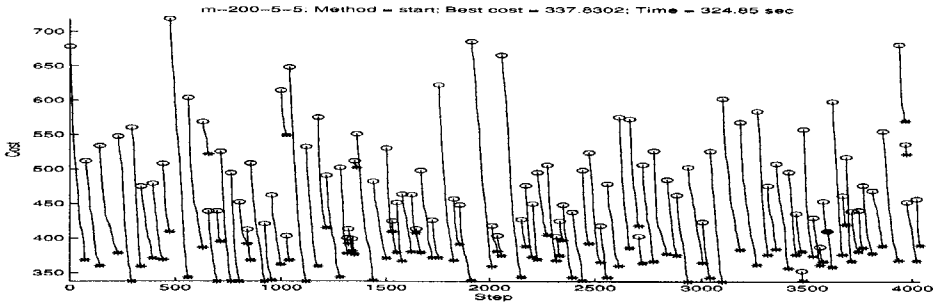


(d)

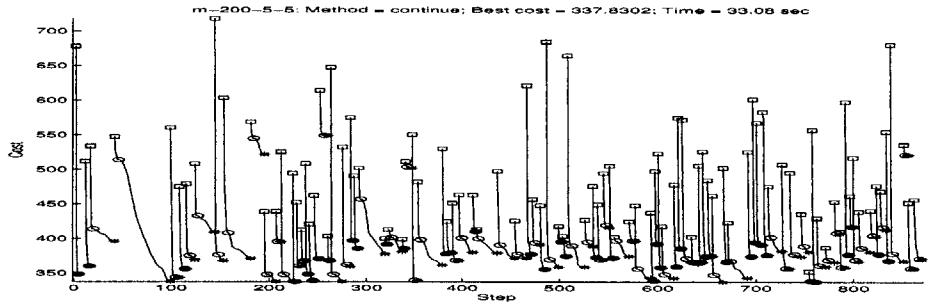
Figure C-17: Dynamics During Iterations of Lloyd's algorithm and Cyclic Exchanges for Dataset m-200-3-5: (a) Lloyd's algorithm, (b) Start Cyclic Exchange, (c) Continue Cyclic Exchange, (d) Hybrid Cyclic Exchange.



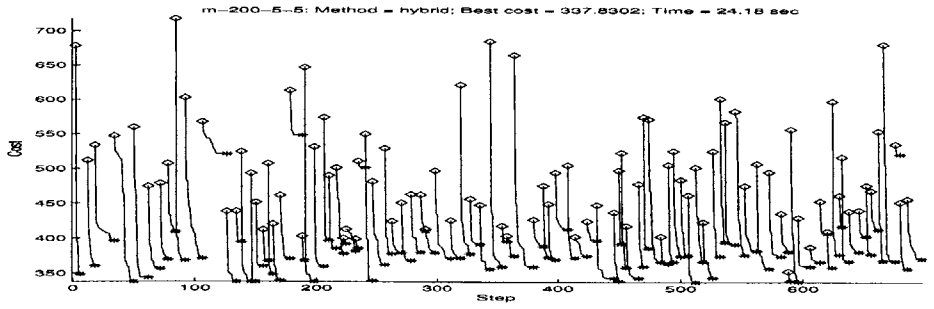
(a)



(b)

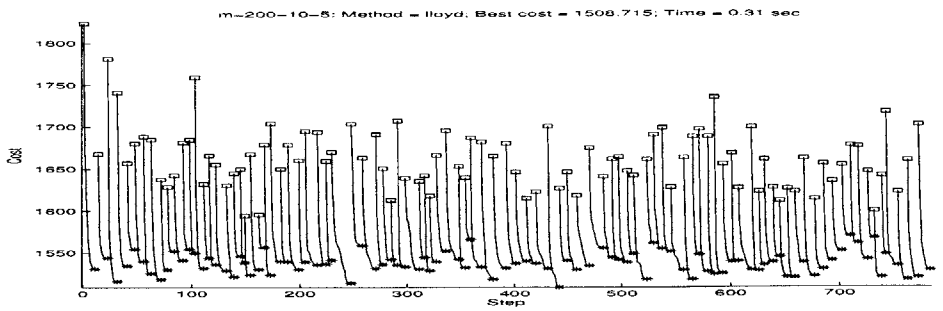


(c)

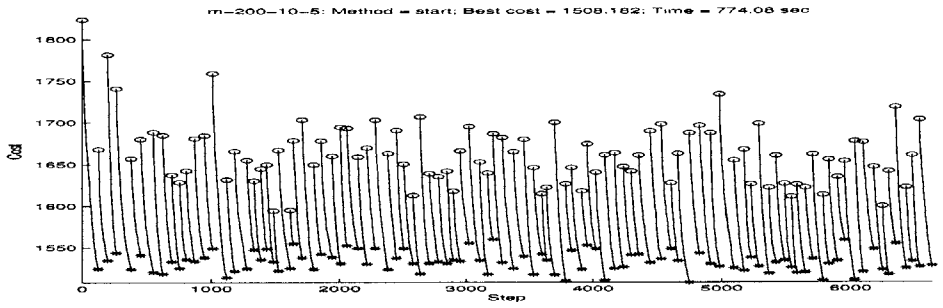


(d)

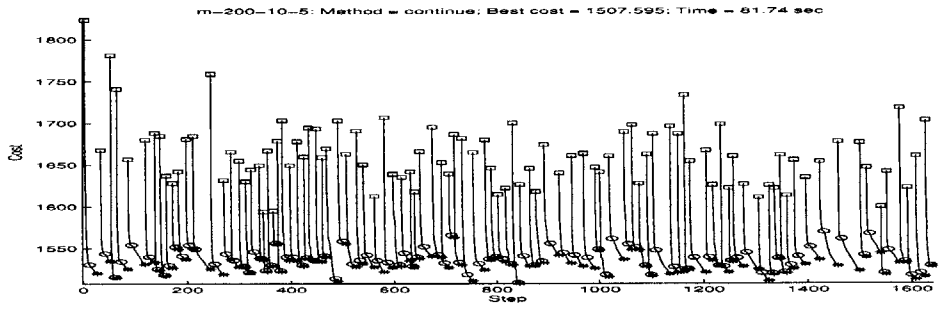
Figure C-18: Dynamics During Iterations of Lloyd's algorithm and Cyclic Exchanges for Dataset `m-200-5-5`: (a) Lloyd's algorithm, (b) Start Cyclic Exchange, (c) Continue Cyclic Exchange, (d) Hybrid Cyclic Exchange.



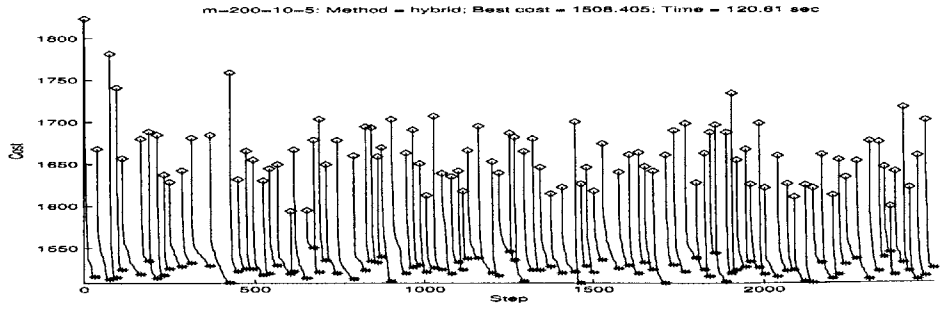
(a)



(b)

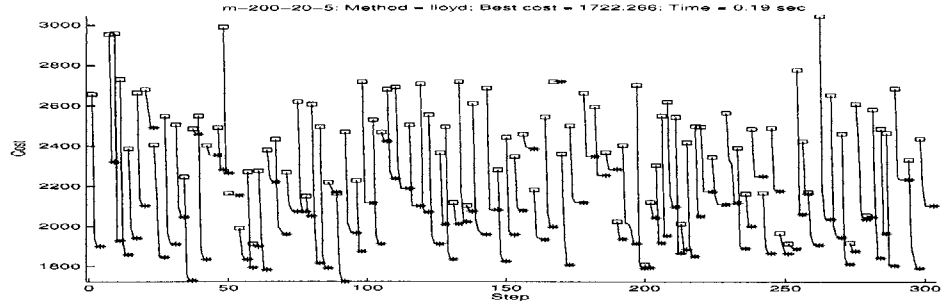


(c)

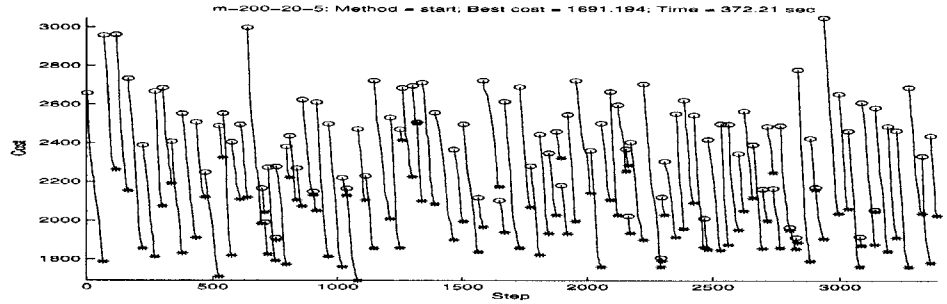


(d)

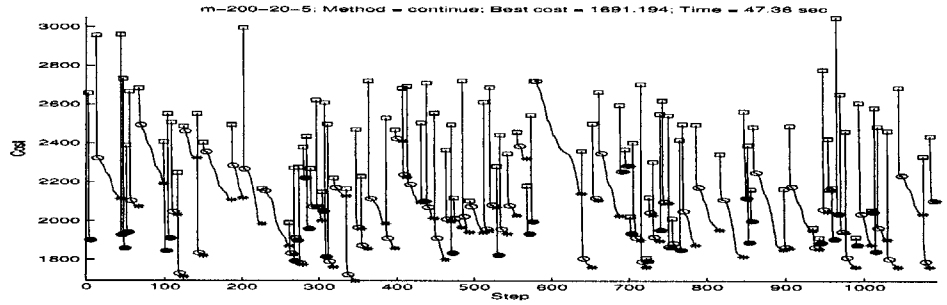
Figure C-19: Dynamics During Iterations of Lloyd's algorithm and Cyclic Exchanges for Dataset `m-200-10-5`: (a) Lloyd's algorithm, (b) Start Cyclic Exchange, (c) Continue Cyclic Exchange, (d) Hybrid Cyclic Exchange.



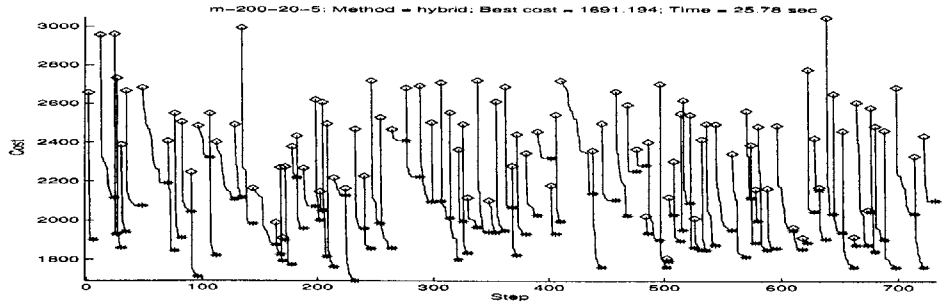
(a)



(b)

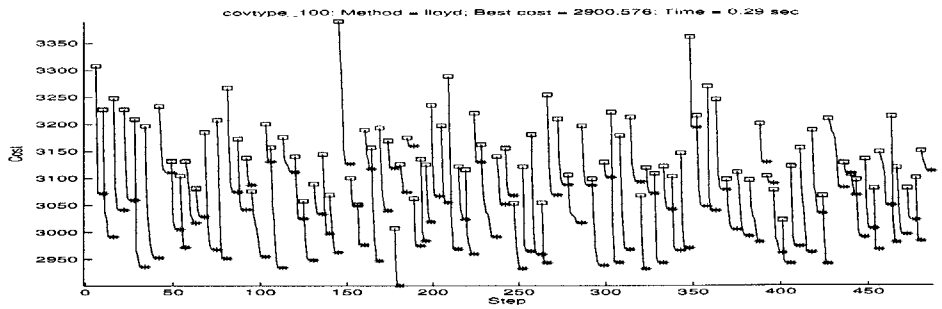


(c)

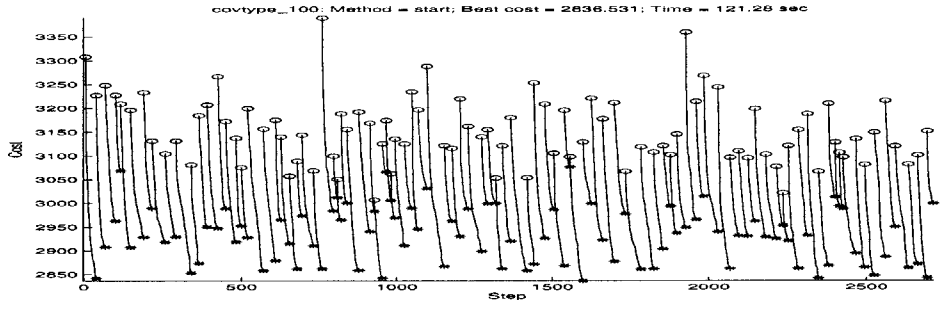


(d)

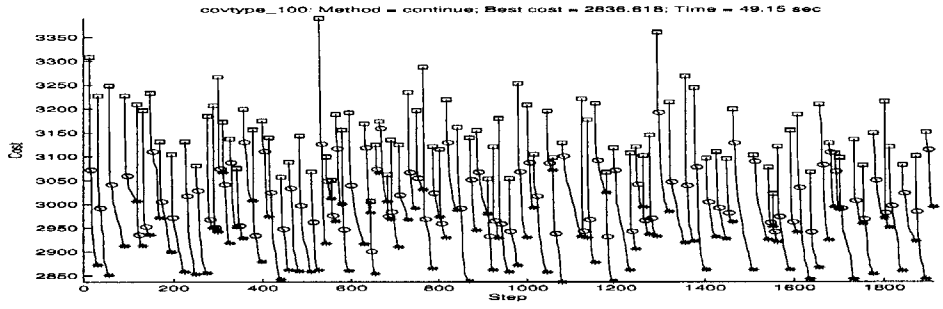
Figure C-20: Dynamics During Iterations of Lloyd's algorithm and Cyclic Exchanges for Dataset `m-200-20-5`: (a) Lloyd's algorithm, (b) Start Cyclic Exchange, (c) Continue Cyclic Exchange, (d) Hybrid Cyclic Exchange.



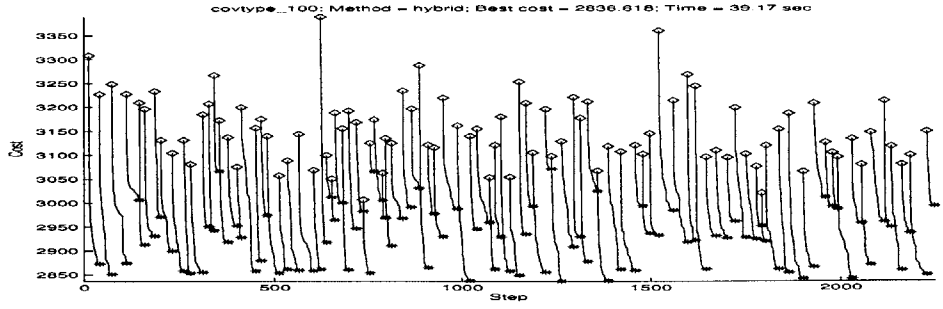
(a)



(b)

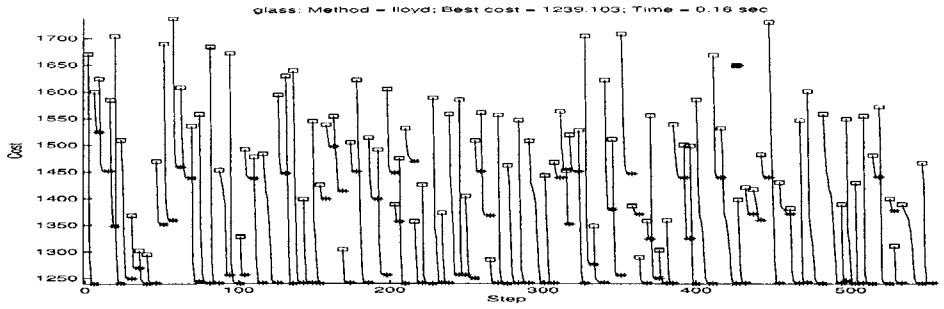


(c)

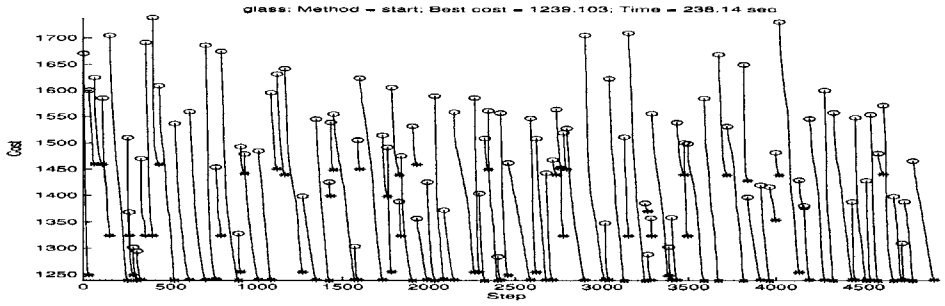


(d)

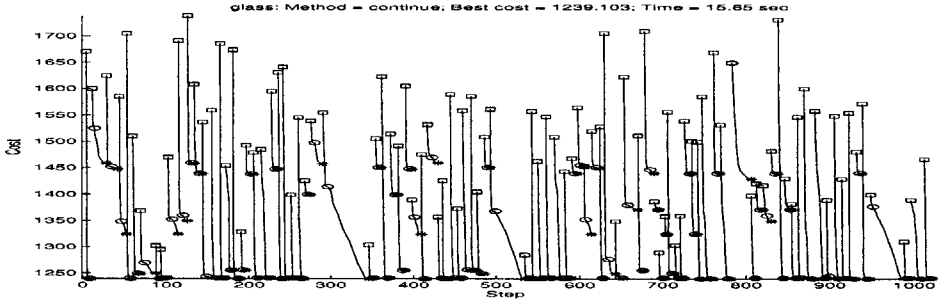
Figure C-21: Dynamics During Iterations of Lloyd's algorithm and Cyclic Exchanges for Dataset `covtype-100`: (a) Lloyd's algorithm, (b) Start Cyclic Exchange, (c) Continue Cyclic Exchange, (d) Hybrid Cyclic Exchange.



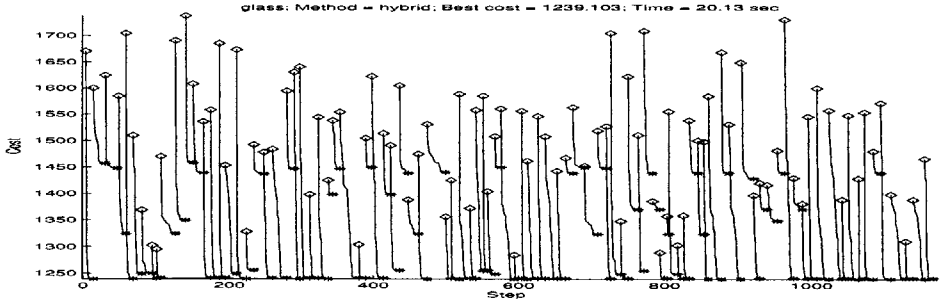
(a)



(b)

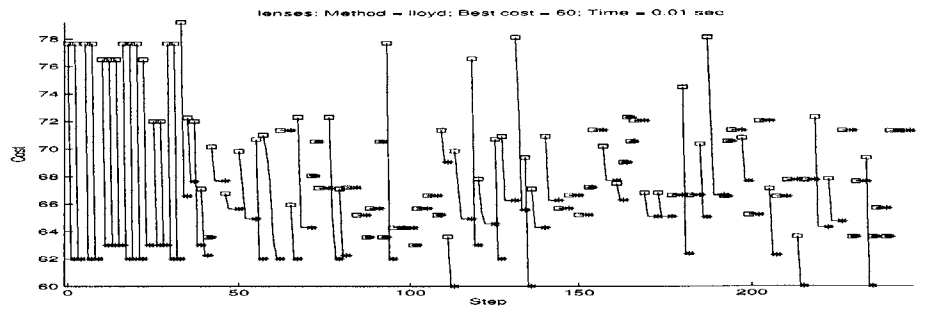


(c)

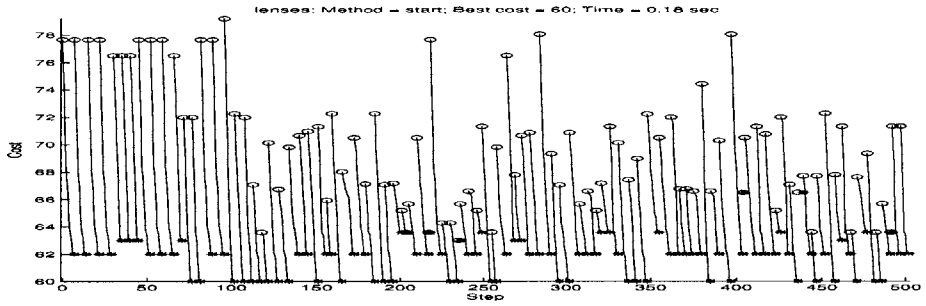


(d)

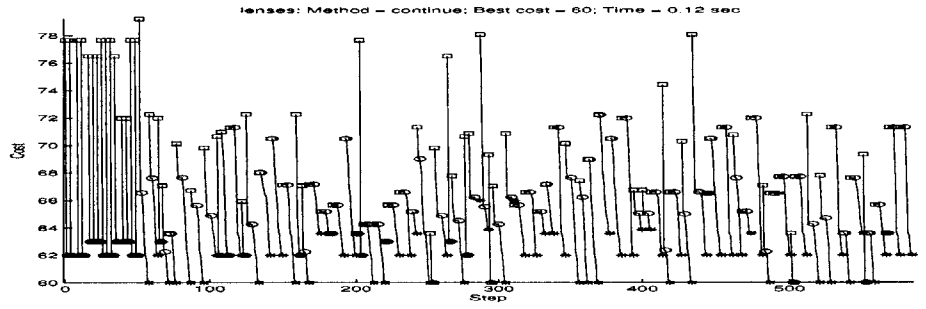
Figure C-22: Dynamics During Iterations of Lloyd's algorithm and Cyclic Exchanges for Dataset `glass`: (a) Lloyd's algorithm, (b) Start Cyclic Exchange, (c) Continue Cyclic Exchange, (d) Hybrid Cyclic Exchange.



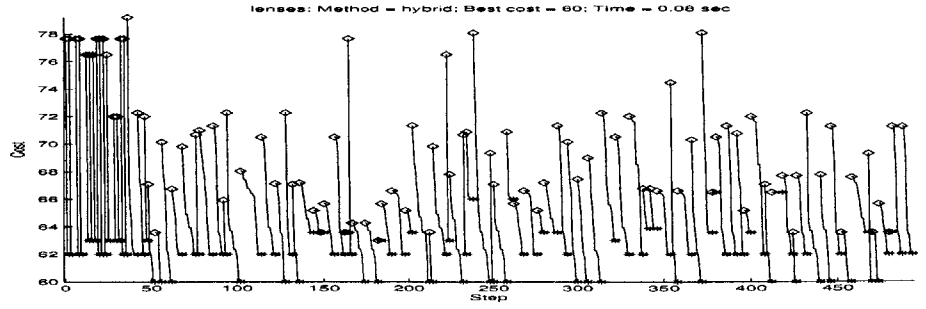
(a)



(b)

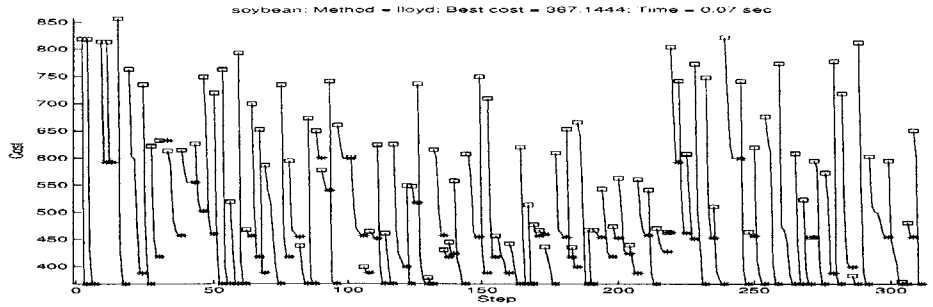


(c)

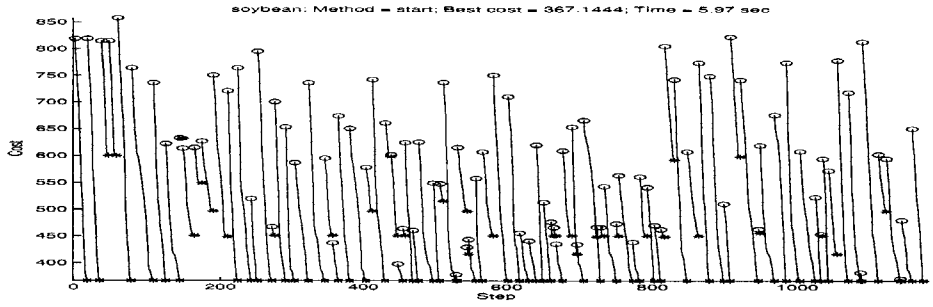


(d)

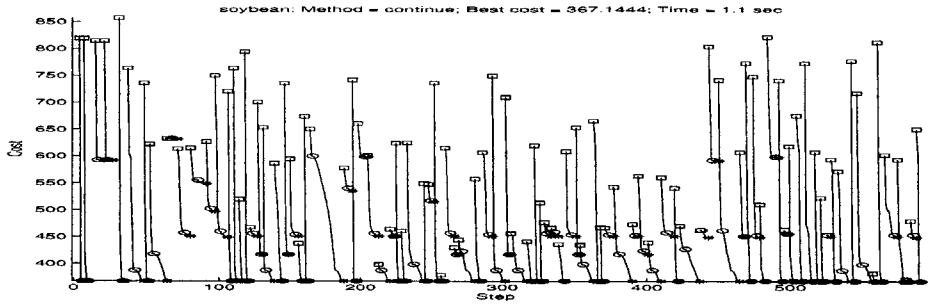
Figure C-23: Dynamics During Iterations of Lloyd's algorithm and Cyclic Exchanges for Dataset `lenses`: (a) Lloyd's algorithm, (b) Start Cyclic Exchange, (c) Continue Cyclic Exchange, (d) Hybrid Cyclic Exchange.



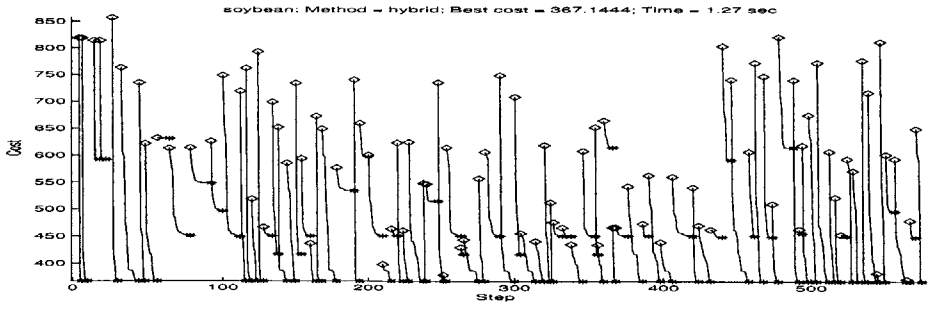
(a)



(b)

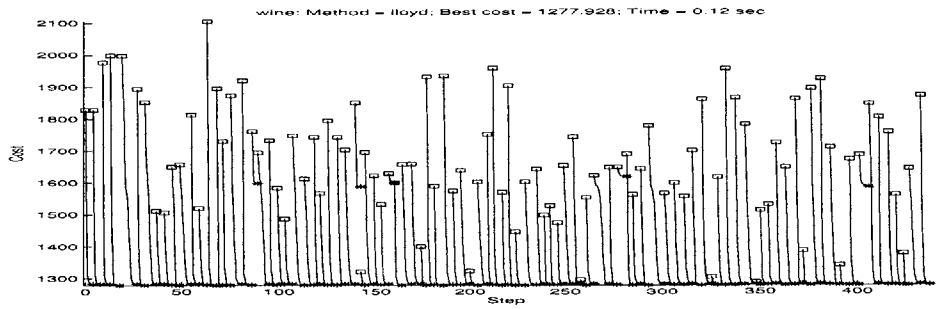


(c)

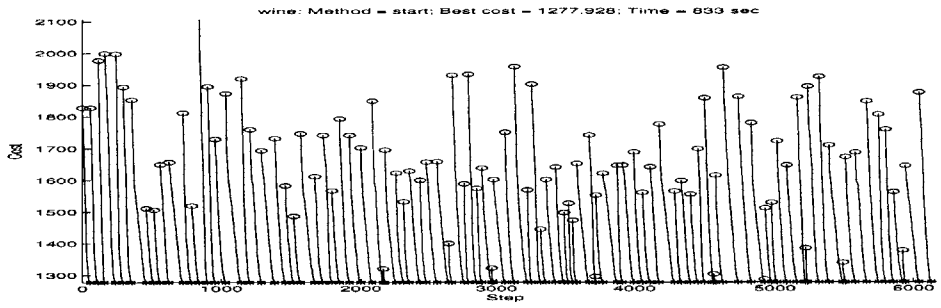


(d)

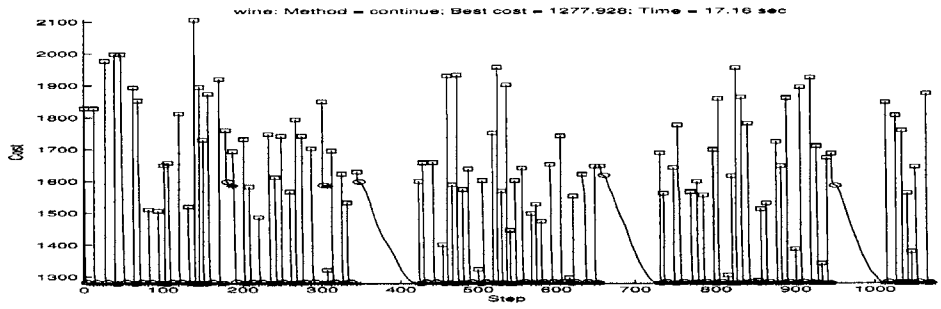
Figure C-24: Dynamics During Iterations of Lloyd's algorithm and Cyclic Exchanges for Dataset **soybean**: (a) Lloyd's algorithm, (b) Start Cyclic Exchange, (c) Continue Cyclic Exchange, (d) Hybrid Cyclic Exchange.



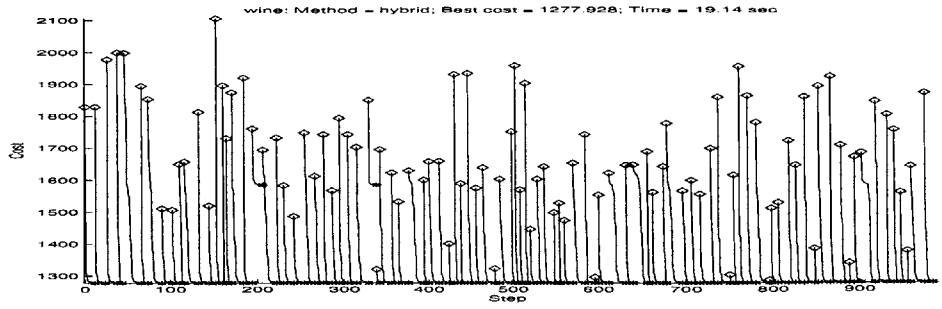
(a)



(b)

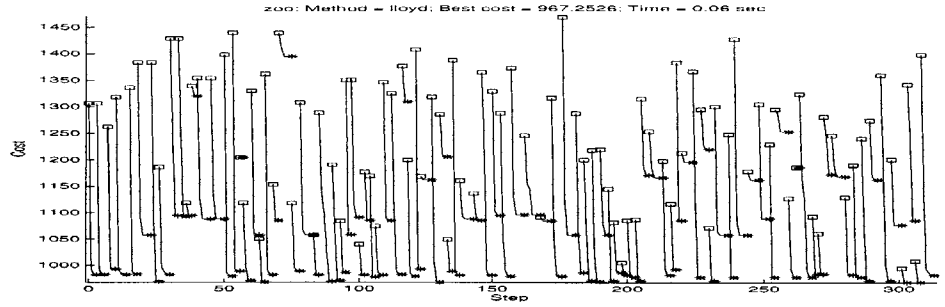


(c)

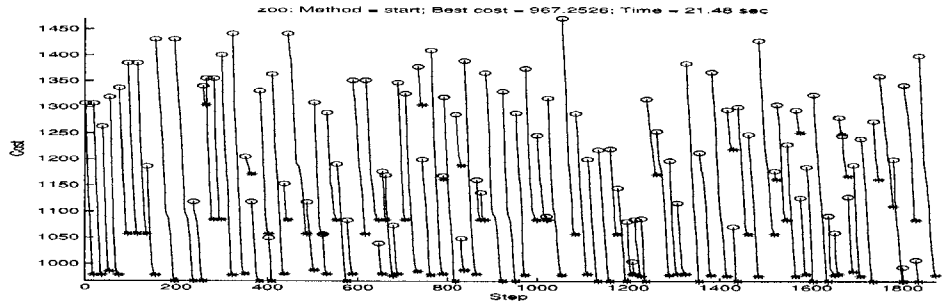


(d)

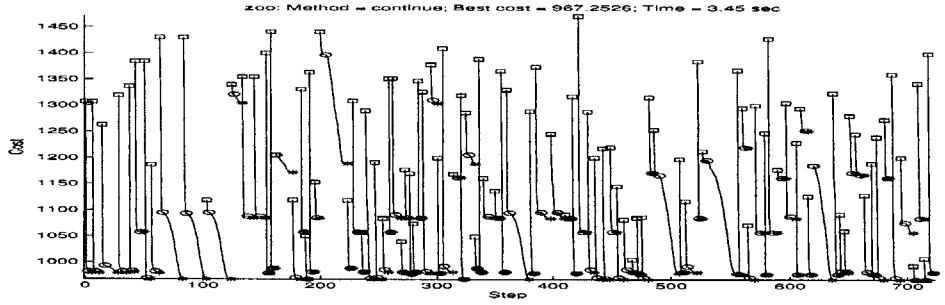
Figure C-25: Dynamics During Iterations of Lloyd's algorithm and Cyclic Exchanges for Dataset `wine`: (a) Lloyd's algorithm, (b) Start Cyclic Exchange, (c) Continue Cyclic Exchange, (d) Hybrid Cyclic Exchange.



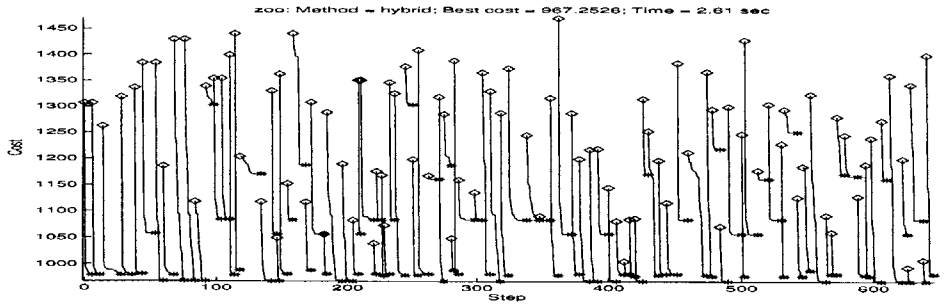
(a)



(b)



(c)



(d)

Figure C-26: Dynamics During Iterations of Lloyd's algorithm and Cyclic Exchanges for Dataset zoo: (a) Lloyd's algorithm, (b) Start Cyclic Exchange, (c) Continue Cyclic Exchange, (d) Hybrid Cyclic Exchange.

Appendix D

Locals Minima from 10000 Iterations of Lloyd's Algorithm

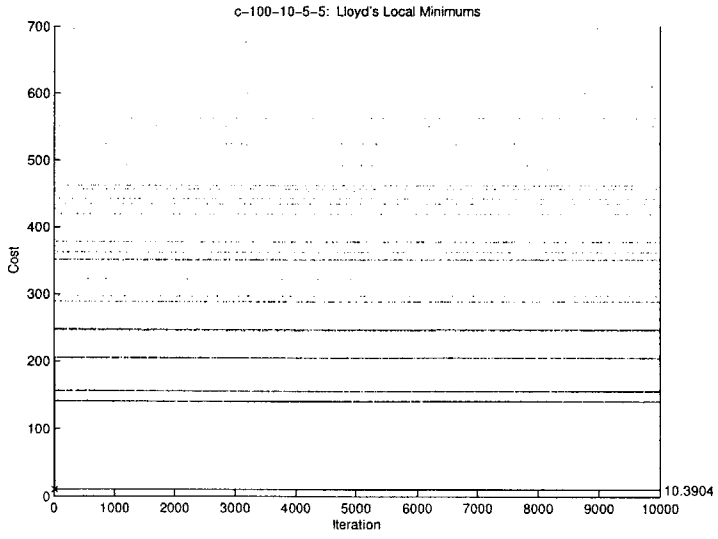


Figure D-1: Costs of Local Minima from Lloyd's Algorithm in 10000 Iterations for dataset c-100-10-5-5

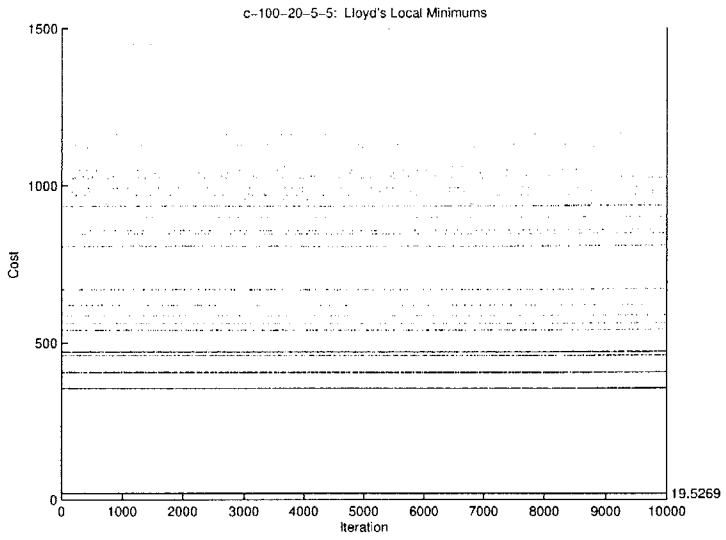


Figure D-2: Costs of Local Minima from Lloyd's Algorithm in 10000 Iterations for dataset c-100-20-5-5

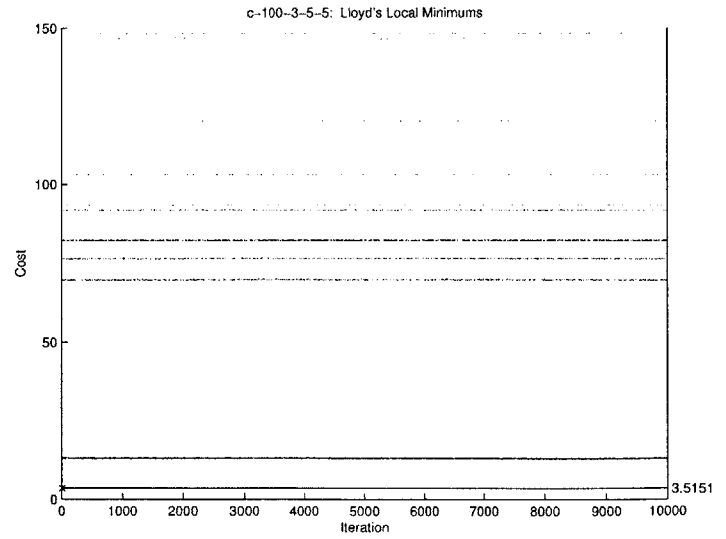


Figure D-3: Costs of Local Minima from Lloyd's Algorithm in 10000 Iterations for dataset c-100-3-5-5

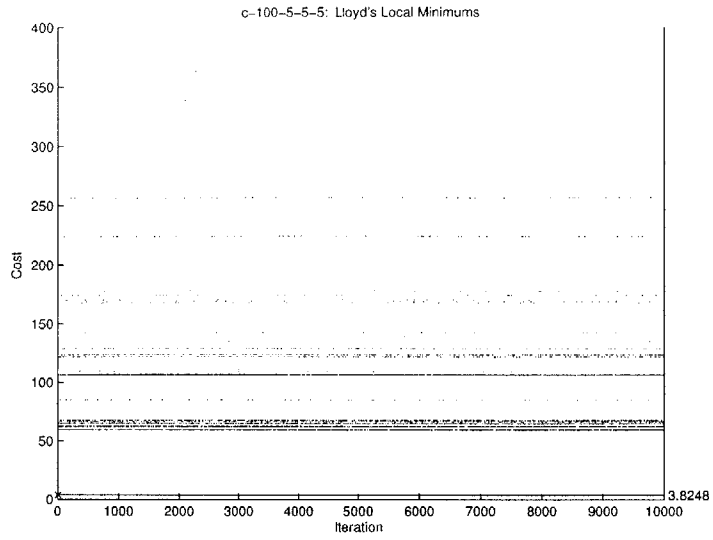


Figure D-4: Costs of Local Minima from Lloyd's Algorithm in 10000 Iterations for dataset c-100-5-5-5

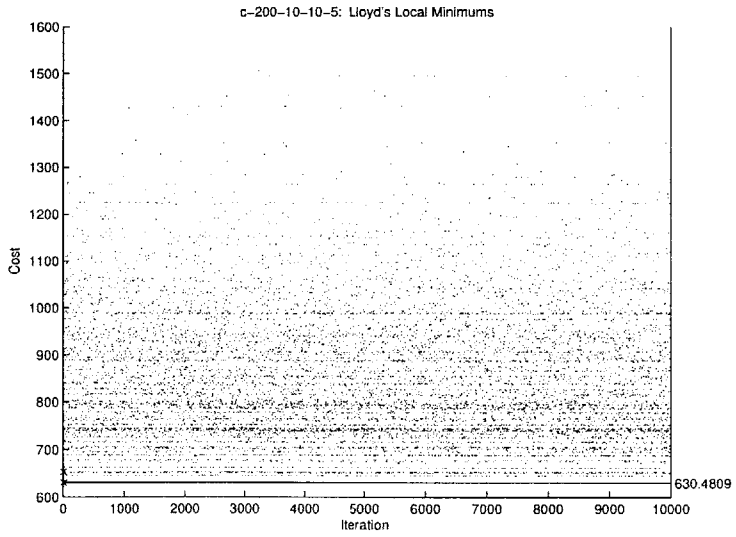


Figure D-5: Costs of Local Minima from Lloyd's Algorithm in 10000 Iterations for dataset c-200-10-10-5

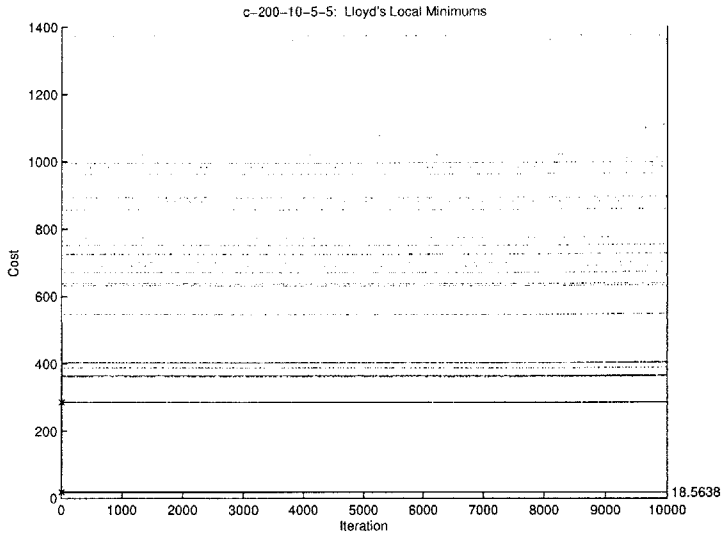


Figure D-6: Costs of Local Minima from Lloyd's Algorithm in 10000 Iterations for dataset c-200-10-5-5

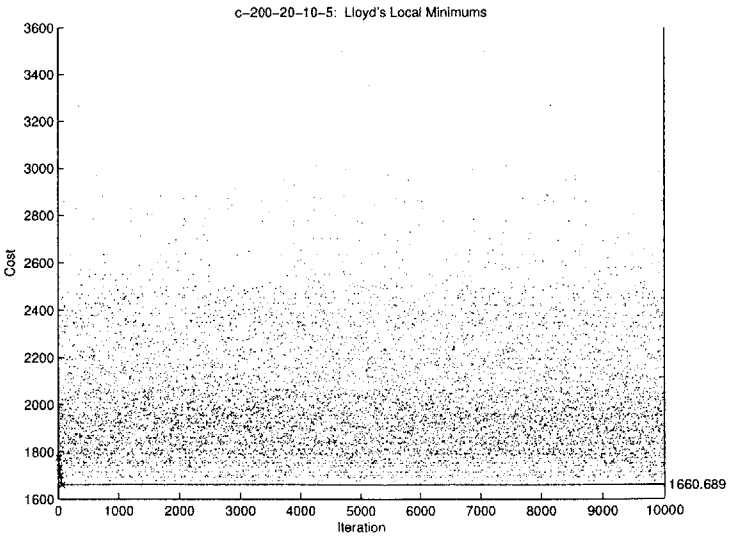


Figure D-7: Costs of Local Minima from Lloyd's Algorithm in 10000 Iterations for dataset c-200-20-10-5

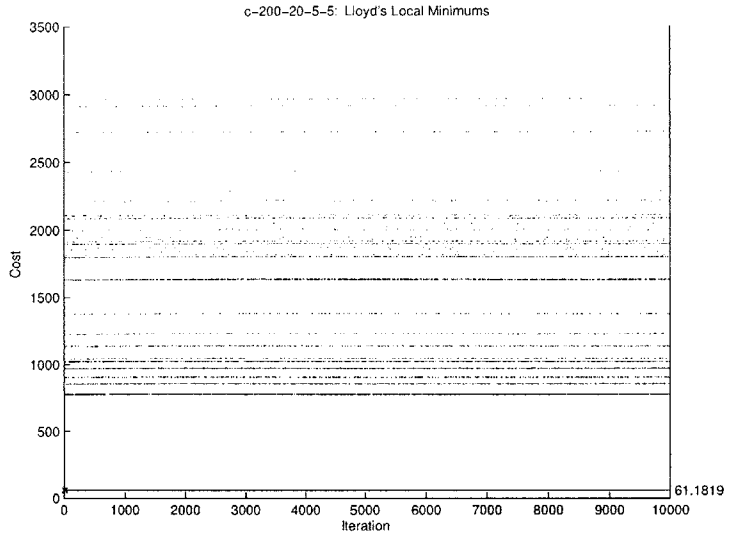


Figure D-8: Costs of Local Minima from Lloyd's Algorithm in 10000 Iterations for dataset c-200-20-5-5

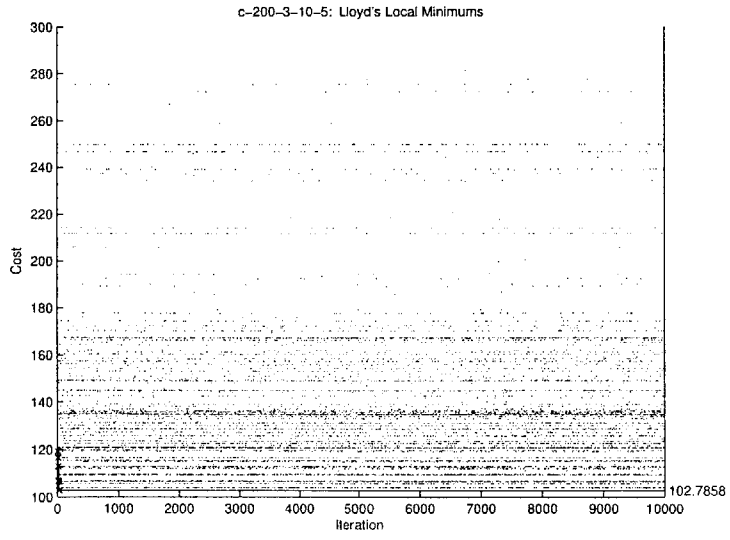


Figure D-9: Costs of Local Minima from Lloyd's Algorithm in 10000 Iterations for dataset c-200-3-10-5

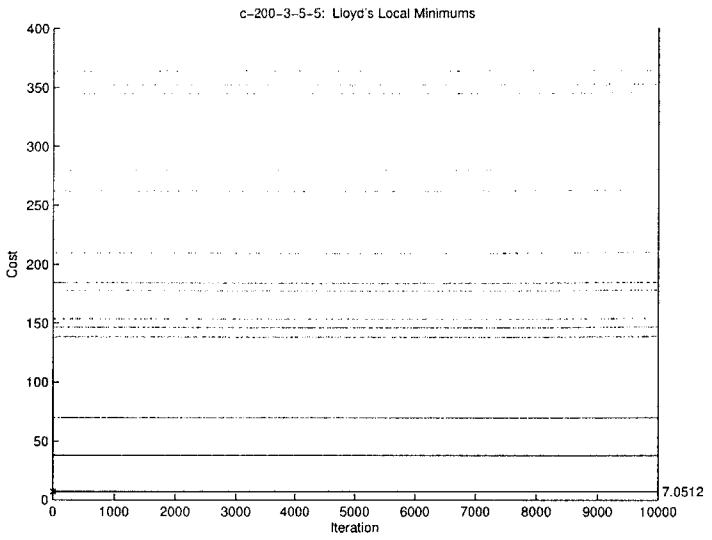


Figure D-10: Costs of Local Minima from Lloyd's Algorithm in 10000 Iterations for dataset c-200-3-5-5

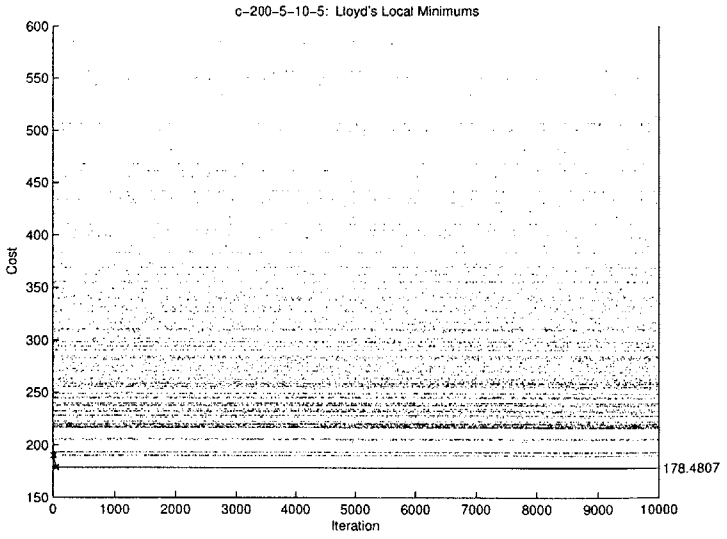


Figure D-11: Costs of Local Minima from Lloyd's Algorithm in 10000 Iterations for dataset c-200-5-10-5

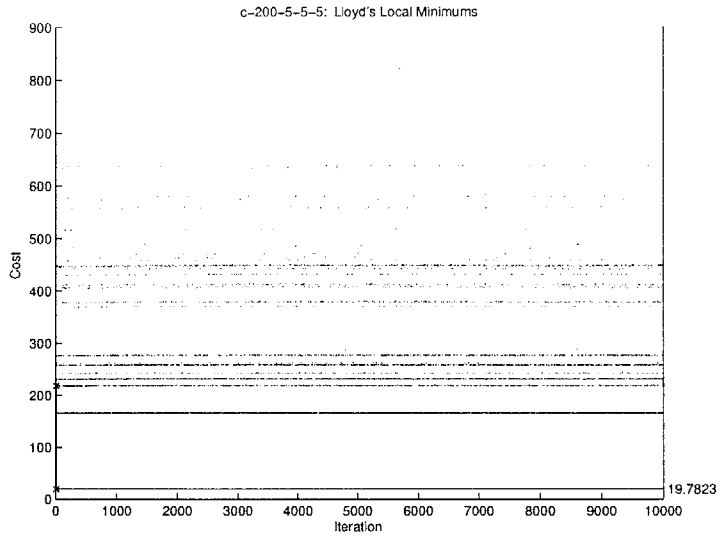


Figure D-12: Costs of Local Minima from Lloyd's Algorithm in 10000 Iterations for dataset c-200-5-5-5

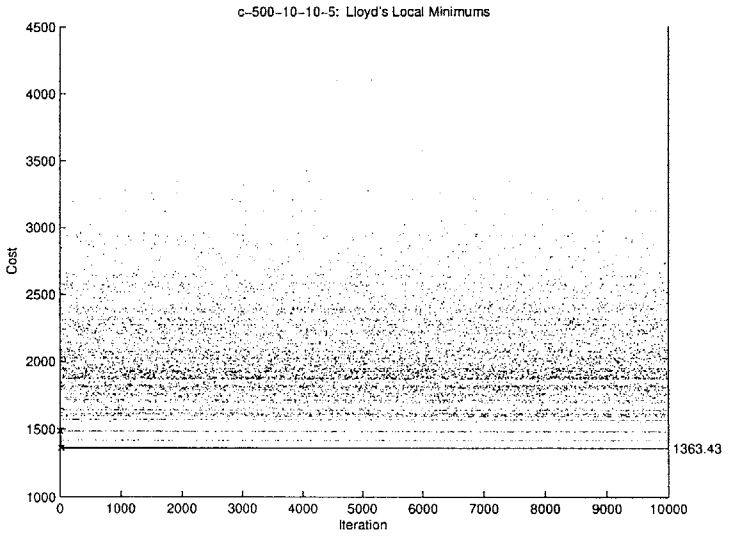


Figure D-13: Costs of Local Minima from Lloyd's Algorithm in 10000 Iterations for dataset c-500-10-10-5

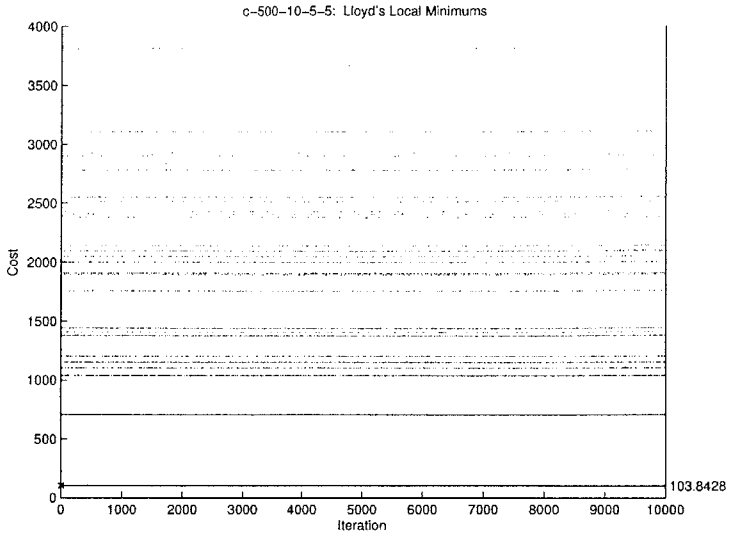


Figure D-14: Costs of Local Minima from Lloyd's Algorithm in 10000 Iterations for dataset c-500-10-5-5

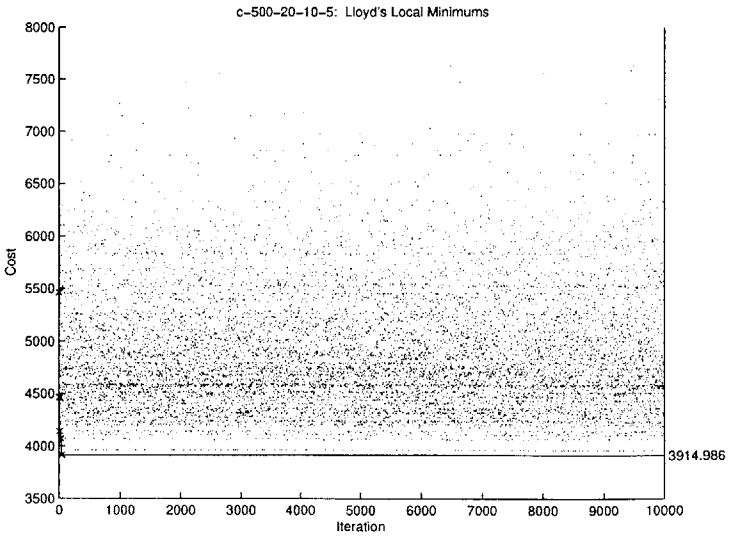


Figure D-15: Costs of Local Minima from Lloyd's Algorithm in 10000 Iterations for dataset c-500-20-10-5

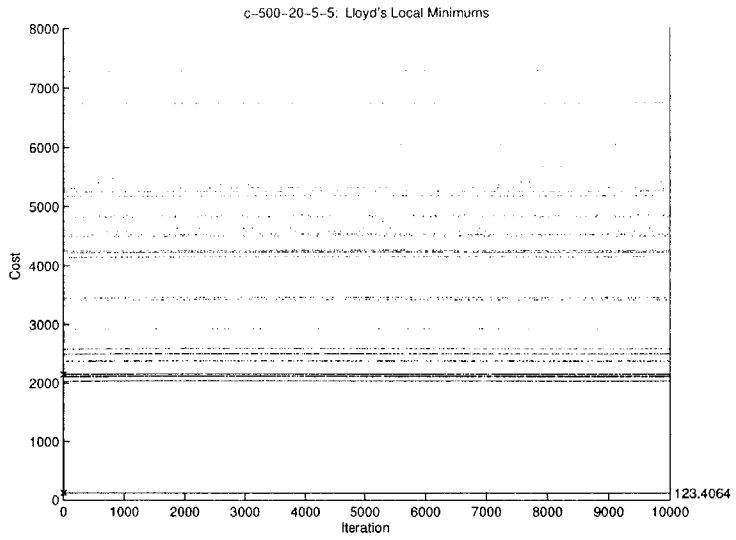


Figure D-16: Costs of Local Minima from Lloyd's Algorithm in 10000 Iterations for dataset c-500-20-5-5

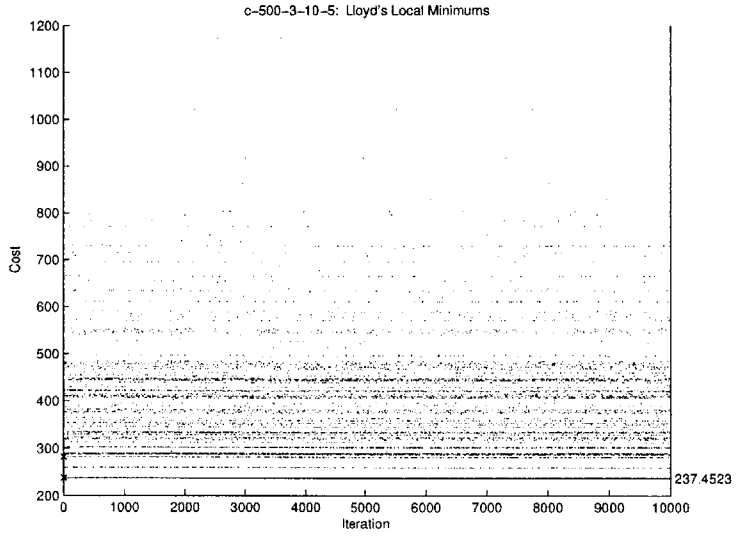


Figure D-17: Costs of Local Minima from Lloyd's Algorithm in 10000 Iterations for dataset c-500-3-10-5

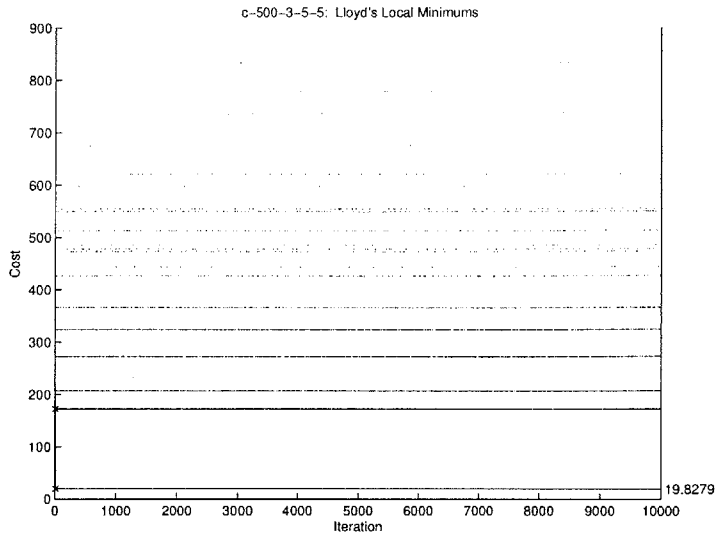


Figure D-18: Costs of Local Minima from Lloyd's Algorithm in 10000 Iterations for dataset c-500-3-5-5

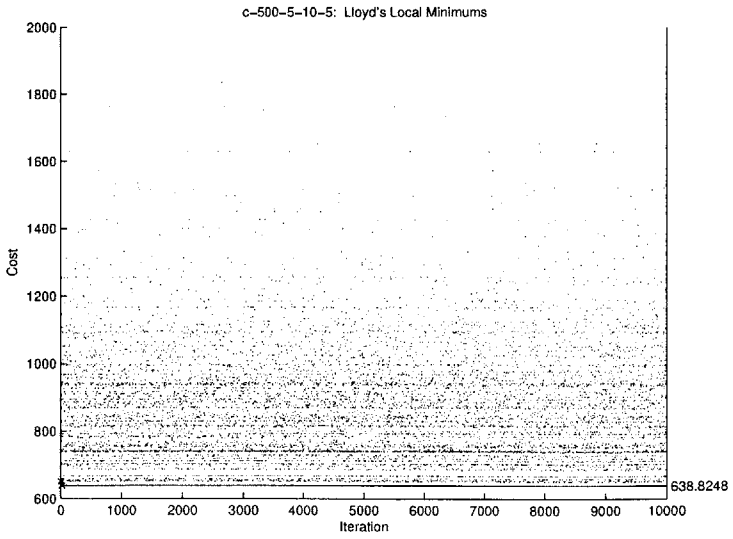


Figure D-19: Costs of Local Minima from Lloyd's Algorithm in 10000 Iterations for dataset c-500-5-10-5

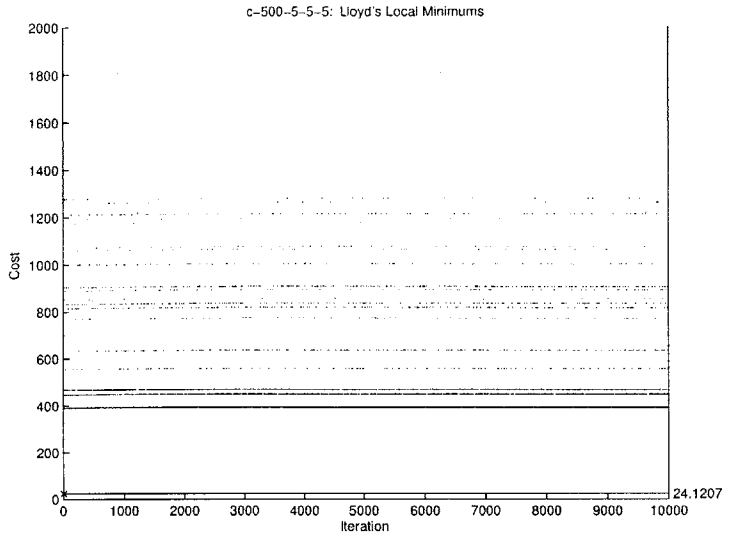


Figure D-20: Costs of Local Minima from Lloyd's Algorithm in 10000 Iterations for dataset c-500-5-5-5

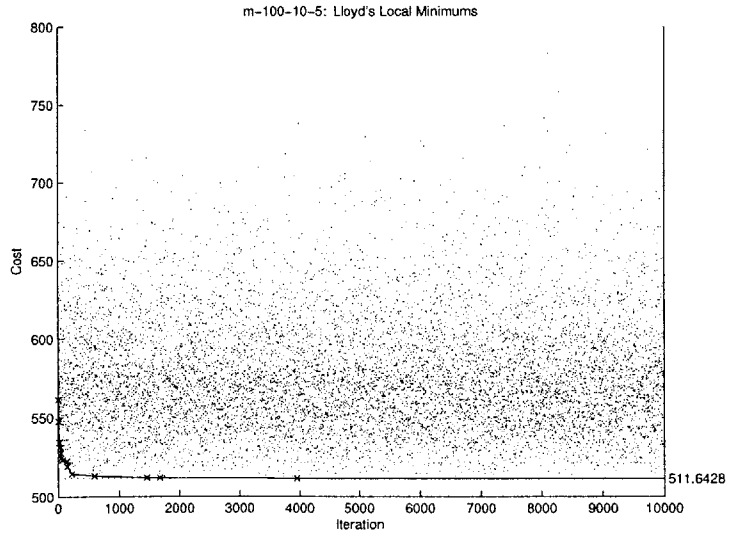


Figure D-21: Costs of Local Minima from Lloyd's Algorithm in 10000 Iterations for dataset m-100-10-5

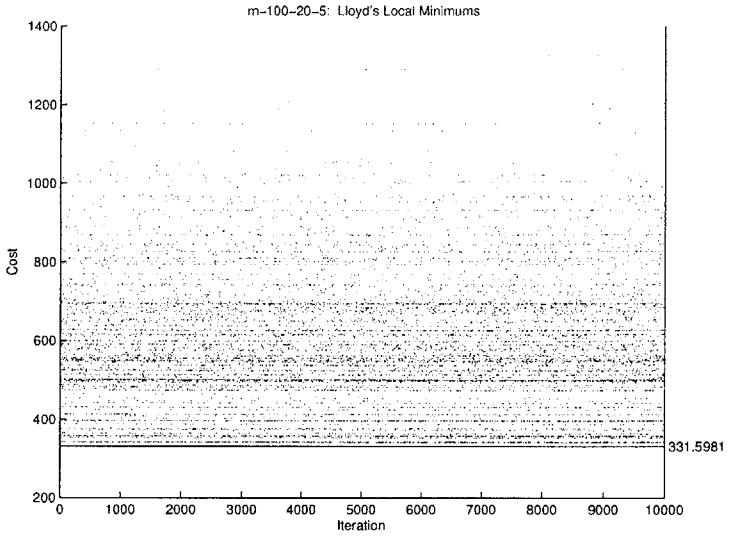


Figure D-22: Costs of Local Minima from Lloyd's Algorithm in 10000 Iterations for dataset m-100-20-5

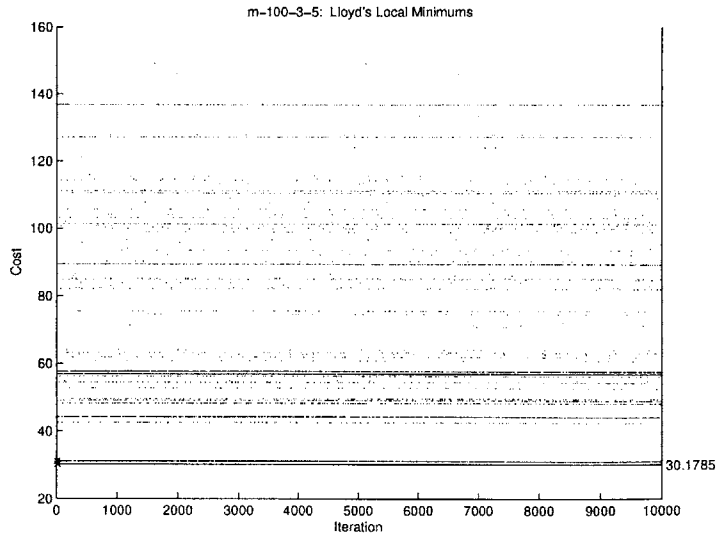


Figure D-23: Costs of Local Minima from Lloyd's Algorithm in 10000 Iterations for dataset m-100-3-5

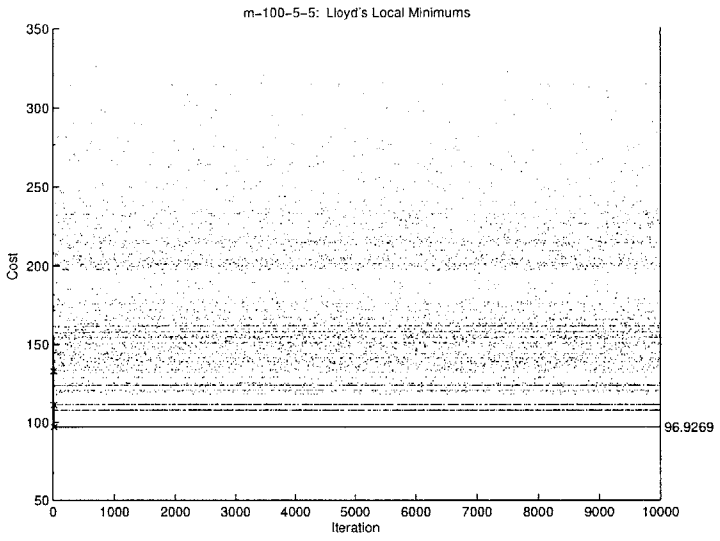


Figure D-24: Costs of Local Minima from Lloyd's Algorithm in 10000 Iterations for dataset m-100-5-5

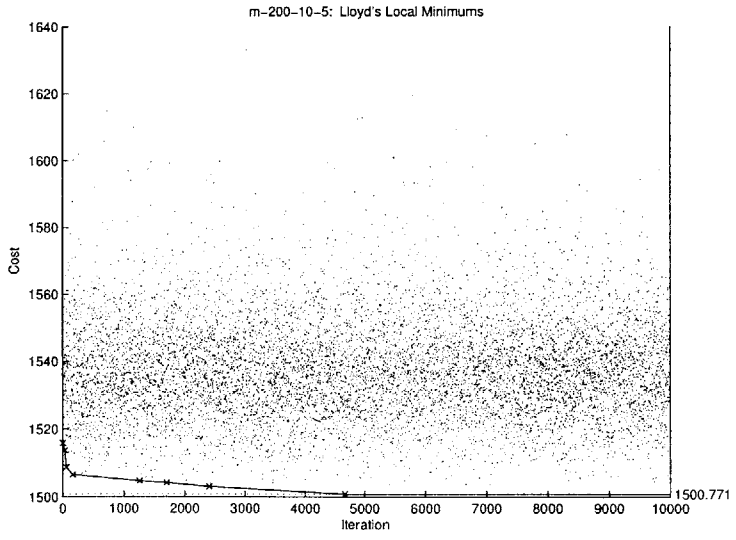


Figure D-25: Costs of Local Minima from Lloyd's Algorithm in 10000 Iterations for dataset m-200-10-5

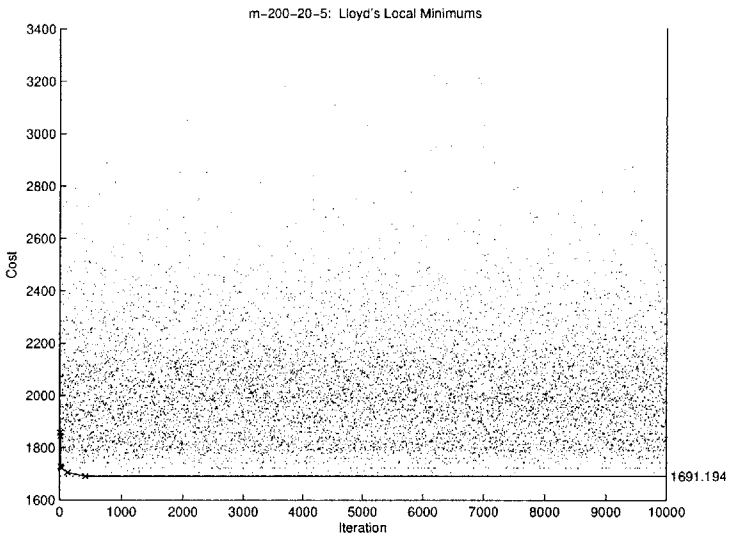


Figure D-26: Costs of Local Minima from Lloyd's Algorithm in 10000 Iterations for dataset m-200-20-5

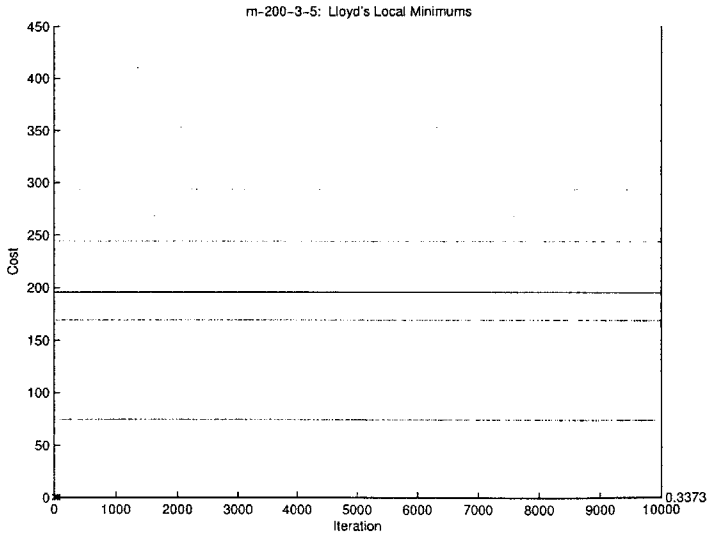


Figure D-27: Costs of Local Minima from Lloyd's Algorithm in 10000 Iterations for dataset m-200-3-5

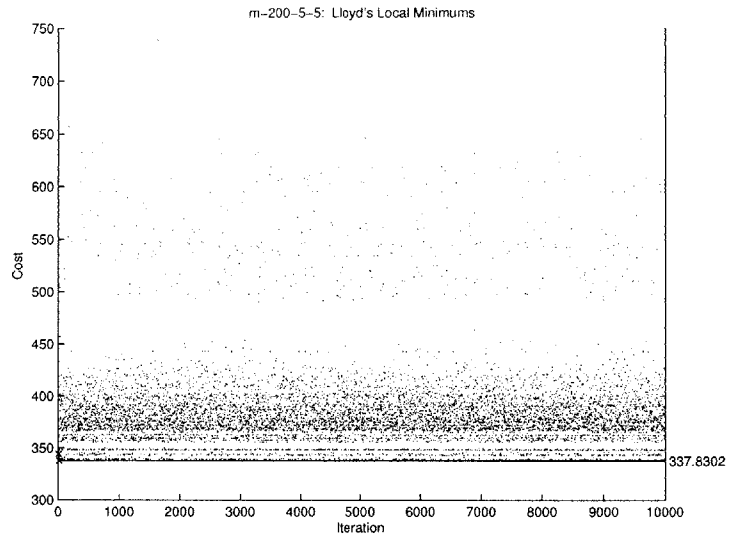


Figure D-28: Costs of Local Minima from Lloyd's Algorithm in 10000 Iterations for dataset m-200-5-5

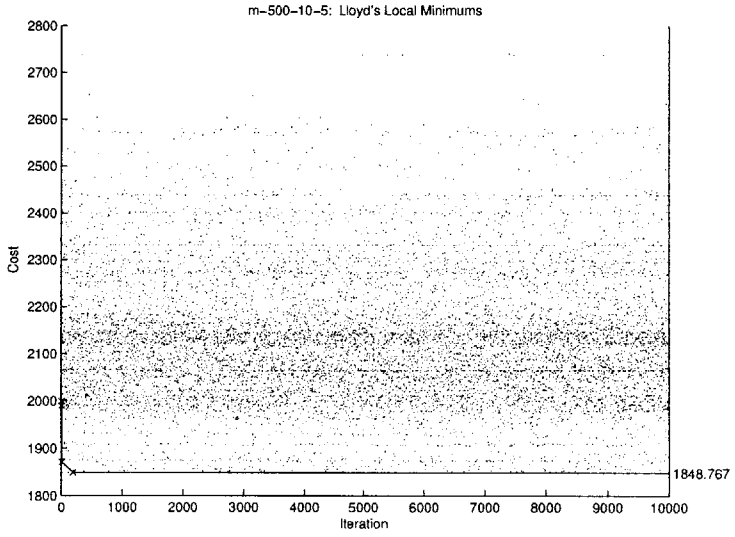


Figure D-29: Costs of Local Minima from Lloyd's Algorithm in 10000 Iterations for dataset m-500-10-5

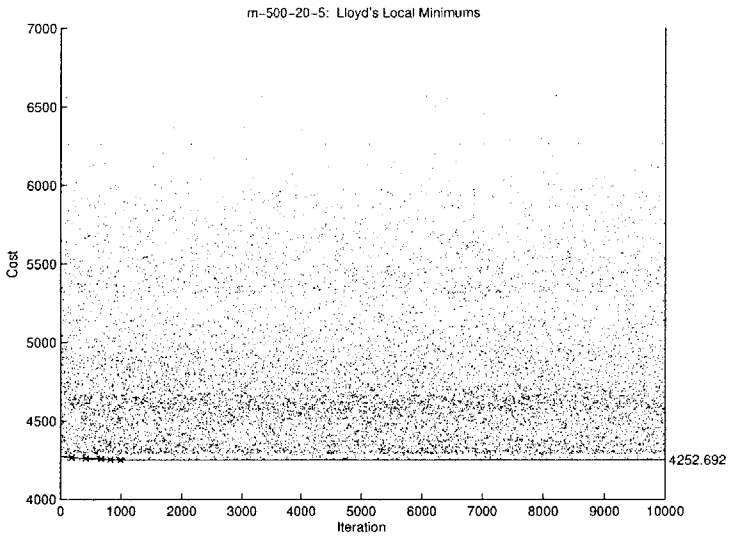


Figure D-30: Costs of Local Minima from Lloyd's Algorithm in 10000 Iterations for dataset m-500-20-5

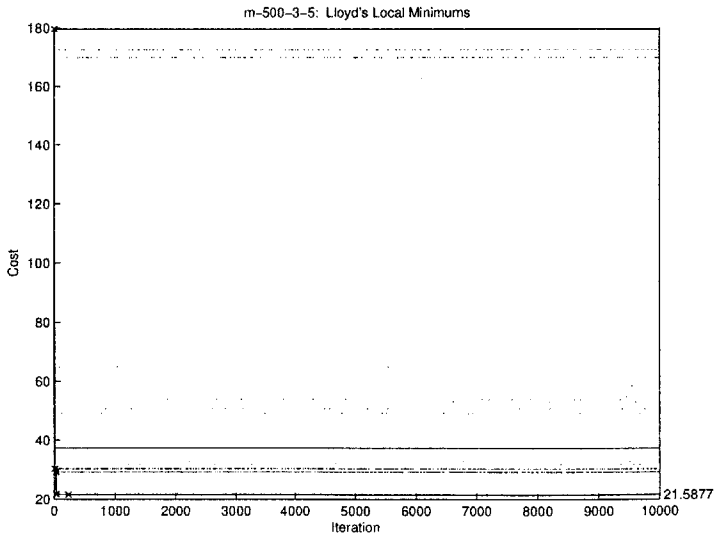


Figure D-31: Costs of Local Minima from Lloyd's Algorithm in 10000 Iterations for dataset m-500-3-5

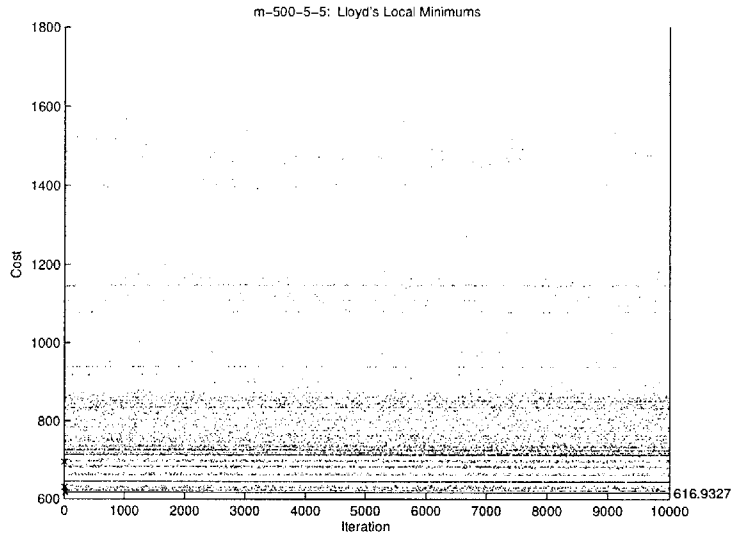


Figure D-32: Costs of Local Minima from Lloyd's Algorithm in 10000 Iterations for dataset m-500-5-5

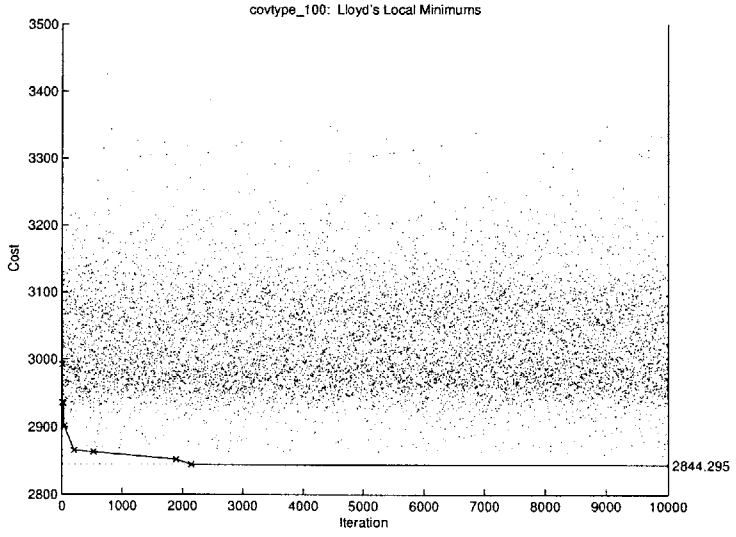


Figure D-33: Costs of Local Minima from Lloyd's Algorithm in 10000 Iterations for dataset covtype-100

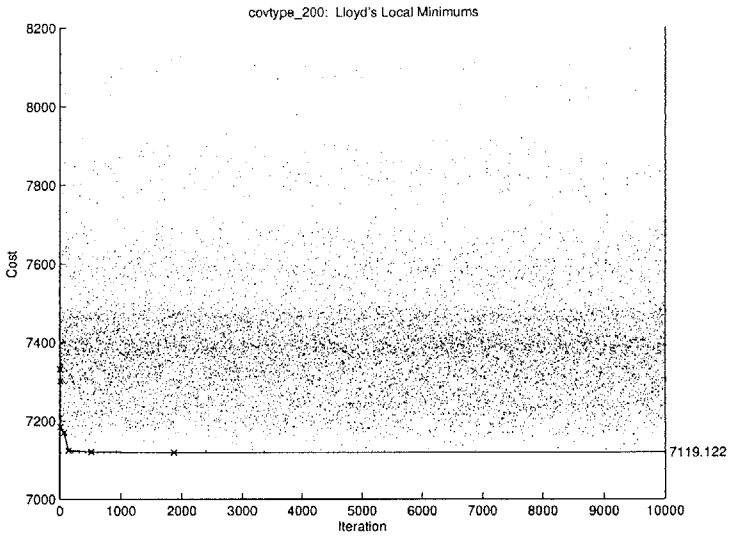


Figure D-34: Costs of Local Minima from Lloyd's Algorithm in 10000 Iterations for dataset covtype-200

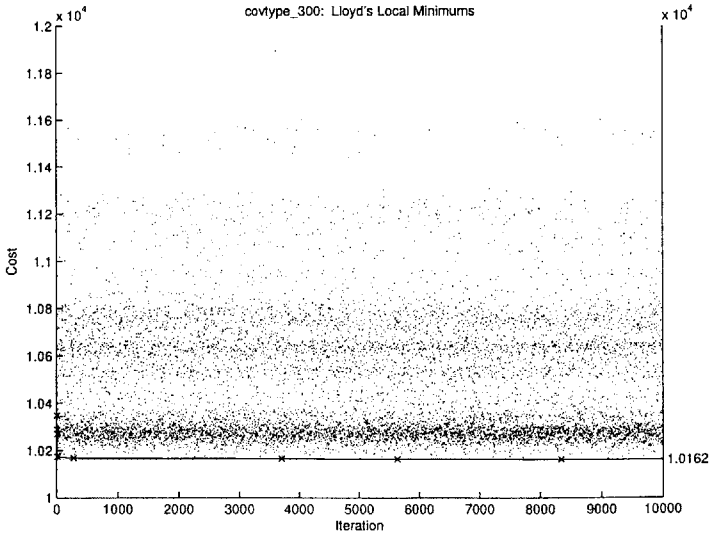


Figure D-35: Costs of Local Minima from Lloyd's Algorithm in 10000 Iterations for dataset covtype-300

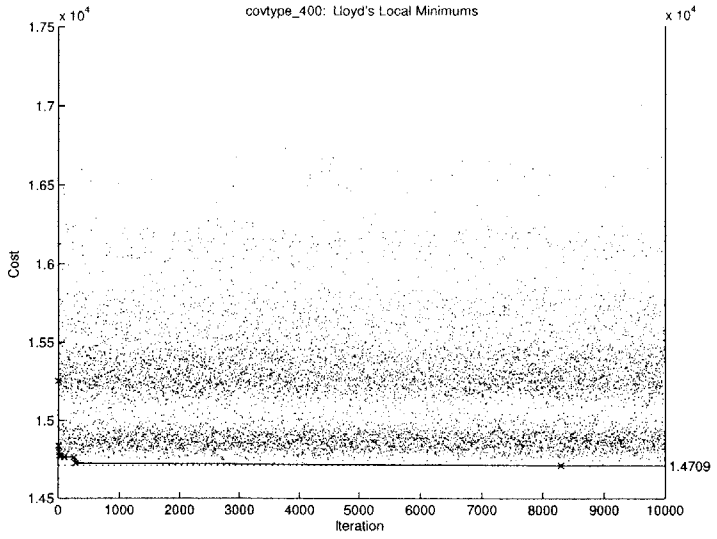


Figure D-36: Costs of Local Minima from Lloyd's Algorithm in 10000 Iterations for dataset covtype-400

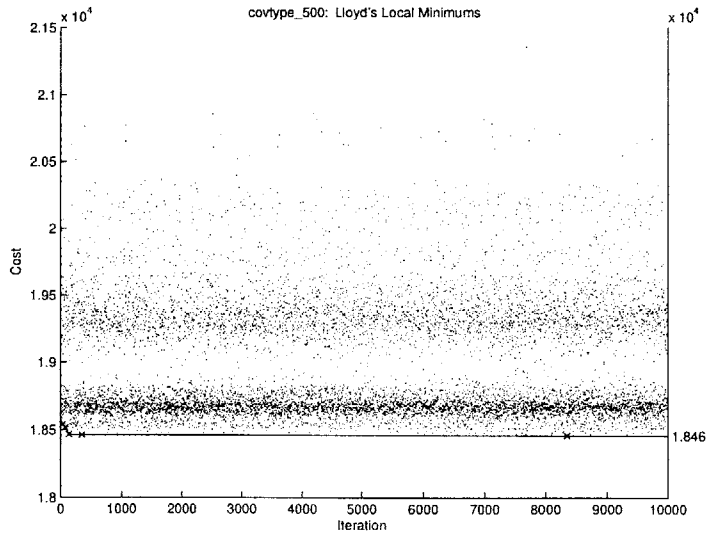


Figure D-37: Costs of Local Minima from Lloyd's Algorithm in 10000 Iterations for dataset covtype-500

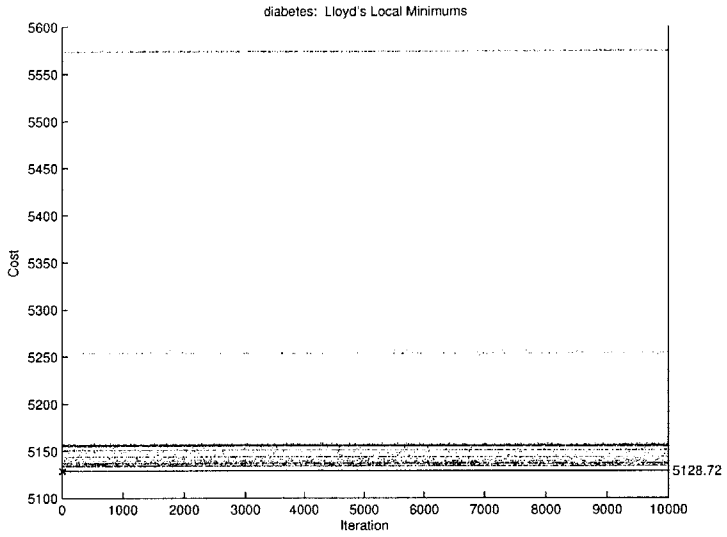


Figure D-38: Costs of Local Minima from Lloyd's Algorithm in 10000 Iterations for dataset diabetes

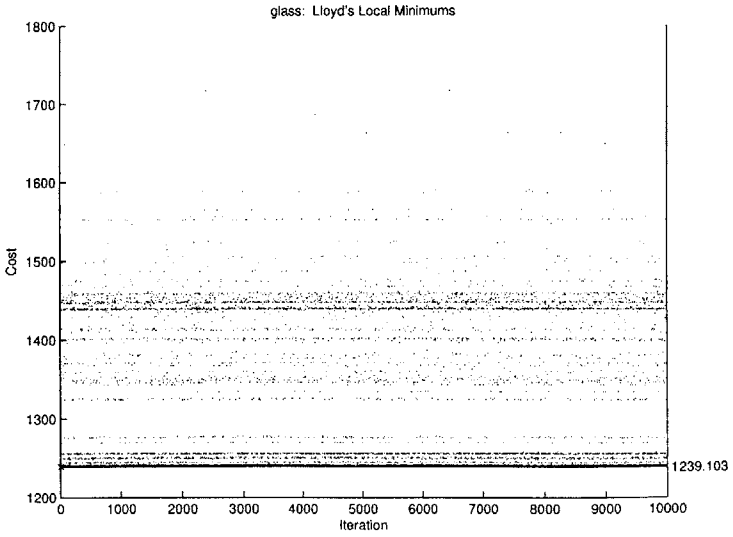


Figure D-39: Costs of Local Minima from Lloyd's Algorithm in 10000 Iterations for dataset glass

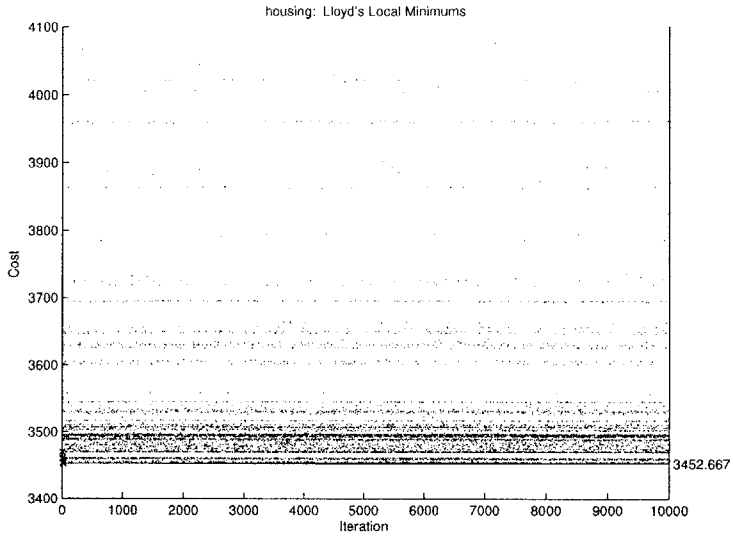


Figure D-40: Costs of Local Minima from Lloyd's Algorithm in 10000 Iterations for dataset housing

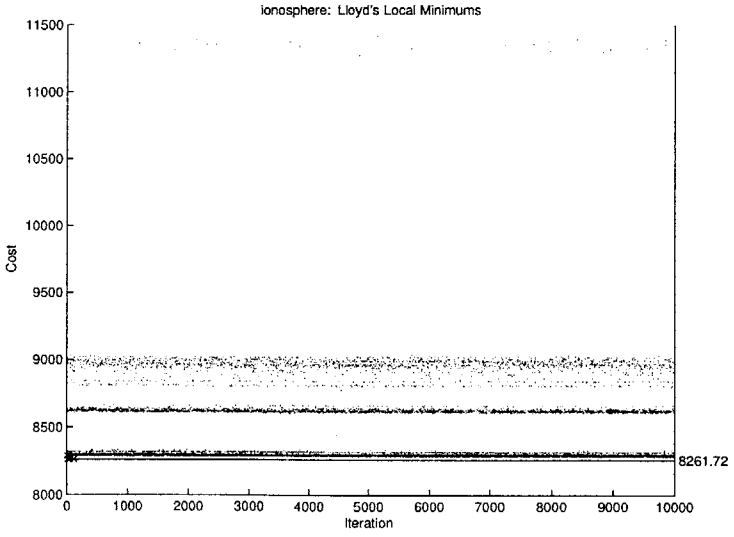


Figure D-41: Costs of Local Minima from Lloyd's Algorithm in 10000 Iterations for dataset ionosphere

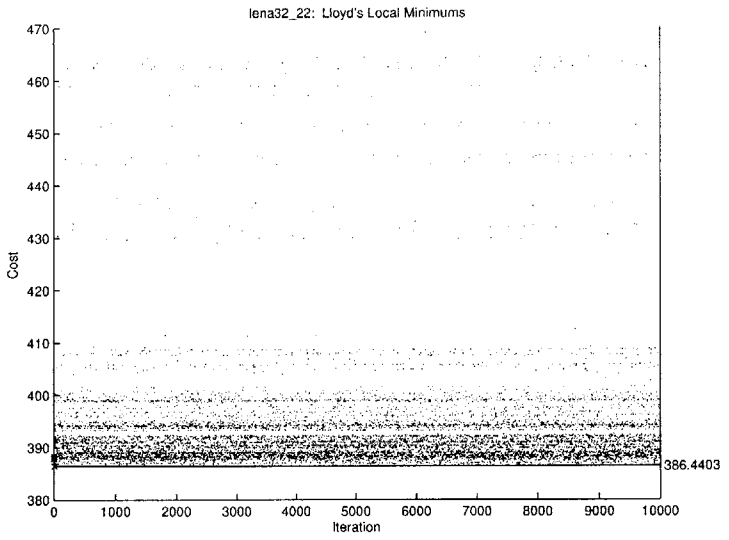


Figure D-42: Costs of Local Minima from Lloyd's Algorithm in 10000 Iterations for dataset lena32-22

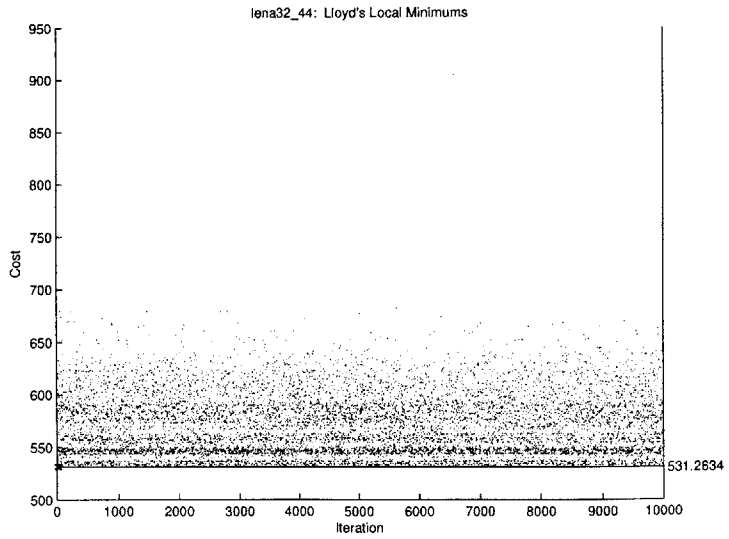


Figure D-43: Costs of Local Minima from Lloyd's Algorithm in 10000 Iterations for dataset lena32-44

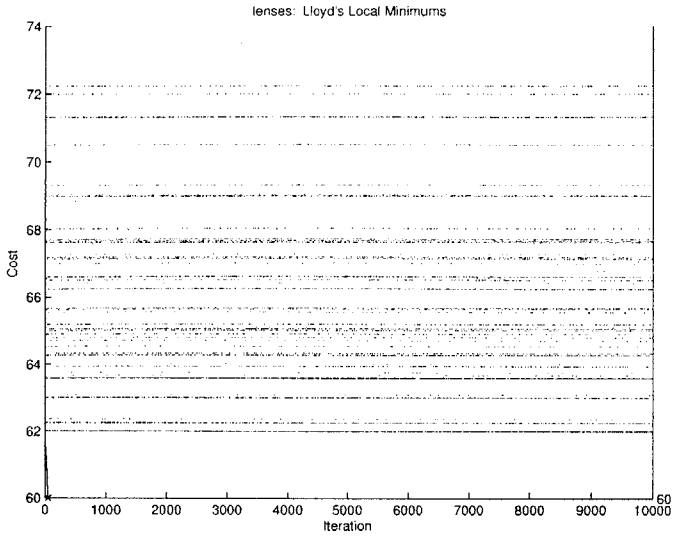


Figure D-44: Costs of Local Minima from Lloyd's Algorithm in 10000 Iterations for dataset lenses

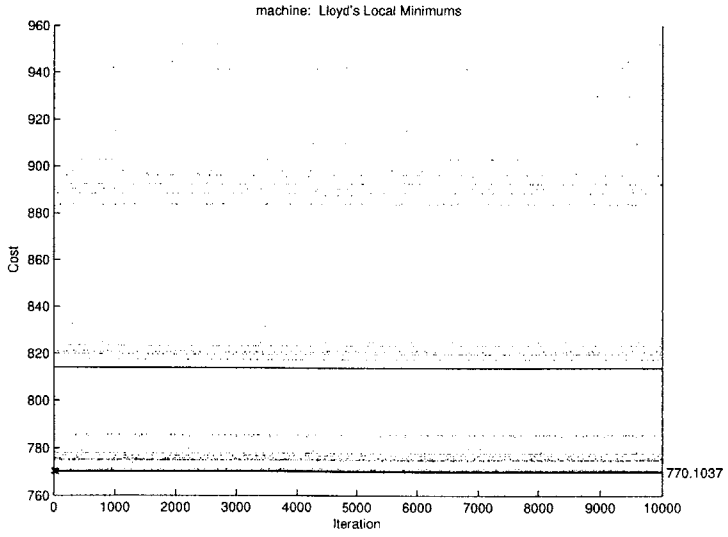


Figure D-45: Costs of Local Minima from Lloyd's Algorithm in 10000 Iterations for dataset machine

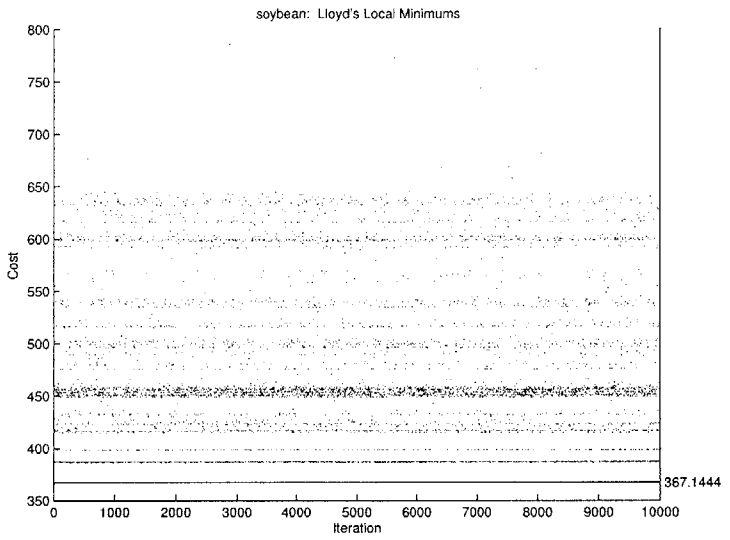


Figure D-46: Costs of Local Minima from Lloyd's Algorithm in 10000 Iterations for dataset soybean

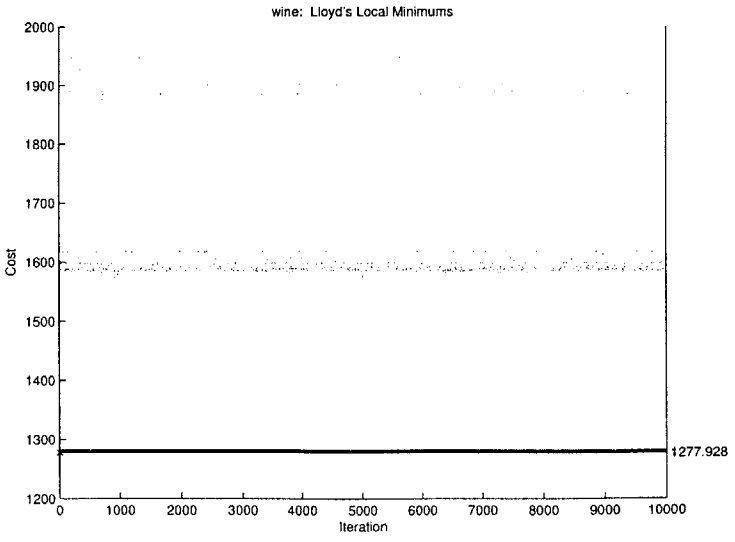


Figure D-47: Costs of Local Minima from Lloyd's Algorithm in 10000 Iterations for dataset wine

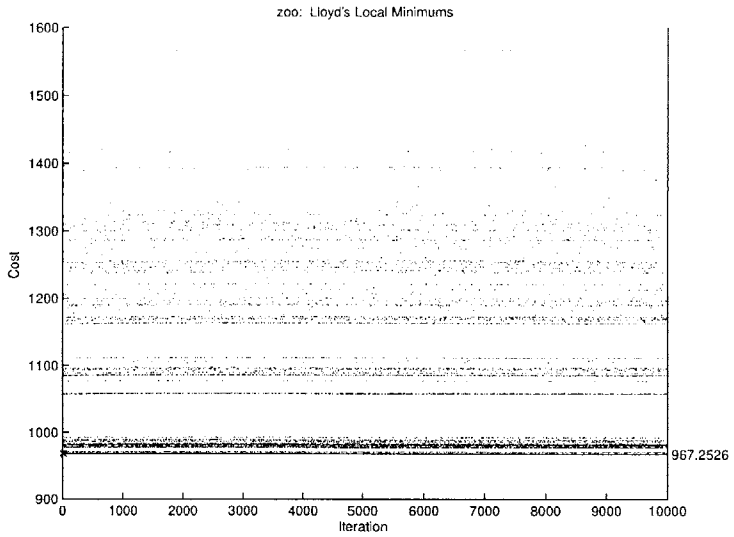


Figure D-48: Costs of Local Minima from Lloyd's Algorithm in 10000 Iterations for dataset zoo

Bibliography

[1] R.K. Ahuja, O. Ergun, and J.B. Orlin. A survey of very large-scale neighborhood search techniques. *Discrete Applied Mathematics*, 123:75–102, 2002.

[2] R.K. Ahuja, J.B. Orlin, and D. Sharma. A composite very large-scale neighborhood structure for the capacitated minimum spanning tree problem. *Operation Research Letter*, 31:185–194, 2003.

[3] C.L. Blake and C.J. Merz. UCI repository of machine learning databases, 1998. University of California, Irvine, Department of Information and Computer Sciences, http://www.ics.uci.edu/~mlearn/MLRepository.html.

[4] Clustan, Ltd. Clustangraphics6. http://www.clustan.com.

[5] E. Diday. The symbolic approach in clustering. In H. H. Bock, editor, *Classification and Related Methods.* North-Holland Publishing Co., 1988.

[6] S. Hettich and S.D. Bay. The UCI KDD archive, 1999. University of California, Irvine, Department of Information and Computer Sciences, http://kdd.ics.uci.edu.

[7] A.K. Jain, M.N. Murty, and P.J. Flynn. Data clustering: A review. *ACM Computing Surveys*, 31(3), September 1999.

[8] T. Kanungo, D.M. Mount, N.S. Netanyahu, C.D. Piatko, R. Silverman, and A.Y. Wu. An efficient k-means clustering algorithm: Analysis and implementation. *IEEE Transaction on Pattern Analysis and Machine Intelligence*, 24(7):881–892, July 2002.

[9] T. Kanungo, D.M. Mount, N.S. Netanyahu, C.D. Piatko, R. Silverman, and A.Y. Wu. A local search approximation algorithm for k-means clustering. In *Proceedings of the Eighteenth Annual Symposium on Computational Geometry*, pages 10–18. Annual Symposium on Computational Geometry, 2002.

[10] A. Likas, N. Vlassis, and J.J. Verbeek. The global k-means clutering algorithm. Technical report, Computer Science Insitute, University of Amsterdam, February 2001.

[11] S.P. Lloyd. Least squares quantization in PCM. *IEEE Transaction on Information Theory*, 28(2):129–137, March 1982.

[12] J.M. Pena and P.J. Larranaga. An empirical comparison of four initialization methods for the k-means algorithm. *Pattern Recognition Letters*, 20:1027–1040, 1999.

[13] P.M. Thompson and J.B. Orlin. The theory of cyclic transfers. Working paper, Operation Research Center, Massachusetts Institute of Technology, August 1989.

[14] T. Zhang, R. Ramakrishnan, and M. Livny. Birch: an efficient data clustering method for very large databases. In *Proceedings of the 1996 ACM SIGMOD international conference on Management of data*, pages 103–114. ACM Press, 1996.