I, Ahmed Mohamed Khedr, hereby submit this as part of the requirements for the degree of:

Doctorate Of Philosophy (Ph.D.)

in:

Computer Science

It is entitled:

Design of Decomposable Algorithms for Distributed Databases

Approved by:

Raj Bhatnagar
John Schlipf
Kenneth Berman
Ali Minai
Lin Liu
Shaaban Abdallah
Design of Decomposable Algorithms
for Distributed Databases

A dissertation submitted to the
Division of Research and Advanced Studies of the University of Cincinnati
in the partial fulfillment of the
requirements for the degree of
Doctor of Philosophy (Ph.D.)
in the Department of
Electrical and Computer Engineering and Computer Science
of the College of Engineering
2003

By
Ahmed Mohamed Khedr
B.S., Zagazig University, Egypt, 1989
M.Sc., Zagazig University, Egypt, 1994

Dissertation Advisor and Committee Chair: Dr. Raj Bhatnagar
Abstract

Design of Decomposable Algorithms for Distributed Databases

Ahmed Mohamed Khedr

Most computer algorithms have been designed for situations in which all relevant data is stored at a single computer site. This is the classical model of a computer based information and control system. The emerging networked knowledge environment requires a significant move away from this classical model. In these situations of geographically distributed but networked systems, the data relevant for a computation may exist in a number of different databases residing at different network sites. An efficient system for computations with such distributed data would work by doing as much work at local sites as possible and then communicating minimum required information among the sites. This is much more efficient than transferring the complete databases to a single site, Join these databases, and then execute algorithms with this data. A common constraint in these situations is that the databases cannot be moved to other network sites due to data-security, size, privacy or data-ownership considerations. Also, for some huge databases it may not be feasible to store and compute with them at one computer site. A number of partitions of this database may be stored at different sites and a set of cooperative algorithms run across the network that produce exactly the same results that would have been obtained if the database were processed at some single site.
In this dissertation we present the results of development, validation, implementation and complexity analysis of the decomposable versions of a number of algorithms. Specifically, decomposable algorithms for following tasks have been investigated: finding a path between two vertices in a directed graph stored as components across various sites of a networks, computing the shortest path spanning tree and the minimum spanning tree for a graph stored as components across various sites of a networks, non-hierarchical clustering in horizontally and vertically partitioned datasets across a number of geographically distributed databases.

The main objective of our algorithms is their self-decomposability for any way in which the graph or the data may be distributed across the sites of a network. Another objective is to perform the tasks by minimizing the number of messages exchanged among the participating sites.

Analytical and empirical results contained in the dissertation show that our decomposable algorithms can obtain either exactly the same results or very close to results that might have been obtained if we had moved all the data to one site and then run the traditional algorithms.
Acknowledgments

As is usual with my work, there are a lot of people who make it possible with influences either direct or indirect. I wish to thank all the people who made this possible.

Unquestionably, I am grateful for the continual support and encouragement from my advisor, Prof. Raj Bhatanagar for providing me with the interesting topic, guiding me through it and encourage me to try my ideas. I would also like to thank him for the patience he shows towards his students, including myself. Although he had other projects in progress, he always had time to give me advice and ideas.

I would like to thank Prof. John Schlipf for accepting to serve on my dissertation committee and for his valuable feedback on my work.

I would like to thank pros. Kenneth Berman, Lin Liu, Ali Minai, and Shaaban Abdallah for taking the time out of their busy lives and accepting to serve on my dissertation committee.

Deep regards to my wife Mrs Abeer Khedr for much care and help during my college years, for her I will always be grateful and also I cannot forget my daughter Salma Khedr who made our life beautiful.

Finally I would like to dedicate this work to my parents and my teacher Imam Hasen Elbana who remain a source of inspiration to me always.
Contents

1 Introduction .................................................. 1

1.1 Graphs ...................................................... 2

1.1.1 Search and Traverse a Graph ............................. 3

1.1.2 Shortest Path in Weighed Graphs and Digraphs .......... 4

1.1.3 Minimum Cost Spanning Tree ............................ 4

1.2 Pattern Recognition ......................................... 5

1.2.1 An Overview .............................................. 7

1.2.2 Unsupervised Versus Supervised Learning ................. 7

1.3 Problem Statement .......................................... 10

1.4 Approaches for Handling Distributed Data .................... 13

1.5 Brief Summary of Results and Contributions ................. 15

1.6 Potential Impact of this Research ............................ 17

1.6.1 Algorithms Design ....................................... 17

1.6.2 Applications ............................................. 18

1.7 Organization of the Dissertation ............................ 20

2 Computations in Distributed Environments ...................... 22
2.1 An Abstraction for Networked Data ........................................ 23
2.2 Nature of Data Distribution ................................................... 26
2.3 Explicit versus Implicit Data .................................................. 29
2.4 An Abstraction for Networks Algorithms ................................. 30
2.5 Decomposition Strategies into Network Algorithms .................... 32
2.6 Implementation of Network Algorithms .................................. 33
2.7 Our Cost Models for Complexity Computing ............................. 33

3 Relevant Research ................................................................. 36

3.1 Graphs Relevant Research ...................................................... 38
3.2 Clustering Relevant Research .................................................. 38

4 Decomposable Algorithms for Path Connectivity ....................... 43

4.1 Graph Representation in Distributed Environments ........................ 44
4.2 Low Granularity Algorithm for Path Finding ............................. 47
  4.2.1 Algorithm Outlines .......................................................... 48
  4.2.2 Algorithm Analysis and Comments ...................................... 50
4.3 Higher Granularity Algorithm for Path Finding ........................ 52
  4.3.1 Algorithm Outline ............................................................ 53
  4.3.2 Algorithm Analysis and Comments ...................................... 54
4.4 More Efficient Algorithm for Path Finding ................................ 58
  4.4.1 The Decomposable Algorithm ............................................ 58
  4.4.2 Example ................................................................. 61
  4.4.3 Algorithm Correctness ..................................................... 63
4.4.4 Algorithm Complexity and Comments .......................... 65
4.5 Conclusions ................................................................. 68

5 Decomposable Algorithms for Computing MST .................. 70
  5.1 Decomposable Algorithm for Computing Shortest Paths Spanning Tree ... 72
    5.1.1 Algorithm Analysis and Comments ............................. 74
  5.2 Higher Granularity Algorithm for Computing MST ................. 75
    5.2.1 Fragment Definitions and Properties ........................... 76
    5.2.2 The Decomposable Algorithm ................................. 77
    5.2.3 Algorithm Correctness ......................................... 80
    5.2.4 Example .......................................................... 82
    5.2.5 Algorithm Complexity and Analysis ......................... 84
  5.3 Conclusions ............................................................. 86

6 Decomposable Clustering Algorithm - Horizontal Partition .... 87
  6.1 Clustering Algorithms and Distributed Databases ............... 88
  6.2 The Decomposable Algorithm ...................................... 90
    6.2.1 Algorithm Outline ............................................. 91
    6.2.2 Example ........................................................ 94
    6.2.3 Consequences of Using Radius and Population .............. 97
    6.2.4 Algorithm Analysis and Comments ............................ 99
  6.3 Conclusion ............................................................ 104

7 Decomposable Clustering Algorithm - Vertical Partition ....... 105
7.1 Decomposition of the k-Means Algorithm .................................. 106
  7.1.1 Decomposable Algorithm ............................................. 107
  7.1.2 Example # 1 ......................................................... 120
  7.1.3 Example # 2 ......................................................... 123
  7.1.4 Analysis and Comments ............................................. 128
  7.1.5 Complexity ........................................................ 129
7.2 Advantages of Decomposable Algorithm .................................... 131
7.3 Conclusion ........................................................................ 133

8 Conclusion and Future Work .................................................. 134
  8.1 Conclusion ................................................................. 134
  8.2 Future Research ........................................................ 137
List of Figures

1.1 Computations in Explicit vs. Implicit Data Spaces ......................... 11

2.1 Distributed Data/Knowledge Sources ........................................... 24

2.2 Horizontally Partitioned Datasets .............................................. 26

2.3 Vertically Partitioned Datasets .................................................. 27

2.4 Data Abstraction ........................................................................ 29

2.5 Computations in Explicit vs. Implicit Data Spaces ......................... 31

4.1 Complete Graph ........................................................................ 44

4.2 Two SubGraphs ......................................................................... 45

4.3 Subgraphs at four sites ............................................................... 62

5.1 Local Fragments at Different Sites ................................................. 82

6.1 A Synthesized Global Strategy ...................................................... 89

6.2 Data at Site-1 ............................................................................. 95

6.3 Data at Site-2 ............................................................................. 96

6.4 Data at Site-3 ............................................................................. 97

6.5 Data points at all three sites ....................................................... 98
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.6</td>
<td>The Local Cluster Centers</td>
<td>99</td>
</tr>
<tr>
<td>6.7</td>
<td>Clustering Using k-means Algorithms</td>
<td>100</td>
</tr>
<tr>
<td>6.8</td>
<td>Clustering Using population Weighted Algorithm</td>
<td>101</td>
</tr>
<tr>
<td>6.9</td>
<td>If Point u has a higher populations and higher radius</td>
<td>102</td>
</tr>
<tr>
<td>7.1</td>
<td>Database at Local Site 1</td>
<td>112</td>
</tr>
<tr>
<td>7.2</td>
<td>Database at Local Site 2</td>
<td>112</td>
</tr>
<tr>
<td>7.3</td>
<td>The Cross product of the Local Databases</td>
<td>113</td>
</tr>
<tr>
<td>7.4</td>
<td>The Cross Product Database after excluding the Exception Data</td>
<td>113</td>
</tr>
<tr>
<td>7.5</td>
<td>TotalError versus Number of Clusters in Distributed and traditional algorithms -1</td>
<td>123</td>
</tr>
<tr>
<td>7.6</td>
<td>TotalError versus Number of Clusters in Distributed and traditional algorithms -2</td>
<td>124</td>
</tr>
<tr>
<td>7.7</td>
<td>Dataset at site-1</td>
<td>125</td>
</tr>
<tr>
<td>7.8</td>
<td>Dataset at site-2</td>
<td>125</td>
</tr>
<tr>
<td>7.9</td>
<td>Dataset at site-3</td>
<td>126</td>
</tr>
<tr>
<td>7.10</td>
<td>Data points at all three sites</td>
<td>126</td>
</tr>
<tr>
<td>7.11</td>
<td>The Local Cluster Centers</td>
<td>127</td>
</tr>
<tr>
<td>7.12</td>
<td>Clustering Using k-means Algorithms</td>
<td>127</td>
</tr>
<tr>
<td>7.13</td>
<td>Clustering Using population Weighted Algorithm</td>
<td>128</td>
</tr>
</tbody>
</table>
List of Tables

4.1 Database at Site-1 ......................................................... 46
4.2 Database at Site-2 ......................................................... 47
4.3 Initialization Step ......................................................... 62
4.4 First Iteration Results .................................................... 63
4.5 Second Iteration Results .................................................. 63

5.1 Summary of the Existing MST Algorithms ............................... 71
5.2 Link Table Stage 1 ........................................................... 83
5.3 Link Table Stage 2 ........................................................... 83
5.4 Link Table Stage 3 ........................................................... 83

6.1 Example of the Database Exists at the Coordinating Site ............. 96

7.1 Example of the Database Exists at one Site ............................. 127
Chapter 1

Introduction

Most network-based algorithms have been designed for environments in which all the relevant data is stored at a single computer site. In the emerging networked knowledge environment computational tasks are encountering situations in which the relevant dataset exists in the form of a number of geographically distributed databases that are connected by wide-area communication networks. Computations involving such distributed data can be performed in two different ways: either by moving and accumulating the data at some common site and then running the algorithm, or by leaving the data at its local site and decomposing the computation into localized subcomputations matching the data present at each site. However, a common constraint in this situation is that the databases cannot be moved to other network sites due to security, size, privacy or data-ownership considerations. The research in this dissertation addresses the problem of decomposing algorithms so that data can stay at their respective locations and sub-computation requests and their results are shared among the participating network sites.

The formulation of our problem, therefore, is the design of algorithms that can decompose
themselves into partial computations to match the distribution of data across network locations, port and execute these partial computations at the participating sites, and compose the received partial results to construct global results.

We have addressed some path finding, shortest paths spanning tree, and minimum spanning tree construction algorithms and some clustering algorithms for designing their decomposable versions.

1.1 Graphs

Graphs play significant role in modeling parallel architectures and algorithms since they serve as models for interconnection networks and many other networks of paths [18, 84, 30]. Many algorithms for the following simple problems have been designed and applied in the context of graphs as abstractions:

- Finding paths, or the shortest paths, among vertices of a network of roads, communication links, or other pathways. Examples of these algorithms include Dijkstra’s algorithm, Floyd’s algorithm and the Bellman Ford algorithm for finding minimum cost paths in a graph.

- Finding minimum spanning tree to connect vertices of a graph.

- Finding optimal flows across a network of pathways.

- Determination of flows and flow patterns in geographic information systems (GIS).

Formally, a graph $G (V, E)$ is a set $V = V(G)$ called vertices, together with a set $E = E(G)$ called edges, such that each edge is an unordered pair of vertices. Two vertices $u$ and $v$ are
called adjacent if they are the two end vertices of an edge $e$ in the graph (that is $uv \in E$).

The set of all vertices adjacent to $u$ is the neighborhood of $u$.

**Digraphs** are graphs with directed edges. Digraphs are natural models for many physical phenomena. For example, the pairings of the players in a (match-play golf, table tennis, etc.)

Formally, a digraph is a set of vertices $V = V(G)$ together with a set $E = E(G)$ called directed edges such that a directed edge is an order pair of vertices.

### 1.1.1 Search and Traverse a Graph

The solution to many important problems requires an examination (visit) of the vertices of the graph. Two standard search techniques are *depth-first search* and *breadth-first search*. Depth-first search and breadth-first search differ in their exploiting philosophies: Depth-first search always longs to see what is over the next hill, whereas breadth-first search visits the immediate neighborhood thoroughly before moving on. After visiting a vertex, breadth-first search explores this vertex (visits all the neighbors of the vertex that have not already been visited) before moving on. On the other hand, depth-first search immediately moves on to an unvisited neighbor, if one exists, after visiting a vertex. Whenever depth-first search is at an explored vertex, it backtracks until an unexplored vertex is encountered and then continues. This backtracking often returns to the same vertex many times before it is explored.
1.1.2 Shortest Path in Weighed Graphs and Digraphs

Given a vertex $r$ in a graph or in a digraph, breadth-first search yields shortest path from $r$ to every vertex reachable from $r$. In many applications, a weight (cost) $W(uv)$ is associated with each edge $uv$. For example a graph might model a network of cities, and the weight might the distance or driving time between adjacent cities. As another example a digraph might model a network of (direct) airplane flights, and the weight of the flight might be its cost or its flying time. The weight length of a path is the sum of the weights of its edges. There are many algorithms to find the shortest path like Bellman-Ford algorithm, prim algorithm, etc.

1.1.3 Minimum Cost Spanning Tree

Given a graph $G$ finding a spanning tree $T$ of minimum weight in $G$ with respect to a weighting $W$ of the edges is very important in many applications. The weight of $T$, denoted $W(T)$, is the sum of the w-weight of its edges. We call such a tree Minimum Spanning Tree ($MST$). The problem of finding a minimum spanning tree in a weighted graph has a variety of applications. The number of spanning trees, even for a relatively small graphs, is usually enormous, making an enumerative brute-force search infeasible. Fortunately, there are many algorithms for computing a minimum spanning tree. The efficient algorithms are based on greedy method like Kruskal’s, Prim’s, and Dijkstra’s algorithms. These algorithms are similar in the sense that their selection functions always choose an edge of minimum weight among the remaining edges. However, Prim’s algorithm selection function only considers edges incident to edges already included in the tree. In particular, the partial solutions built
by Prim’s algorithm are trees, whereas the partial solutions built by Kruskal’s algorithm are forests. For undirected graphs Dijkstra’s algorithm follows a similar strategy to Prim’s algorithm, differing only in the way the next edge is selected. At each step in Dijkstra, instead of selecting an edge of minimum weight in $\text{Cut}(T)$ where $\text{Cut}(T)$ is the set of all edges in $\mathcal{G}$ having exactly one vertex in $T$, we select an edge $uv \in \text{Cut}(T)$, $u \in V(T)$, so that the path from $r$ to $v$ in the augmented tree $(T \cup uv)$ is shortest, that is such that $\text{Dist}(u) + W(uv)$ is minimized over all the edges $uv \in \text{Cut}(T)$, where $\text{Dist}(u)$ is the length of a path from $r$ to $u$ in $T$.

As a part of this dissertation we present self-decomposing versions of five simple graph algorithms. The first three algorithms are for finding a path in a graph stored in parts on distributed databases. The fourth algorithm is for computing the shortest paths spanning tree to graph stored in parts on distributed databases. The fifth algorithm is for finding the minimum spanning trees to graph stored in distributed databases [60, 63, 64] An objective of our algorithms is their self-decomposability for any way in which the graph may be distributed across the network. Another objective of the algorithms is to perform their tasks by minimizing the number of messages exchanged among the participating sites.

1.2 Pattern Recognition

Pattern Recognition is the scientific discipline whose goal is the classification of objects into a number of categories or classes. Depending on the application, these objects can be images or signal waveforms or any type of measurements that need to be classified. Pattern Recognition has a long history, but before 1960s it was mostly the output of theoretical research in the
area of statistics. As with everything else the advent of computers increased the demands of practical application of Pattern Recognition, which in turn set new further theoretical developments. Pattern Recognition is an integral part in most machine intelligent systems built for decision making. Cover [33] describes the central aim of Pattern Recognition as a systematic theoretical and experimental attempt to develop simple computational means for placing abstract objects into categories, with the eye-brain computer [33] as one model. Pattern Recognition is one of the means by which generalization can come about. There are two approaches to the study of Pattern Recognitions:

- Statistical Pattern Recognition.
- Structural Pattern Recognition.

Statistical Pattern Recognition [37, 43] is a well defined discipline, solidly in mathematical statistics, with an extensive literature. In this approach, features describing an object are treated only as statistical attributes of data without any regard to their meaning.

Structural Pattern Recognition [44, 45] corresponds in a direct sense to the human intuitive notation of what takes place in Pattern Recognition. An object is examined for its characteristic features and their interrelations and then, according to those features, it is assigned to a given class. Early work in this direction [16, 17, 44, 45] aimed to find formal methods of describing shapes by attempting to generalize the concepts of formal languages to two dimensional images. Because of their basis in formal language theory, these methods are also called as linguistic or syntactic Pattern Recognition.
1.2.1 An Overview

The general Pattern Recognition problem can be stated as *Given a collection of objects belonging to a predefined set of classes and a set of measurements on these objects, identify the class of membership of each of these objects by a suitable analysis of the measurements.* Given an object, the process of identifying its membership class can typically be broken down into the following steps: Measurements, Noise filtering and Signal Conditioning Segmentation, Description and Decision. Related to these every Pattern Recognition must also contain the following subsystems: A set of sensors, Analog/Digital Converter, Signal Conditioning module (Pre-processor), Feature Extractor module, Decision module (Classifier), Post-processor and an output module. Sensors and ADC serve as inputs to the system. Pre-processor typically includes measurement and filtering and these precede the algorithmic part of Pattern Recognition. The feature extractor converts the set of measurements to a form useful to the classifier. This output forms the input to the classifier that classifies the input to one of the classes. The output of classifier may be *No Decision* or a *unique decision* or a list of possibilities, perhaps with a confidence weight attached to each of them. The post-processor evaluates these possibilities, also taking into account the environment or context of the unknown object, and then narrows down the list of possibilities to a final decision.

1.2.2 Unsupervised Versus Supervised Learning

If we assumed a set of training data are available, and the classifier was designed by exploiting this a priori known information. This is known as Supervised Pattern Recognition. There
are some distributed algorithms to handle the supervised learning. An example of this is the work by Chan and Stolfo [25] in which they have investigated the idea of multiple learning processes with supervised inductive learning. These methodologies run the learning algorithm in each of these subsets of data, and combine the results by identifying two main meta-learning components: *combiner* and *arbiter*. Both components are independent of the learning algorithms used with individual components in generating the classifiers. The combiner attempts to reveal relationships among the learned classifiers’ prediction patterns and the arbiter tries to determine the correct prediction when the classifiers have different opinions. Various schemes for the *combiner* and *arbiter* have been developed [19, 20, 21, 22, 23, 24].

However, this is not the only case, and there is another type of Pattern Recognition tasks for which training data, of known class labels, are not available. In this type of problem, we are given a set of feature vectors $x$ and the goal is to unravel the underlying similarities, and cluster (group) similar vectors together. This is known as Unsupervised Pattern Recognition or clustering. such tasks arise in many applications in social sciences and engineering, such as remote sensing, image segmentation, and image and speech coding.

A clustering algorithm can be employed to reveal the groups in which feature vectors are clustered in the $l$-dimensional feature space. Points that correspond to the same ground cover type, such as water, are expected to cluster together and forms groups. Once this is done the analyst can identify the type of each cluster by associating a sample of points in each group with available reference ground data, that is, maps or visits. Clustering also is widely used in the social science in order to study or correlate survey and statistical data and draw useful conclusion, which will then lead to the right action.
A major issue in Unsupervised Pattern Recognition is that of defining the "similarity" between two feature vectors and choosing an appropriate measure for it. Another issue of importance is choosing an algorithmic scheme that will cluster (group) the vectors on the basis of the adopted similarity measure. In general, different algorithmic schemes may lead to different results, which the expert has to interpret.

The clustering problem relates to classifying a collection of data points into a set of naturally close clusters without any a priori knowledge. Many clustering algorithms have been proposed by researchers in the past, for example, see [6, 7, 29, 56, 92]. The hierarchical clustering algorithms can be divided into agglomerative and divisive methods. The agglomerative methods merge together the most similar points or sub-clusters at all successively higher levels. In divisive methods, initially the set of all objects is viewed as one cluster and at each level some clusters are divided into smaller clusters. Non-hierarchical methods produce a single clustering. They are quite straightforward and fast methods. In most of them all the feature vectors are presented to the algorithm once, or a few times. These algorithms tend to produce compact and hyper-spherical or hyper-ellipsoidal shaped clusters, depending on the distance metrics used.

In distributed environment there are some algorithms for example Kargupta et al. in [59] have presented an algorithm for clustering high-dimensional heterogeneous data using a distributed principal component analysis (PCA) technique called the collective PCA. In their approach [59] partial results (principle eigenvectors) are computed at each site and transmitted to a central site along with a number of data tuples corresponding to each eigenvector. The error between the actual global result and the computed result decreases as the number of tuples transmitted from each site to the central site increases.
Our proposed work is dealing with unsupervised learning (Clustering). We design two algorithms for clustering dataset exists in a partitioned form (vertically and horizontally), across a number of geographically distributed databases [61, 62]. We seek to decompose the computations of \textit{k-means} algorithm in such a way that no actual data tuples need to be transmitted among the sites and the error is kept low by transmitting intermediate results only. An objective of our algorithms is their self-decomposability for any way in which the data may be distributed across the network. Another objective of the algorithms is to perform their tasks by minimizing the number of messages exchanged among the participating sites.

### 1.3 Problem Statement

Our problems involves adapting and design new graph and clustering algorithms to work with a distributed databases. We operate under a constraint that data cannot be transferred between sites. Let us say a result \textit{Result} is obtained by applying an algorithm \textit{Alg} to a dataset \( \mathcal{D} \) that is:

\[
Result = Alg(\mathcal{D}) \tag{1.1}
\]

In the case of algorithms for distributed databases, \( \mathcal{D} \) cannot be made explicit and is known only implicit in terms of the explicit components \( D_1, D_2, \ldots, D_n \). The implementation of \textit{Alg} in the last equation can be redesigned by a functionally equivalent formulation:

\[
Result = Alg'(f_1(D_1), f_2(D_2), \ldots, f_n(D_n)) \tag{1.2}
\]

That is, a local computation \( f_i(D_i) \) is performed at \textit{Site}_i using the database \( D_i \). The
results of these local computations are aggregated using the operation $Alg'$. We present self-decomposing versions of graph and clustering algorithms, work on which has already been completed by us. The set of $Alg'$ and $f_i$ operators would vary with the $D_i$s and hence a dif-

Figure 1.1: Computations in Explicit vs. Implicit Data Spaces

ferent set would need to be generated for each new instance of $D_i$s that is encountered by the computing agent at a site. A schematic in Figure 1.1 shows the process by which a network algorithm would compute $Alg$ from the $D_i$s. The components operators of a decomposition ($Alg'$ and $f_i$s), therefore, need to be dynamically determined for each instance of $Alg(D)$ depending on the participating sites; the attributes contained in their native databases; and the overlap pattern of their attributes as reflected in the set of shared attributes.

An objective of our algorithms is their self-decomposability for any way in which the data may be distributed across the network. Another objective of the algorithms is to perform
their tasks by minimizing the number of messages exchanged among the participating sites, the following tasks are completed

- Develop decomposed algorithms for finding a path in a graph stored in parts on distributed databases.

- Develop decomposed algorithms to compute shortest paths spanning tree in a graph stored in parts on distributed databases.

- Develop decomposed algorithms to determine a minimum cost spanning tree in a graph stored in parts on distributed databases.

- Develop decomposed algorithms for performing unsupervised learning by clustering.

We design a decomposed version of the algorithms that can handle the following situations:

- Horizontally partitioned datasets, and

- Vertically partitioned datasets.

The central problem we study in this dissertation is succinctly stated as

_We seek a means to improve the efficiency and the accuracy and keep the data security, and the data privacy of graph and clustering algorithms applied to very large amounts of data that exist in a distributed form._
1.4 Approaches for Handling Distributed Data

The main focus of most of the parallel and distributed algorithms has been on closely coupled processor systems where data can be easily shared by the processors and the number of processors assigned to an algorithm may depend on the data to be processed. The parallel algorithms are tailored for situations in which: (i) it is assumed that all data resides in main memory or distributed shared memory of a set of closely connected processors; (ii) they need a large number of closely connected processors at a single site that can access the shared memory to achieve the reasonable performance; and (iii) Inter-processor communication is extremely fast and involves reading data in the main or the shared memory.

Grossman and et.al. describes three modes for mining geographically distributed databases. These are based on transferring (i) results; (ii) models; or (iii) data from distributed sites to a central site. Distributed knowledge discovery (DKD) (Chan and Stolfo; Lam and Segre; Yamanishi ; Crestana and Soparkar; Tumer and Ghosh ; Grossman et. al.; Park et. al.) embraces the growing trend of merging computation with communication and explores all facets of the KDD process in the context of the emerging distributed computing environments. (DKD) accepts the fact that the data may be inherently distributed among different loosely coupled sites connected by a network and the sites may have heterogeneous data. It offers techniques to discover new knowledge through distributed data analysis and modeling using minimal communication of data. DKD must deal with different possibilities of data distribution. Different sites may contain data for a common set of features of the problem domain. In the case of relational data this would mean a consistent database schema across all the sites. This is the homogeneous case. Most algorithms that have been developed for
DKD work by moving either final results or final models among the sites.

Kargupta et al. have presented an algorithm for clustering high-dimensional heterogeneous data using a distributed principal component analysis (PCA) technique called the collective PCA. In their approach partial results (principle eigenvectors) are computed at each site and transmitted to a central site along with a number of data tuples corresponding to each eigenvector. The error between the actual global result and the computed result decreases as the number of tuples transmitted from each site to the central site increases. In our approach presented in this dissertation seek to decompose the computations of \textit{k-means} algorithm in such a way that no actual data tuples need to be transmitted among the sites and the error is kept low by transmitting intermediate results only.

**Distributed Data Sources:** Situations that are emerging in the networked infrastructure require data and knowledge from a number of geographically distributed but networked sites to be simultaneously considered for computations. A number of such local databases together form a global dataset that contains the data relevant for a computation. For example, some computation may require simultaneous consideration of data, parts of which reside in census databases, labor statistics databases, health databases, and employment related databases. Each of these is a huge and complex database and resides on a different site in a different city. One cannot hope to move all these databases to a single computer site, merge or \textit{join} them, and then analyze the resulting huge database. It would be desirable to have algorithms that let the individual databases reside at their own sites and work with an imagined implicit \textit{join} of the databases by decomposing themselves into sub-computations such that each subcomputation can be performed locally within a single physical database.

In our work our approach has been to decompose the steps of an algorithm into localized
computational steps that can be executed at the participating sites and the intermediate results thus obtained are transmitted to the central site. This may have to be repeated a number of times until an algorithm is completed. It thus differs from the three modes of mining distributed databases and works by exchanging partial results during the execution of an algorithm.

Our algorithms are tailored for situations in which we don’t have closely connected processors. There are multiple processors but they are independent and reside at geographically distant sites and communication among them may not be very fast. This communication is generally many orders of magnitude slower than the rates of inter-processor communication in a set of closely coupled processors.

1.5 Brief Summary of Results and Contributions

Most graph and clustering algorithms assume that the entire database is available in a single computer. In the emerging networked environments, this may not be true because of various reasons such as the sheer volume of the data, priority concerns, data security, a host of other reasons. The existing graph and clustering algorithms need to be adapted for distributed databases and hence have to be adapted appropriately. In this dissertation we develop decomposed algorithms for finding path, computing shortest paths spanning tree, computing minimum spanning tree, and performing unsupervised learning by clustering. The research would relate to the design, implement and compute complexity analysis of decomposable graph and clustering algorithms in the context of distributed databases. Some specific aims of the research can be described as:
1. How traditional algorithms can be adapted for a distributed database situation.

2. Design new decomposable algorithms to work with distributed databases.

3. We use a heuristic computation at each site to iteratively converge.

4. Performance of the clustering and graph operation tasks using minimum inter-site communication of local summaries and interfaces to obtain the same results that would have been obtained, were the computations performed with all the data transferred to a single site.

5. Implementation and discussion of the complexity of the new distributed algorithms.

6. Discussion of some general issues relating to adaptability of graph and clustering algorithms to distributed data situations.

7. Design algorithms are decomposable at run time depending on the collection of sites with which a site can connect and communicate and also the set of shared among the database.

8. The designed algorithms can be executed with very low communication overhead. In almost all situations the cost of exporting datasets to a common site and then rebuilding the global data at a local site for processing will take more communication energy and time than what is needed by the algorithms presented here.

9. Design independent algorithms of the manner in which the database may interface and overlap with each other (via the sets of shared among them) and also the set of sites that may be available for participation in a particular instance of a computation.
10. Design algorithms preserve the privacy and security of the data at individual sites by requiring transmission of only minimal information to other sites.

1.6 Potential Impact of this Research

We consider our work is very important because these databases cannot be moved to a single site due to size, security, privacy and data-ownership concerns. All of these databases together constitute the dataset and we run our algorithms without moving the data to the requiring site.

1.6.1 Algorithms Design

- These algorithms can perform tasks in a number of databases residing at different network sites without having to move them to a single site the communication cost of such decomposable algorithms turns out to be a very low.

- These algorithms can prove extremely useful for many applications with databases at different sites of a network and still obtain the same or too close results to the one would have been obtained if we had moved all these databases to a single site.

- These algorithms preserve the privacy and security of the data at individual sites by requiring transmission of only minimal information to other sites.
1.6.2 Applications

A very important contribution of these results is that some graph operations and clustering tasks can be performed in a number of databases residing at different network sites without having to move them to a single site. This research will have an impact on every field that uses computer algorithms and need to process data residing at different sites of a computer networks. There are a large number of applications which will benefit from this research and will help conserve the communication resource, in addition to preserving the security and integrity of databases. The following are some situations in which our algorithms can be applied:

- Let us consider a number, $n$, of airlines, each of which has its own database of flight segments represented as a graph. Some destinations may be served by more than one airline and some may be served by only one of them. When these airlines decide to collaborate they may want to find possible paths or minimum cost paths in their combined network. They may not want to transfer their entire databases to a single computer because of size, privacy, security, or extremely dynamic nature of the databases. An algorithm that can dynamically consult all the individual databases to infer the minimum cost paths is desirable, and our algorithms seek solutions for these and similar problems.

- Consider the example of a financial institution that maintains credit card accounts. A number of databases used by this institution, which typically reside in different cites, are:
  
  - A database containing fixed data about customers such as income and address
information.

- A database of credit card purchases and payments made by customers.
- A database of vendors that accept the card.
- A database containing credit-rating related facts about customers.

Some simple computations can be performed by retrieving a few records from each database. However some other computations require a large amount of data to be consulted at each site. In the latter kind of computations that will tremendously benefit by the dynamic and data dependent composition of algorithms proposed here. A few examples of such computations are:

1. Discovery of customer or vendor patterns.
2. A decision about a particular customer or vendor based on his/her historical credit, payment, and purchases data.
3. Computation of a discriminant function to screen either the risky transactions or the risky customers.
4. Search for the best customers according to some criterion which depends on the data from various databases.

In the above situation the computation cannot be performed by retrieving a few specific records but depends on extensive processing of the entire data at each site. In addition we have some constraints like the database sites can not be transferred due to security, size, data ownership, privacy considerations, etc. Constraints similar to these in many commercial, financial, scientific, and defense databases. Despite these constants it is
possible to perform many computations in the collectives databases, without moving
the large quantities of data from their native sites.

- Design devices and engineering structures such as turbines aircraft systems and offshore
oil platforms. This requires selection of a synchronized set of parameters from a number
of networked data sites. Each data site may be generating information/knowledge
from Computational Fluid Dynamics (CFD), Finite Element Analysis (FEA/FEM),
Computer Aided Design (CAD), and other computational and simulation tools.

1.7 Organization of the Dissertation

In Chapter 2 We overview the distributed knowledge environment. We give a brief outline
of an abstraction for cooperating sites in a networked environment, our cost models, and
also an abstraction for defining a computation in such an environment. These abstractions
are the basis on which we have developed.

In Chapter 3 We give the relevant research to our work and how it is different from the
one addressed by much of the existing parallel algorithms and parallel processing research
and the one addressed by the Distributed Shared Memory (DSM) algorithms.

In Chapter 4 We present three self-decomposing algorithms to find the path between two
vertices in a graph stored in parts across networks and the analysis of each algorithm. The al-
gorithms start from low granularity to higher granularity to reduce the number of exchanged
messages among the participating sites.

In Chapter 5 two self-decomposing algorithms for finding shortest paths spanning tree and
minimum spanning tree for a graph stored across a number of distributed databases are pre-
sented. The complexity analysis for each algorithm is given. Also the algorithm start from low granularity to higher granularity to reduce the number of exchanged messages among
the participating sites.

In Chapter 6 we present self-decomposing versions of \textit{k-means} clustering algorithm. Algorithms presented in this chapter seek to emulate the behavior and performance of \textit{k-means} algorithm for horizontally partitioned datasets. Another objective of the algorithms is to perform clustering by minimizing the number of messages exchanged among the participating sites.

In Chapter 7 we present a clustering algorithm for a dataset that is partitioned either horizontally or vertically across various sites of a network. One of the objectives of the algorithms presented here is to exchange minimum number of messages among themselves while clustering the data distributed across the participating sites.

We conclude, in Chapter 8, by discussing the contributions and research directions of this work.
Chapter 2

Computations in Distributed Environments

Most algorithms in the field of Computer Science have been designed for environments in which all the relevant data is stored at a single geographical computer site. In the emerging networked knowledge environment computational tasks are encountering situations in which the relevant dataset exists in the form of a number of geographically distributed databases that are connected by wide-area communication networks. A common constraint in these situations is that the databases cannot be moved to other network sites due to security, size, privacy or data-ownership considerations [11, 12, 13]. This networked environment is very different from the one addressed by much of the existing research in parallel algorithms and parallel processing where algorithms have been designed and analyzed for the contexts of closely coupled processors working on a shared and easily accessible dataset [69]. The networked environments are also very different from the one addressed by the Distributed Shared Memory (DSM) algorithms. Two major differences between our problem context and
the DSM algorithms are:

1. The optimization problem in DSM is to minimize the number of participating processors whereas, in our algorithms the number of participating processors are fixed and we seek to minimize the number of exchanged messages among the participating sites.

2. In DSM model a message from one processor to the other is limited to reading data from the shared memory whereas in our problem, the messages sent to other processors are for executing some functions with the local data whose results are then returned to the requesting processor.

For our problem formulation, therefore, we need algorithms that can decompose themselves into partial computations to match the network locations of data components, port and execute these partial computations at the participating sites, and compose the received partial results to construct global results.

We give here a brief outline of an abstraction for cooperating sites in a networked environment, and also an abstraction for defining a computation in such an environment. These abstractions are the basis of our research.

### 2.1 An Abstraction for Networked Data

A computation in the environment of networked knowledge, represented in Figure 2.1 below, may present itself as follows:

1. Some network site Init-Site wants to perform a computation $C$ which requires a body of data $D$. The entire dataset $D$ may not be available on Init-Site itself, or on any one
site alone.

2. A search is performed over the network to identify those other sites that can provide some relevant parts of $D$ for the computation and are willing to cooperate. Init-Site then selects a sufficient set of participant sites that together constitute the body of data $D$. Attributes of individual databases at different participating sites may be unique to their sites or may overlap with those of other databases at the participating sites.

3. The computation initiator site Init-Site determines a decomposition of computation $C$, compatible with the distribution and overlap of attributes of $D$ across various databases. It then seeks results of local computations from the participating sites and composes them to construct the global result. A number of (decomposition, partial computation, composition) iterations may be required for completion of $C$. Not all computations are decomposable in this manner and our goal is to determine the decompositions of some of the useful computations.

![Network Diagram]

Figure 2.1: Distributed Data/Knowledge Sources
A characteristic difficulty encountered in this situation is that data cannot be moved from one network site to the other where an algorithm may be executing, but partial computations must be transported to the network sites where the relevant data components reside. Also, each new instance of the same computation may encounter a different set of participating sites and, therefore, a different distribution (and overlap Pattern) of data attributes across the participating sites. This requires a completely different decomposition of the same computation. This is in contrast to many algorithms in the parallel processing domain where an efficient decomposition of an algorithm and data is created under the assumption that the closely-coupled processors and/or shared memory will have the capability to move the data to the designed processing sites. The algorithms that adapt to the data distribution and overlaps of attributes across the participating sites and decompose themselves successfully will be referred to as network algorithms.

A desirable property of the decomposed versions of algorithms (the network algorithms) is that they reduce the number of messages exchanged among the participating sites, and that they transmit only the minimal knowledge that must be shared for completing a particular computation. The objective of the research proposed here is to develop least expensive networks algorithms for performing various numeric and non-numeric computations in environments of arbitrary data distribution and overlap across the network sites.

The dataset $\mathcal{D}$ with which graph operations and clustering tasks are to be performed is considered in the form of a relation defined on a set of attributes $\mathcal{A}$. In a networked knowledge environment this relation $\mathcal{D}$ does not exist in the form of a single explicit table. Instead, it exists in a decomposed form. The two different ways in which this decomposition of $\mathcal{D}$ may be encountered are as in the following section.
2.2 Nature of Data Distribution

Let us say there are \( n \) different sites \( S_1, S_2, \ldots, S_n \), containing databases \( D_1, D_2, \ldots, D_n \) respectively. Depending on the sets of attributes contained in each \( D_i \), there are two primary ways in which the databases, together, may be seen as forming an implicit global dataset \( D \).

![Diagram of horizontally partitioned datasets](image)

**Figure 2.2: Horizontally Partitioned Datasets**

**Horizontally Partitioned Datasets:** Figure 2.2 shows a partitioning of \( D \) into components \( D_1, D_2, \ldots, D_n \) such that each \( D_i \) contains tuples consisting of an identical set of attributes \( A \); but a distinct set of tuples exists at each site. Each \( D_i \) resides on a different site of the network and the tuples contained in all the \( D_i \)'s, taken together, constitute the complete dataset \( D \).
**Vertically Partitioned Datasets:** Figure 2.3 shows another way in which components of \( \mathcal{D} \) may be partitioned across a network. In this case each component \( D_i \) consists of tuples made with a different set of attributes but each \( D_i \) may share some attributes with those of other databases, \( D_j, j \neq i \). Each \( D_i \) may also contain some attributes that are unique to the local site and are not shared with a database at any other site. In effect, each \( D_i \) is a projection of an implicit global \( \mathcal{D} \).

The objective of our algorithms is to perform the computations using inter-site communication of local summaries and obtain the same results that would have been obtained with the computations performed with all the data transferred to a single site. The algorithms should be able to decompose themselves into local computations for any arbitrary partitioning of attributes encountered in an instance of computation.

Figure 2.1 schematically shows \( n \) databases located at \( n \) different network sites that together constitute the data \( \mathcal{D} \) for some computation. As an abstraction, we model the data at each site by a relation represented as a set of tuples. When the actual data at some Site \( m \)
is not organized as a relation, (as in a database of maps or images) the interface agent \( A_m \) can, in effect, create and provide a virtual view of a relation for the underlying data to the other network sites.

The set of attributes contained in a relation \( D_i \) is represented by \( X_i \). For any pair of relations \( D_i \) and \( D_j \) the corresponding sets \( X_i \) and \( X_j \) may have a set of shared attributes given by \( S_{ij} \). The dataset \( \mathcal{D} \) with which the computation is to be performed is a subset of a set of tuples generated by a join operation performed on all the relations. \( D_1, D_2, \ldots, D_m \). However, the tuples of \( \mathcal{D} \) cannot be made explicit at any one site because entire \( D_i \)s cannot be moved to other sites. The tuples of \( \mathcal{D} \), therefore, must remain only implicitly specified. \textit{This inability to make explicit the tuples of \( \mathcal{D} \) is at the root of the for designing decomposable version of algorithms.}

To facilitate computations with implicitly specified sets of tuples for \( \mathcal{D} \), we define a set \( S \) that is the union of all the attribute intersection sets \( S_{ij} \)s defined above. That is,

\[
S = \bigcup_{i,j,i\neq j} S_{ij} \tag{2.1}
\]

The set \( S \) contains the names of all those attributes that occur in more than one \( D_i \). We define also a new relations called \textit{Shareds} which contains all possible tuples that can be enumerated for the attributes in set \( S \). The example in Figure 2.4 shows two databases \( D_1 \) and \( D_2 \); the set \( S \) of shared attributes contains only one attribute \( B \); the relation \textit{Shareds} consisting of all all the tuples for attribute in \( S \) is enumerated in the third table; the explicit join of \( D_1 \) and \( D_2 \) containing all the tuples is shown in the fourth table. The tuples that would be included in the complete explicit \textit{Join} but are not needed for the mining problem
should be excluded from consideration by the decomposed algorithms. The above example shown only one kind of data distribution. In another case each database may contain exactly the same set of attributes.

2.3 Explicit versus Implicit Data

Vertically Partitioned datasets require computations to be performed in an implicit Join, \( \mathcal{D} \), of all the \( D_i \)s, but without ever making explicit the tuples of \( \mathcal{D} \). The computation must appropriately account for all the shared attributes that would have played a role in enumerating the tuples of the Join-ed \( \mathcal{D} \), if it were to be made explicit.

For example a database at Site 1 may store tuples with attributes \( A \) and \( B \) and a database at Site 2 may store tuples formed with attributes \( B \) and \( C \). A Join of the two databases will produce all those tuples in the \( A - B - C \) space that are consistent with the two projections. However, some of these tuples may not be consistent with the domain knowledge and the constraints of the implicit 3-D space. We define Exclusions to be a set of
domain knowledge based constraints that help us prune the database produced by the Join of all the local databases. For example, the set Exclusions may contain the condition that values \( A = 1 \) and \( C = 3 \) are inconsistent and no tuple in \( D \) should have this combination of values. This information can then be used to eliminate from the Join all those tuples that satisfy the conditions included in the set of conditions Exclusions. Therefore, the implicit set \( D \) with which we want to perform the clustering task is given as:

\[
D = (D_1 \bowtie D_2 \bowtie \ldots \bowtie D_n) - E
\]  

(2.2)

Where \( E \) is the set of all tuples that are consistent with the conditions in set Exclusions. Our algorithms are designed to perform computations with a \( D \) that can take care of the conditions in the Exclusions set and exclude from implicit Join those tuples that are known to inconsistent.

### 2.4 An Abstraction for Networks Algorithms

Let us say a result Result is obtained by applying an algorithm Alg to a dataset \( D \) that is:

\[
Result = Alg(D)
\]  

(2.3)

In the case of algorithms for distributed databases, \( D \) cannot be made explicit and is known only implicit in terms of the explicit components \( D_1, D_2, \ldots, D_n \). The implementation of
$Alg$ in the last equation can be redesigned by a functionally equivalent formulation:

$$\text{Result} = Alg'(f_1(D_1), f_2(D_2), \ldots, f_n(D_n))$$

That is, a local computation $f_i(D_i)$ is performed at $Site_i$ using the database $D_i$. The results of these local computations are aggregated using the operation $Alg'$. The set of $Alg'$ and $f$ operators would vary with the $D_i$s and hence a different set would need to be generated for each new instance of $D_i$s that is encountered by the computing agent at a site. A schematic in Figure 2.5 shows the process by which a network algorithm would compute $Alg'$ from the $D_i$s. The components operators of a decomposition ($Alg'$ and $f_i$s), therefore, need to by dynamically determined for each instance of $Alg(D)$ depending on the participating sites; the attributes contained in their native databases; and the overlap

![Diagram](image-url)
pattern of their attributes as reflected in the set of shared attributes.

2.5 Decomposition Strategies into Network Algorithms

There are three possible approaches for decomposing complex algorithms into decomposable network algorithms. In order of increasing complexity they are:

1. **Algebraic Only Decompositions**: In this approach we determine the decomposition operators (sets of $f_i$'s and $Alg$'s) for all common algebraic primitive and store them as library functions at each site. At execution time the parameters needed by these functions are the attribute-names and their possible-values at each participating site and the set of shared attributes in that particular instance of the computation. The initiator site Init-Site then executes the algorithms at its own site and requests summaries from participating sites whenever an algebraic computation needs to be made. This type of decomposition can be extend to any kind of algebra, including real/integer algebra, and some graph-theoretic and network-flow problems formulated as an algebra. The decompositions presented in [11, 12] are this kind of decompositions.

2. **Mixed Decompositions**: In this approach a number of algebraic primitives required by a chunk of algorithmic Primitives can be decomposed into a single set of decomposition operators. For example, the iterations of a simple While-loop structure can be compressed into a single set of decomposition operators. Whenever the Init-Site encounters the known kind of algorithmic primitives it can generate the corresponding decomposition operators for the participating sites.
3. **Algorithmic Decompositions:** In this approach algorithmic primitives such as while-loops and conditionals are decomposed in terms of their parameters and the Init-Site can at run time integrate the algebraic and the algorithmic primitives to, in effect, decompose any algorithm it may encounter.

### 2.6 Implementation of Network Algorithms

There are two main ways in which a network algorithm can be implemented:

1. **Stationary Agents:** Such an agent remains at its native site, determine the decomposition operators for each step of the algorithm, ships out \( f_1 \)'s to the participating sites, gather the results, applies the \( Alg' \) operator and moves on to repeat the same with the next sequential algorithmic primitive until the algorithm is completed .

2. **Mobile Agents:** Such an agent starts from the Init-Site site , visits a participating site, executes all the decomposition operators that it can that site and then moves on to the next site. It keeps on moving until the computation is complete and then returns to the Init-Site site.

### 2.7 Our Cost Models for Complexity Computing

Since the objective of our algorithms is to obtain the desired results with minimum exchange of messages among the participating sites, we need to examine our algorithms with a cost model different from the ones that typically measure the cpu-cycles consumed and memory accesses made by an algorithm. We choose the following three cost models for examining
the efficiency of our algorithms. In the first model we only count the number of messages exchanged among the sites, in the second model we consider a combination of the communication cost and the local computation cost and in the third model we include the gains made by simultaneous processing of messages at the participating sites.

1. **Communication cost only:** In this cost model we count the number of messages, $N_m$, that must be exchanged among all the participating sites in order to complete the execution of the algorithm. One exchange of message includes one message sent by the querying site requesting some information and the reply message sent by the responding site. This cost model is relevant because in many algorithms the time spent in exchanging a message between two network sites is orders of magnitude higher than the cost of a local computation.

2. **Communication Plus Local Computation Cost:** In this model we examine a weighted sum of the number of messages exchanged and the number of local operations performed. If $N_m$ messages are exchanged among the sites and a total of $N_q$ queries are performed at all the sites combined then the algorithm’s cost is given by:

\[ a \times N_m + b \times N_q \]  

Where $a$ and $b$ are the weights representing the relative costs of exchanging a message and performing a local operation.

3. **Elapsed communication plus computation cost:** In this model we examine a weighted sum of the number of messages exchanged and the number of local opera-
tions performed, while discounting the effects of messages and operations that can be executed in parallel, simultaneously at different sites. If \( N_m \) messages are exchanged among the sites and a total of \( N_q \) queries are performed at all the sites combined, then the algorithm’s cost is given by:

\[
(\alpha * N_m + \beta * N_q) / p
\]  

(2.6)

Where \( \alpha \) and \( \beta \) are the weights representing the relative costs of exchanging a message and performing a local operation, and \( p \) is the average number of messages that can be exchanged in parallel (the number of participating sites).

The first cost model is useful when we want to optimize the number of messages exchanged among the sites and the time for local computations within individual sites is negligible compared to the time taken to exchange a message among two sites of a wide-area network. The second cost model is useful when the local computation time within a site is not negligible and must be included within the cost models. When the databases stored at the sites are huge, as in many scientific and data-mining applications, the time to execute a local computation may be comparable to the time taken for exchanging a message across a wide-area network. The third cost model seeks to optimize the total elapsed time for a computation by discounting for the simultaneous exchange of messages and execution of local computations at the participating sites. The first two cost models are useful when either the number of messages exchanged or the energy consumed at a site is the criterion for selecting an algorithm. The third cost model is useful when our criterion is the total elapsed time for executing the algorithm.
Chapter 3

Relevant Research

The major thrust in the field of design and analysis of algorithms has been directed towards designing computationally efficient algorithms. There has been a lot of research in algorithms for sequential and parallel architectures [3, 4, 5, 57]. The main focus of the parallel and distributed algorithms has been on parallel algorithms and parallel processing research. In this research algorithms have been designed and analyzed for the situation in which a set of closely coupled processors work on a common dataset that is easily accessible and sharable by each processor [69]. The field of distributed knowledge environments, especially where data cannot be shared easily, and the corresponding set of distributed algorithms, has received a very little attention. The decomposition can be considered as regular algorithms being implemented by a number of coordinating agents either by exchanging messages among themselves or by visiting participating sites to gather results of local computations. Significant research has been done in the field of Multi Agent learning and distributed AI [31, 35, 42]. Research in multi-agent systems (MAS), or distributed artificial intelligence (DAI), dates back to the late 1970’s. Initial work on the area focused on distributed inter-
pretation of sensor data, organizational structuring and generic negotiation protocols. But several recent developments have helped reshape the focus of the field. Like the rest of AI, the field has matured from being largely exploratory in nature to focus on formal theory of negotiation, distributed reasoning, multi-agent learning and communication languages the field is also maturing to the point of developing its first few fielded applications. Multi-agent systems research has focussed on a number of issues relating to the distribution of knowledge and processing capability over a loosely connected network. In most of this work [26, 38, 54, 81], agents are modeled as having only a limited view of the global resources and knowledge. The objective of learning by each agent is that the whole society should coverage to an optimal operating point after each agent has individually learned about its own optimal performance.

In the algorithms that we have implemented, the computing agent at each site does not possess any uncertainty about the state of data/knowledge of other data sites or computing agents. It only needs to ask for information and/or do some local computation and it would be truthfully given all the needed information sought by the other agents. The goal of the agents of our formulation is to minimize the exchange of information among themselves for performing a task. We designed, implemented and analyzed decomposable graph algorithms like finding path, shortest paths spanning tree, minimum spanning tree and decomposable algorithms for clustering datasets.
3.1 Graphs Relevant Research

Graph algorithms for single and parallel processor architectures have been studied and analyzed extensively (path, shortest path, minimum spanning tree, etc.) [27, 28, 48, 51, 55].

The main focus of most of the parallel and distributed algorithms has been on closely coupled processor systems where data can be easily shared by the processors and the number of processors assigned to an algorithm may depend on the total number of vertices of the graph to be processed [15, 27, 51, 55].

Most parallel algorithms for finding minimum spanning tree of a graph are based on finding minimum cost edges emanating from each vertex of the graph. Each processor is given access to the whole graph and is assigned the task of finding the minimum cost edge emanating from a single site [15]. This algorithm can complete in $\log(n)$ time steps but requires $n^2/2$ processors, each of which has access to the entire graph.

In environments of widely distributed knowledge, network sites where sub-graphs reside are fixed and the data cannot easily be moved to a single site to construct the complete graph. Algorithms for processing such distributed graph structures have received very little attention [11, 12]. Focus of our work is on such situations of widely distributed knowledge and therefore it significantly differs from that of the parallel and DSM algorithms for graph operations.

3.2 Clustering Relevant Research

Parallel implementations of pattern analysis algorithms take advantage of the high performance of multiprocessor computer systems [68, 73, 78, 89] and work by transferring data
form one processor top the other. It is desirable to implement clustering algorithms in parallel and even build specialized hardware chips for clustering when large amounts of data are available at a single geographical site. Rasmussen and Willett [78] discuss parallel implementation of the single link clustering methods on an SIMD array processor. Their parallel implementation of the SLINK algorithm does not decrease the $O(n^2)$ time required by the serial implementation, but a significant constant speedup factor is obtained. Li and Fang [68] describe parallel partitioning clustering (the k-means clustering algorithm) and parallel hierarchical clustering (single link clustering algorithm) on an $n$-site hypercube and an $n$-site butterfly. Olson [73] has described several implementations of hierarchical clustering algorithms which achieve $O(n)$ time in $n$ site CRCW PRAM and $O(n\log n)$ time on $n/\log n$ site butterfly networks or tree. All these parallel clustering algorithms are tailored for situations in which: (i) all data resides in main memory or distributed shared memory of a set of closely connected processors; (ii) there is a large number of closely connected processors at a single site that can access the shared memory to achieve the reasonable performance; and (iii) Inter-processor communication is extremely fast and involves reading data in the main or the shared memory.

Our algorithms are tailored for situations in which we don’t have closely connected processors. There are multiple processors but they are independent and reside at geographically distant sites and communication among them may not be very fast. This communication is generally many orders of magnitude slower than the rates of inter-processor communication in a set of closely coupled processors.

In [49] Grossman describes three modes for mining geographically distributed databases. These are based on transferring (i) results; (ii) models; or (iii) data from distributed sites
to a central site. Distributed knowledge discovery (DKD) (Chan and Stolfo [25]; Lam and Segre [67]; Yamanishi [90]; Crestana and Soparkar [34]; Tumer and Ghosh [86]; Grossman et. al. [49]; Park et. al. [75]) embraces the growing trend of merging computation with communication and explores all facets of the KDD process in the context of the emerging distributed computing environments. (DKD) accepts the fact that the data may be inherently distributed among different loosely coupled sites connected by a network and the sites may have heterogeneous data. It offers techniques to discover new knowledge through distributed data analysis and modeling using minimal communication of data. DKD must deal with different possibilities of data distribution. Different sites may contain data for a common set of features of the problem domain. In the case of relational data this would mean a consistent database schema across all the sites. This is the homogeneous case. Most algorithms that have been developed for DKD work by moving either final results or final models among the sites. An example of this approach is the work by Chan and Stolfo [25] in which they have investigated the idea of multiple learning processes with supervised inductive learning. These methodologies run the learning algorithm in each of these subsets of data, and combine the results by identifying two main meta-learning components: combiner and arbiter. Both components are independent of the learning algorithms used with individual components in generating the classifiers. The combiner attempts to reveal relationships among the learned classifiers’ prediction patterns and the arbiter tries to determine the correct prediction when the classifiers have different opinions. Various schemes for the combiner and arbiter have been developed [19, 20, 21, 22, 23, 24].

Kargupta et al. in [59] have presented an algorithm for clustering high-dimensional heterogeneous data using a distributed principal component analysis (PCA) technique called the
collective PCA. In their approach [59] partial results (principle eigenvectors) are computed at each site and transmitted to a central site along with a number of data tuples corresponding to each eigenvector. The error between the actual global result and the computed result decreases as the number of tuples transmitted from each site to the central site increases. In our approach presented in this dissertation we seek to decompose the computations of \textit{k-means} algorithm in such a way that no actual data tuples need to be transmitted among the sites and the error is kept low by transmitting intermediate results only.

Our approach is different since our approach has been to decompose the steps of an algorithm into localized computational steps that can be executed at the participating sites and the intermediate results thus obtained are transmitted to the central site. This may have to be repeated a number of times until an algorithm is completed. The \textit{k-means} algorithm presented here is in this same spirit. It thus differs from the three modes of mining distributed databases suggested in [49] and works by exchanging partial results during the execution of an algorithm.

As discussed in chapter two \textbf{Distributed Data Sources:} Situations that are emerging in the networked infrastructure require data and knowledge from a number of geographically distributed but networked sites to be simultaneously considered for computations. A number of such local databases together form a global dataset that contains the data relevant for a computation [11, 12]. For example, some computation may require simultaneous consideration of data, parts of which reside in census databases, labor statistics databases, health databases, and employment related databases. Each of these is a huge and complex database and resides on a different site in a different city. One cannot hope to move all these databases to a single computer site, merge or join them, and then analyze the resulting huge
database. It would be desirable to have algorithms that let the individual databases reside at their own sites and work with an imagined implicit \textit{join} of the databases by decomposing themselves into sub-computations such that each subcomputation can be performed locally within a single physical database.
Chapter 4

Decomposable Algorithms for Path Connectivity

Finding path is one of the most fundamental graph problems. This problem comes up in practice and arises as a subproblem in many graph optimization algorithms. Most path finding algorithms have been designed for environments in which all the relevant data is stored at a single computer site.

In this chapter we present self-decomposing versions of three graph algorithms for finding a path in a graph stored in parts on distributed databases. An objective of our algorithms is their self-decomposability for any way in which the graph may be distributed across the network. Another objective of the algorithms is to perform their tasks by minimizing the number of messages exchanged among the participating sites. Here is one of our algorithms application. Let us consider a number, $n$, of airlines, each of which has its own database of flight segments represented as a graph. Some destinations may be served by more than one airline and some may be served by only one of them. When these airlines decide to
collaborate they may want to find possible paths, or minimum cost paths in their combined network. They may not want to transfer their entire databases to a single computer because of size, privacy, security, or extremely dynamic nature of the databases. An algorithm that can dynamically consult all the individual databases to infer the paths is desirable, and our algorithms seek solutions for these and similar problems.

4.1 Graph Representation in Distributed Environments

The representation of a graph for the cases of closely coupled processors and the widely distributed sites can be compared as follows. In the first case a graph $G$ with vertices $V$ and edges $E$ is stored either on a single computer in the form of an adjacency table or in the shared memory addressable by each of the closely coupled processors. In a widely distributed environment overlapping components of a graph may be stored on different sites of a network. Figure 4.1 shows an example graph that may be stored on a single site. Figure 4.2 shows

Figure 4.1: Complete Graph
two graphs that may be stored on two different sites of a network. The two subgraphs together constitute the same graph as in Figure 4.1 and share some of the vertices. This situation can be easily generalized to \( n \) network sites, each containing a part of the complete graph. One major assumption made by the closely coupled processors paradigm is that

![Graphs](image)

**Figure 4.2: Two SubGraphs**

the data can be efficiently moved around among the processors and can be made available very quickly to a processor site that needs it. This assumption does not hold good in the knowledge environment where potentially huge databases exist at geographically dispersed sites, connected by wide-area networks. From the perspective of elapsed time, it is very expensive to move data around such networks. The alternative for us is to design algorithms that can decompose themselves according to the distribution of data across the sites. Partial computations resulting from such decompositions can then be sent to the sites where data resides and the results obtained then combined to complete the desired computations. Since a different decomposition of an algorithm may be required for each type of distribution of data across sites, we would like to develop algorithms that can decompose themselves into
relevant partial computations to suit the distribution of data.

Algorithms can process a graph structure of Figure 4.1 when it is stored on a single network site. Our goal is to design algorithms that obtain the same results with the multiple sub-graphs that are stored on different sites of a network, without having to assemble the complete graph at a single site of a wide-area network. The algorithms should also minimize the communication cost among the sites. For example, we may want to determine the edges that should be retained in a minimum cost spanning tree of the complete graph by the network sites having to exchange minimum number of messages among themselves.

The vertices and edges of the two subgraphs of Figure 4.2 can be represented by the following two tables at their respective sites:

<table>
<thead>
<tr>
<th>vertex1</th>
<th>vertex2</th>
<th>weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>1</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>8</td>
<td>7</td>
<td>5</td>
</tr>
</tbody>
</table>

Table 4.1: Database at Site-1
Table 4.2: Database at Site-2

We now present three decomposable algorithms for determining whether two vertices of the complete global directed graph $G$ connected or not and return the path between them.

It is assumed that any of the sites authorized to access the databases can initiate an algorithm for any of the above operations.

We start by introducing the concept of **Granularity** of algorithm decomposition as the following.

If $A$ is an algorithm for all data on a single site then we say algorithm $B$ is a low granularity decomposition of $A$ if $B$ is the decomposition of every atomic step of $A$, and we say $C$ is a high granularity decomposition of $A$ if $C$ is the decomposition of $A$ at a higher functional level.

### 4.2 Low Granularity Algorithm for Path Finding

The objective of this algorithm is to check whether a path exists between two vertices of the complete global connected directed graph, subgraphs of which are stored on different sites of a network. Each network site may have one or more components of the global graph. Here in
this algorithm we are constrained to exchange only one summary per message. This situation arises when the databases stored at different sites can request each other for information by issuing SQL queries over the network. Only one queries may be issues in each message sent across the network. The other alternative is to collect a number of SQL queries for a site and then send them in a wrapper to the other site so that only one message needs to be exchanged for a number of queries. This would require the algorithm decomposition to be done in a different way so that multiple queries can be collected before a message is dispatched to a site.

4.2.1 Algorithm Outlines

**Input:** \( v_i, v_j \) are the vertices between which we want to check the existence of a path in the complete graph \( G \).

**Output:** The path between \( v_i \) and \( v_j \) and “false”, otherwise.

**Data Structure:** All the following are stored at the network site initiating the query.

- An array \( Mark[1 : n] \) initialized to 0’s to keep track the checked vertices,

- A Queue \( Q \) initialized to empty,

- An integer \( MessageCounter = 0 \) Counts the number of messages exchanged
Algorithm:

Mark vertex $v_i$ by 1
call Enqueue($v_i$)
$PathFound = false$

while notemptyQueue and $PathFound = false$ do
    d = Dequeue()
    Send Message to every $site_i$ to find all the reachable vertices from the vertex $d$
    Result = Select all $vertex_2$’s from $SG_i$ that are reachable from $d$
    MessageCounter+= n
    For each element $R \in Result$ set do
        if $(R = v_j)$
            $PathFound = true$, Return the path
        else if $(Mark[R] = 0)$
            call Enqueue(R)
            $Mark[R] = 1$
        end if
    end else
    end for
end while

return $PathFound$

end Algorithm This algorithm can be initiated by any one of the sites authorized to access the databases at the participating sites. All the data structures and local variables described in the algorithm are maintained at the site initiating and controlling the algorithm.
In effect, this algorithm works by sending messages to every site to find out the graph vertices that are connected to a selected vertex. A trail of the list structure can be saved to determine the path between the two vertices. The complexity of this algorithm in terms of the cost models mentioned above is computed in the following sections.

4.2.2 Algorithm Analysis and Comments

We analyze below the path finding algorithm for its complexity from the perspective of the three cost models mentioned in chapter two Section 2.7 and the following two cases.

1. **Worst-Case Complexity**

   We define the worst-case scenario to be the situation when there is no path between the vertices $v_i$ and $v_j$ or a path is found almost towards the end of the search process.

   - **Cost Model # 1:**

     We assume that $SG_1, SG_2 \ldots SG_n$ have a total of $m$ vertices labeled $1, 2, 3 \ldots m$. The querying site maintains an auxiliary array $Mark[1 : m]$ to keep track of vertices that have been visited. In the worst case, and also in the case when there is no path between $v_i$ and $v_j$, the algorithm will send one message to each of the $n$ network sites, for each of the $m$ vertices of the graph. Therefore, a total of $m \times n$ messages will have to be exchanged. If $a$ is the total time taken to exchange a message the total cost, therefore, will be

     $$a \times m \times n \quad (4.1)$$

     The total cost, therefore, depends on the number of participating sites and the
number of vertices in the combined graph.

- **Cost Model #2**
  
  For each exchanged message, this algorithm performs one SQL query at the responding site. This query is from databases storing graph incidence matrices as shown in tables 1.1 and 1.2. The total cost of the algorithm will be the cost of exchanged messages plus the cost of performing SQL queries. Therefore, the total cost of the algorithm in the worst case is:

  \[ a \times m \times n + b \times m \times n \]  

(4.2)

Where \(a\) is the time taken to exchange a message and \(b\) is the time taken to perform a query at a site.

- **Cost Model #3**
  
  In this case the querying site can send all the messages in an iteration at the same time and for each exchanged message this algorithm performs one SQL query at the responding site. Therefore, the total cost of the algorithm in the worst case is:

  \[ (a \times m \times n + b \times m \times n)/n = (a + b) \times m \]  

(4.3)

Where \(a\) is the time taken to exchange a message and \(b\) is the time taken to perform a query. This cost is independent of the number of participating sites but depends on the total number of vertices in the combined graph.

2. **Best-Case Complexity**
The best case scenario is described by the situation when both $v_i$ and $v_j$ exist on the same site and there is a path between the two vertices completely residing on the same site. Because each site will have to answer a query about a path between any pair of edges so it still may be we have to send many messages to each local site, i.e., there is no difference between the worst case and the best case.

In the next two algorithms each local site will have to answer a query about the path between any pair of shared vertices instead of about edges between a pair of vertices as in the previous version of the algorithm. In that scenario the best case will be different from the worst case.

4.3 Higher Granularity Algorithm for Path Finding

We now present a different algorithm for finding a path between two vertices of the complete global graph. This algorithm works by using only the shared vertices among the subgraphs. If $SG_i$ is the subgraph residing on the site $D_i$ then some relevant entities are as follows:

- The complete graph $G = \bigcup_i SG_i$;
- $S_{ij}$ is the set of vertices shared between $SG_i$ and $SG_j$.
- The union of $S_{ij}$s for all values of $j$ is the set $Sh_i$ containing all the vertices shared by the subgraph $SG_i$ residing at site $D_i$. That is, $Sh_i = \bigcup_j S_{ij}$.
- The $Shared$ set is the union of all $Sh_i$s.

The main intuition behind this algorithm is that any interaction between a pair of sites is only through the shared vertices. Therefore, a path between two vertices, if not contained within a single site, must be connected only via the sites in the set $Sh_i$. This algorithm exploits this fact to compute a path between two vertices.
4.3.1 Algorithm Outline

\textit{PathFound} = \text{false}

MessageCounter = 0

for all \( D_i \) in \( \mathcal{D} \) do

send message to find if there is a path between \( v_i \) and \( v_j \).

MessageCounter += 1

If there is a path then \( \text{PathFound} = \text{true} \)

else

send message to each \( D_i \) to determine the vertices in \( Sh_i \) that are connected to \( v_i \) by a path.

\( XSet = \{ x : x \in Sh_i \text{ and there is a path from } v_i \text{ to } x \} \)

send message to each \( D_i \) to determine the vertices in \( Sh_i \) that are connected to \( v_j \) by a path.

\( YSet = \{ y : y \in Sh_i \text{ and there is a path from } y \text{ to } v_j \} \)

while (\( \text{PathFound} = \text{false} \))

with next pair of \((x \text{ in } XSet \text{ and } y \text{ in } YSet)\)

call Algorithm FindPath1 \( ((x, y),\text{Shared}) \)

end while

end else

return \( \text{PathFound} \)

end PathFind
4.3.2 Algorithm Analysis and Comments

In this algorithm each network site is assumed to contain only those graph vertices that it shares with other participating network sites. Therefore, the algorithm complexity will depend only on the count of the shared vertices. If this is much smaller than the total number of vertices in the combined graph then we will need to exchange many fewer messages among the participating sites. However, each site will have to answer a query about a path between any pair of shared vertices instead of about edges between a pair of vertices as in the previous version of the algorithm.

We analyze below the path finding algorithm for its complexity from the perspective of the three cost models mentioned in chapter two Section 2.7 and the following two cases.

1. **Worst-Case Complexity**

We define the worst-case scenario to be the situation when there is no path between the vertices \( v_i \) and \( v_j \) or a path is found almost towards the end of the search process.

- **Cost Model # 1:**

The algorithm will send \( n \) messages in step one to check if there is a direct path between \( v_i \) and \( v_j \) on any of the sites. If there are \( K_x \) and \( K_y \) members in the sets \( XSet \) and \( YSet \) respectively, and the number of total shared vertices in the set \( Shareds \) is \( K_{Shared} \), then algorithm \# 1 may have to be called, in the worst case, \( K_x * K_y \) times. Each time it would require to send a maximum of \( n * K_{Shared} \) messages as per the analysis in the preceding sections. Therefore, the total cost
of the algorithm in the worst case is:

\[ a(n + K_x * K_y * n * K_{Shared}) \] (4.4)

in the worst case, or in the case when there is no path between \( v_i \) and \( v_j \) where \( a \) is the time taken to exchange a message. The cost results shows that the number of message that need to be exchanged among the sites is not dependent on the number of vertices of the local subgraph at each site. This is very significant because it shows that as the sizes of the individual subgraphs grow, the communication complexity of our algorithm would remain unaffected.

- **Cost Model # 2**

For each exchanged message, this algorithm performs a local computation at the responding site. This local computation is at databases storing graph incidence matrices as shown for Figure 4.2. The total cost will be the cost of exchanged messages plus the cost of the local computations. Therefore, the total cost of the algorithm in the worst case will be the

\[ a * n * (1 + K_x * K_y * K_{Shared}) + b * n * (1 + K_x * K_y * K_{Shared}) \] (4.5)

where \( a \) is the time taken to exchange a message and \( b \) is the average time taken at a site to perform a local computation. This cost does not depend on the total number of vertices in the graph but instead depends on the number of vertices shared among all the network sites \((K_{Shared})\) and the number of participating
sites.

- **Cost Model # 3**

Since this cost model takes into account the possibility of simultaneous execution of a number of messages at different sites, the cost in this case will be the Cost Model # 2 taking into account the average number of messages that can be exchanged in parallel, which is $n$ in our situation. Therefore, the total cost for this algorithm in the worst case will be:

$$(a + b) * n * (1 + K_x * K_y * K_{shared}) / n = (a + b) * (1 + K_x * K_y * K_{shared})$$

(4.6)

where the variables have the same meaning as in the section above.

This cost is the same as in the preceding section except that it does not depend on the number of sites $n$ participating in a computation.

2. **Best-Case Complexity**

The best case scenario is described by the situation when both $v_i$ and $v_j$ exist on the same site and there is a path between the two vertices completely residing on the same site.

- **Cost Model # 1:**

since the best case occurred, when the path exists at the first site therefore the algorithm will send *one* message. Therefore, the total cost will be

$$a * 1$$

(4.7)
where $a$ is time taken to exchange a message.

- **Cost Model # 2**

For each exchanged message, this algorithm performs one local computation at the responding site. This local computation is at databases storing graph incidence matrices as shown for Figure 4.2. i.e., the total cost will be the cost of exchanged messages plus the cost of performing local computations. Therefore, the total cost of the algorithm in the best case will be:

$$ (a \times 1 + b \times 1) = a + b \quad (4.8) $$

Where $a$ is the time taken to exchange a message and $b$ is the average time taken at a site to perform a local computation.

- **Cost Model # 3**

Since this cost model takes into account the possibility of simultaneous execution of a number of messages at different sites, i.e. the cost in this case will be the Cost Model # 2 taking into account the average number of messages that can be exchanged in parallel, which is $n$ in our case. Therefore, the total cost for this algorithm in the best case will be:

$$ (a + b) \times n/n = (a + b) \quad (4.9) $$

Where the variables have the same meaning as in the section above. This cost is the same as in the preceding section except that it does not depend on the
number of sites \((n)\) participating in a computation.

4.4 More Efficient Algorithm for Path Finding

Here we present the third self-decomposing algorithm for finding paths in a graph whose component subgraphs are distributed across a number of sites that can communicate with each other. As we discussed before the abstract problem can be described as follows. Each of a number of data sites contains a subgraph. For each site, an implicit global connected directed graph can be formed by combining the local subgraphs of all the data sites with which it can communicate.

4.4.1 The Decomposable Algorithm

Let us say there is a path from \(v_i\) to \(v_j\) in implicit global \(G\) and it is written as a sequence of vertices:

\[
v_i, v_{sh_1}, v_{sh_2}, \ldots, v_{sh_p}, v_j
\] (4.10)

Where \(v_{sh_i}\) is the \(i^{th}\) shared vertex along the path. Any two consecutive vertices in this sequence are connected by a path completely residing within a local subgraph. That is, there is a path within a local subgraph that goes from \(v_i\) to \(v_{sh_1}\). The vertex \(v_{sh_1}\) is a shared vertex and exists in a number of subgraphs on different sites. The path then hops from one site to the other, and continues from \(v_{sh_1}\) to \(v_{sh_2}\) on the second site. So, it continues by covering some more distance on this site up to another shared vertex, then hops off to some other site that has the same shared vertex and continues up to the desired end
point. Whenever the path jumps from one site to another while staying on the same shared vertex, we call it a **hop** in the path. Our goal is that a site should be able to determine a minimum-hops path between any two vertices of the global graph around it. We now present our decomposable algorithm for finding a path between two vertices of the implicit global directed graph. This algorithm coordinates the local computations by means of vertices that are shared among pairs of subgraphs. If $SG_i$ is the subgraph residing on the vertex $D_i$ then some relevant notation is as follows:

- The implicit complete global graph $G = \bigcup_i SG_i$;
- $S_{ij}$ is the set of vertices shared between subgraphs $SG_i$ and $SG_j$.
- The union of $S_{ij}$s for all values of $j$ is the set $Sh_i$ containing all the vertices shared by the subgraph $SG_i$ residing at site $D_i$. That is, $Sh_i = \bigcup_j S_{ij}$.
- The **Shared** set is the union of all $Sh_i$s.

The main idea behind this algorithm is that any interaction between a pair of sites is only through the vertices shared among the subgraphs. Therefore, a path between two vertices of $G$, if not contained within a single site, must be a connection of a number of paths within local sites, joined at the shared vertices. This algorithm exploits this fact to compute a path between two vertices of $G$.

**Algorithm Outline**

**Data Structure:** The sets called $XSet_{in}, XSet_{archive}, XSet_S, XSet_{new}, YSet_{in}, YSet_{archive}, YSet_S$, and $YSet_{new}$. are maintained at the coordinator site and they store tuples, <
$x_i, y_i, S >$, containing information about the connectivity of shared vertices on individual sites. Each tuple in the Sets is to be interpreted as:

- The name, $x_i$, of a vertex that is the starting point of a path in the local subgraph;
- The name, $y_i$ of a vertex that is the end point of a path in the local subgraph;
- The name, $S$, of the site on which the path between $x_i$ and $y_i$ exists.

1. $PathFound = false$;

2. Send message to each $D_i$ to find if there is a path between $v_i$ and $v_j$ residing completely within the local subgraph of a site.

3. If there is a path within a site locally then Exit() with the message $PathFound = true$.

4. If there is no path between $v_i$ and $v_j$ completely within a local subgraph then initialize $XSet_{in}$ and $XSet_{archive}$ to $<null,v_i,null>$ and $YSet_{in}$ and $YSet_{archive}$ to $<v_j,null,null>$ and then Repeat the following steps:

   (a) Send current $XSet_{in}$ to each local site $S$ for update as follows:

   \[
   XSet_S = XSet_{in} \cup \{ x \in S : x \text{ is an end vertex in a tuple in } XSet_{in}, y \text{ is a shared vertex on site } S, \text{ and there is a path from } x \text{ to } y \text{ in the subgraph located at } S \}\]

   (b) $XSet_{new} = \bigcup_{all-sites} XSet_S$

   (c) $XSet_{in} = XSet_{new} - XSet_{in}$

   (d) $XSet_{archive} = XSet_{archive} \cup XSet_{in}$
(e) Send $Y_{Set_{in}}$ to each local site $S$ for update as follows:

$$Y_{Set_{S}} = Y_{Set_{in}} \cup \text{all } x, y, S \triangleright s : y \text{ is a starting vertex in a tuple in } Y_{Set_{in}}, x \text{ is a shared vertex on site } S, \text{ and there is a path from } x \text{ to } y \text{ in the subgraph located at } S.$$ 

(f) $Y_{Set_{new}} = \bigcup_{\text{all - sites}} Y_{Set_{S}}$

(g) $Y_{Set_{in}} = Y_{Set_{new}} - Y_{Set_{in}}$

(h) $Y_{Set_{archive}} = Y_{Set_{archive}} \cup Y_{Set_{in}}$

**Until** (i) Either there is a tuple $X$ in $X_{Set_{archive}}$ and a tuple $Y$ in $Y_{Set_{archive}}$ such that the end-point vertex in $X$ and the starting-point vertex in $Y$ are the same shared vertex; or

(ii) An iteration returns both $X_{Set_{in}}$ and $Y_{Set_{in}}$ as empty sets.

5. If the algorithm terminates due to the first condition then a path between $v_i$ and $v_j$ has been found. The path can be constructed by examining the tuples in $X_{Set_{archive}}$ and $Y_{Set_{archive}}$. If the algorithm terminates due to the latter condition then there is no path in $\mathcal{G}$ between the vertices $v_i$ and $v_j$.

### 4.4.2 Example

We show here an example execution of this algorithm. We show four subgraphs existing at four different network sites Figure 4.3 across a wide area network, possibly existing in different cities or even continents. The four subgraphs together implicitly define a global graph $\mathcal{G}$. The algorithm’s objective here is to determine if there is a path in $\mathcal{G}$ from vertex 1 in subgraph 1 to vertex 30 in subgraph 4. We initialize $X_{Set_{in}}$ and $X_{Set_{archive}}$ to
Figure 4.3: Subgraphs at four sites

<Null, 1, Null> and \( YSet_{in} \) and \( YSet_{archive} \) to <30, Null, Null>. The members of the \( XSet_{archive} \) and \( YSet_{archive} \) after successive iterations of the above algorithm are shown in the following three tables below.

<table>
<thead>
<tr>
<th>( XSet_{archive} )</th>
<th>( YSet_{archive} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;Null, 1, Null&gt;</td>
<td>&lt;30, Null, Null&gt;</td>
</tr>
</tbody>
</table>

Table 4.3: Initialization Step
\begin{table}[h]
\centering
\begin{tabular}{|c|c|}
\hline
$X Set_{archive}$ & $Y Set_{archive}$ \\
\hline
< Null, 1, Null> & <30, Null, Null> \\
<1, 7, 1> & <20, 30, 4> \\
\hline
\end{tabular}
\caption{First Iteration Results}
\end{table}

\begin{table}[h]
\centering
\begin{tabular}{|c|c|}
\hline
$X Set_{archive}$ & $Y Set_{archive}$ \\
\hline
<Null, 1, Null> & <30, Null, Null> \\
<1, 7, 1> & <20, 30, 4> \\
<7, 8, 2> & <22, 20, 3> \\
<7, 15, 2> & <15, 20, 3> \\
\hline
\end{tabular}
\caption{Second Iteration Results}
\end{table}

From the last update in $X Set_{archive}$ and $Y Set_{archive}$ we see that the shared vertex 15 exists as an end vertex in a tuple in $X Set_{archive}$ and also as a start vertex in a tuple in $Y Set_{archive}$. This means that there exists a path in global $\mathcal{G}$ between the vertices 1 and 30. This path can be inferred by examining $X Set_{archive}$ and $Y Set_{archive}$ and described as follows: Take path from vertex 1 to shared vertex 7 on site 1; then from shared vertex 7 to shared vertex 15 on site 2; then from shared vertex 15 to shared vertex 20 on site 3; and finally from shared vertex 20 to vertex 30 on site 4.

\subsection{Algorithm Correctness}

We first show that our algorithm will always find a path in global $\mathcal{G}$ if there exists one such path and then show that this will be the path requiring minimum number of hops across the
sites.

**Assertion:** If there exist paths in $G$ from a $v_i$ to a $v_j$ (both vertices are in $G$) then the algorithm will always find one of these paths.

**Proof:** If the path from $v_i$ to $v_j$ is completely within a local subgraph then it may not contain any shared vertices and will be found at step 3 of the algorithm. If the path from $v_i$ to $v_j$ is not completely within a local subgraph then it must contain a sequence of shared vertices. The number of shared vertices in a path cannot exceed the total number of shared vertices, (cardinality of the $Shareds$ set). This is because no vertex can be repeated in a path.

After the first iteration the set $XSet_{archive}$ contains all paths that can be reached from $v_i$ and containing at least one shared vertex.

After $k$ iterations the set $XSet_{archive}$ contains all paths starting from $v_i$ and containing at least $k$ distinct shared vertices ($k$ hops).

Similarly, after the first iteration the set $YSet_{archive}$ contains all paths containing at least one shared vertex that can reach $v_j$.

After $k$ iterations the set $YSet_{archive}$ contains all paths containing at least $k$ shared vertices that end in the vertex $v_j$.

In effect, $XSet_{archive}$, after $k$ iterations must contain the $k$-length prefix of the path in expression 4.10 above. Also, $YSet_{archive}$, after $k$ iterations must contain the $k$-length suffix of the path in expression 4.10.

If there is a path from $v_i$ to $v_j$ containing $k$ shared vertices, then after $k/2$ iterations we must have an overlapping prefix and suffix in the two sets. Therefore the path will always be found by this algorithm. (end of proof)
**Assertion:** The path found by our algorithm will be the one containing minimum number of hops from one site to the other.

**Proof:** Suppose there are two paths from $v_i$ to $v_j$ one with $q$ hops and the other with $q + l$ hops and $l > 1$ we show that our algorithm returns the one with $q$ hops.

After $k$ iterations the set $X_{Set_{archive}}$ contains all paths starting from $v_i$ and containing at least $k$ distinct shared vertices ($k$ hops).

Similarly, after these same $k$ iterations the set $Y_{Set_{archive}}$ contains all paths containing at least $k$ shared ($k$ hops) vertices that end in the vertex $v_j$.

The algorithm terminates as soon as a shared vertex joining a prefix path in $X_{Set_{archive}}$ and a suffix path in $Y_{Set_{archive}}$ is found. Once the combined length of the prefix path from $X_{Set_{archive}}$ and the $Y_{Set_{archive}}$ reaches $q$, the path with $q$ hops will be identified and algorithm terminated. Therefore, the algorithm will terminate with the path containing the minimum number of hops and will not continue to explore paths containing larger number of hops. (end of proof)

### 4.4.4 Algorithm Complexity and Comments

In our algorithm each network site can be assumed to contain only those vertices that it shares with other participating network sites. Therefore, the algorithm complexity will depend only on the count of the shared vertices. If this is much smaller than the total number of vertices in the combined graph then we will need to exchange much fewer messages among the participating sites. We give below an expression for the number of messages that need to be exchanged for finding a path between two vertices.
1. There are a total number of $m$ vertices in the complete global graph $G$.

2. There are $n$ subgraphs, $D_1 \ldots D_n$, residing at $n$ different network sites.

- **Cost Model # 1:**
  In this cost model we count the number of messages, $N_m$, that must be exchanged among all the participating sites. In each iteration of the algorithm given in section 4.4.1 we send $XSet_m$ and $YSet_m$ in one message to every local site. We know that each successive iteration of the algorithm causes $XSet_{archive}$ and $YSet_{archive}$ to contain all paths containing one more hop than the sets contained in the previous iteration. Therefore, in the worst case the maximum number of iterations will be equal to the number of hops in the longest path in the global graph $G$. Since each shared vertex can appear at most in a path, the maximum number of iterations required will be:

$$n \times (1 + q)$$  \hspace{1cm} (4.11)

where $q$ is the number of shared vertices in the system. The above results shows that the number of message that need to be exchanged among the sites is dependent on the number of sites and the number of shared vertices and is not dependent on the size of the subgraphs at each site. This is very significant because it shows that as the sizes of the individual subgraphs grow, the communication complexity of our algorithm would remain unaffected.

- **Cost Model # 2:**
  In this model we examine a weighted sum of the number of messages exchanged and
the number of local operations performed. For each exchanged message, this algorithm finds some paths within a local subgraph. Finding a path in a graph containing $l$ vertices requires $O(l)$ time in the worst case. If there are $p$ paths to be found on one site in an iteration then this cost is $p \cdot O(l)$ computations. Therefore, the total cost for the algorithm will be

$$a \cdot n(1 + q) + n \cdot b \cdot O(pl)$$  \hspace{1cm} (4.12)

where $a$ and $b$ are the weights representing the relative costs of exchanging a message and performing a local operation. Here again the above results shows that the number of message that need to be exchanged among the sites is not dependent on the size of the subgraphs at each site but the computational cost of local computations would grow with the subgraph size at each individual site but our decomposable version has an advantage in this regard also over the transport, concatenate, and then run the traditional algorithm.

- **Cost Model # 3:**

  In this model we examine a weighted sum of the number of messages exchanged and the number of local operations performed, while discounting the effects of messages and operations that can be executed simultaneously at different sites. Therefore, the total cost for the algorithm will be

  $$[a \cdot n(1 + q) + n \cdot b \cdot O(pl)]/n = a \cdot (1 + q) + b \cdot O(pl)$$  \hspace{1cm} (4.13)

  Where $a$ and $b$ are the weights representing the relative costs of exchanging a message
and performing a local operation and \( n \) is the number of sites participating in the computation.

The above analyses show that according to the cost model 1, the maximum number of messages a site would need to exchange is proportional to the path with the longest hops in the global graph. If there is a path between the vertices of interest then the algorithm will stop before the worst case scenario and when there is no path between the vertices the algorithm will terminate in the worst case scenario. From the time perspective (cost model 3) the algorithm will be executed in time proportional to the communication required with any one other site. That is, if the messages are transmitted in parallel and executed simultaneously at the local sites then the elapsed time of the algorithm does not depend on the number of sites with which the site forms the collective global graph \( G \) and engages in a collaborative computation.

### 4.5 Conclusions

We have demonstrated that it is possible to very efficiently find paths across a global graph formed by overlapping subgraphs stored on possibly local sites of a wide-area network. These algorithms are decomposable at run time depending on the collection of sites with which a site can connect and communicate and also the set of shared vertices among their subgraphs. We have analyzed the complexity of our algorithms from the perspective of three different cost models that take into account the communication cost across the sites of a communication network. It turns out that these algorithms can be executed with very low communication overhead. This is very desirable for the local data sites that must preserve their security
and privacy and also execute in shortest possible time. In almost all situations the cost of exporting subgraphs to a common site and then rebuilding the global graph at a local site for processing will take more communication energy and time than what is needed by the algorithms presented here. Also, these algorithms are independent of the manner in which the subgraphs may interface and overlap with each other (via the sets of shared vertices among them) and also the set of sites that may be available for participation in a particular instance of a computation. This guarantees the flexibility data site requires because of uncertainty about the other data sites that may exist in its neighborhood at any given time. These algorithms also preserves the privacy and security of the data at individual sites by requiring transmission of only minimal information to other sites.
Chapter 5

Decomposable Algorithms for Computing MST

The minimum weight spanning tree problem is one of the most typical and well-known problems of combinatorial optimization. Efficient solution techniques had been known for many years. However, in the last two decades asymptotically faster algorithms have been invented. Each new algorithm brought the time bound one step closer to linearity and finally Karger, Klein and Tarjan proposed the only known expected linear-time method. Classical algorithms make use of more advanced data structures and appear to be more complicated to implement. Most authors and practitioners refer to these but still use the classical ones, which are considerably simpler but asymptotically slower. MST is important and very popular for the following reasons:

1. It has a direct applications in the design of computer and communication networks, power and leased-line telephone networks, wiring connection, links in a transportation
network, Piping in a flow network, etc.

2. It offers a method of solution to other problems to which applies less directly, such as network reliability, clustering and classification problems.

3. It occurs as a subproblem in the solution of other problems. For example MST algorithms are used in several exact and approximation algorithms for the traveling salesman problem, the multi-terminal flow problem, the matching problem and the capacitated MST problem.

4. Efficient solution techniques capable of solving large instances exist.

Numerous authors have studied different forms of the classical algorithms and proposed efficient implementation techniques [69, 57, 58, 46, 52]. Table below summaries and presents the existing MST algorithms

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>(Data Structure)</th>
<th>Time Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boruvka</td>
<td>Disjoint set Union Algorithm</td>
<td>$O(m \log n)$</td>
</tr>
<tr>
<td>Kruskal</td>
<td>Disjoint set Union Algorithm</td>
<td>$O(m \log n)$</td>
</tr>
<tr>
<td>Prim</td>
<td>-</td>
<td>$O(n \times n)$</td>
</tr>
<tr>
<td>Prim</td>
<td>Binary Heap</td>
<td>$O(m \log n)$</td>
</tr>
<tr>
<td>Prim</td>
<td>d-Heap</td>
<td>$O(nd \log n + m \log n)$</td>
</tr>
<tr>
<td>Prim</td>
<td>F- Heap</td>
<td>$O(n \log n + m)$</td>
</tr>
<tr>
<td>Yao</td>
<td>Heap of size k</td>
<td>$O(m \log n)$</td>
</tr>
<tr>
<td>Cheriton and Tarjan</td>
<td>Doubly linked queue</td>
<td>$O(m \log \log n)$</td>
</tr>
<tr>
<td>Fredman and Tarjan</td>
<td>F-heap, doubly linked queue</td>
<td>$O(m \beta(m, n))$</td>
</tr>
<tr>
<td>Gabow et al.</td>
<td>F-heaps with packets, disjoint set union</td>
<td>$O(n \log n + m)$</td>
</tr>
<tr>
<td>Karger</td>
<td>Randomization</td>
<td>$O(m)$</td>
</tr>
<tr>
<td>Karger et al.</td>
<td>Randomization, recursion,</td>
<td></td>
</tr>
</tbody>
</table>

Table 5.1: Summary of the Existing MST Algorithms
Where \( n \) is the number of vertices, \( m \) is the number of edges in a graph \( G \), and

\[
\beta(m, n) = \min\{i|\log^i n < m/n\} \quad \text{and} \quad \log^0 n = n
\]

(5.1)

In this chapter we present self-decomposing versions of two graph algorithms for computing shortest paths spanning tree and minimum spanning trees to graphs stored in distributed databases. An objective of our algorithms is their self-decomposability for any way in which the graph may be distributed across the network. Another objective of the algorithms is to perform their tasks by minimizing the number of messages exchanged among the participating sites.

5.1 Decomposable Algorithm for Computing Shortest Paths Spanning Tree

Here we start by presenting the outline of the decomposable algorithm adapted from the Dijkstra’s algorithm

**Input:** a connected graph with a vertex set \( V \) and edge set \( E \) divided into \( m \) parts \( SG_1(v_i, E_1), SG_2(v_j, E_2) \ldots SG_m(V_n, E_n) \), each part residing at a different site and any two subgraphs may share some vertices.

\( r \) a vertex of \( V \) (the tree root)

\( W \) weight functions on \( E_1, E_2, \ldots E_n \)

\( InTheTree[1 : m] \) global array initialized to \( F \).

**Output:** \( Parent[1 : m] \) global array of a shortest path spanning tree.
$Dist[1:m]$ global array of lengths of shortest paths from $r$ initialized to $\infty$.

Algorithm

$MessageCounter = 0$

$Parent[r] = 0$

For step = 1 to $m$ do

select a vertex $u$ that minimizes $Dist[u]$ over all $u$ such that $InTheTree[u] = F$

set $InTheTree[u] = T$

Send message to every $site_i$

Result = select $vertex_2$, weight from $SG_i$

where $vertex_1 = u$, and there is an edge between $vertex_1$ and $vertex_2$

$MessageCounter + = n$

while($R = $NextElement(Result))

$R$ is an object from local sites contains $vertex_2$ and weight

$NextElement$ is a function take element by element from the local site results

$a = R.vertex_2$ and $b = R.weight$

if $(Dist[a]) > (Dist[u] + b)$

$Dist[a] = Dist[u] + b$

$Parent[a] = u$

end if

end while

end procedure Distributed-Dijkstra
5.1.1 Algorithm Analysis and Comments

We analyze below the MST computing algorithm for its complexity from the perspective of the three cost models mentioned in chapter two Section 2.7 as the following:

• **Cost Model # 1**

   We assume that $SG_1, SG_2, \ldots, SG_n$ have a total of $m$ vertices labeled with $1, 2, 3, \ldots m$. We maintain a boolean array $InTheTree$ to keep track of the vertices in/not in the tree. We then select the next edge $uv$ to be added to the tree by maintaining an array $Dist[1 : m]$ at the algorithm initiating site where $Dist[v]$ is the distance from the root vertex $r$ to $v$ in the tree and $Dist[v] = \infty$ if $v$ is not yet in the tree. In the algorithm we have $m$ stages, and for each stage the querying site needs to send only one message to every other site. Therefore, a total of $m \times n$ messages will have to be exchanged to complete this algorithm. The total cost for the algorithm will be:

   \[ a \times m \times n \]  

   Where $a$ is the taken time to exchange a message. From the above results the total number of exchanged messages is dependent on the number of sites and the size of the global graph.

• **Cost Model # 2**

   For each stage there are $m$ exchanged messages, and for each exchanged messages this algorithm performs only one SQL query at the responding site. The total cost of the algorithm will be the cost of exchanged message plus the cost of performing query at
each local site. Therefore, the total cost of the algorithm will be:

\[ a \times m \times n + b \times m \times n \]  

(5.3)

Where \( a \) is the time taken to exchange a message and \( b \) is the average time taken to perform a simple SQL query at a site.

- **Cost Model # 3**

Since it is possible to perform one message at each participating site simultaneously, i.e. the cost in this case will be the cost of Cost Model # 2 taking into account the average number of messages that can be exchanged in parallel which is \( n \) in our case. Therefore, the total cost for this algorithm will be:

\[ \frac{(a \times m \times n + b \times m \times n)}{n} = (a \times m + b \times m) \]  

(5.4)

Where \( a \) is the time taken to exchange a message and \( b \) is the average time taken to perform an SQL query at a site. From the above results the number of exchanged messages is not dependent on the number of local sites.

### 5.2 Higher Granularity Algorithm for Computing MST

In this section self-decomposing algorithm for finding minimum spanning tree for a graph stored across a number of distributed databases is presented. A subgraph exists at each site of the network. Each site executes the same algorithm and exchanges messages with other
sites until the the minimum spanning tree is constructed. The total number of messages required is \( n(1 + k^2) \) where \( n \) is the number of sites and \( k \) is the average number of shared vertices at each site.

### 5.2.1 Fragment Definitions and Properties

Here we first introduce the fragment definitions.

**Definition 1** A fragment \( f \) of an MST is a subtree of the MST.

**Definition 2** An outgoing edge \( e \) of a fragment \( f \) is an edge that has one of its vertices in \( f \) and the other vertex out of \( f \).

**Property #1** Given a fragment of an MST, let \( e \) be the minimum weight outgoing edge of the fragment. Then joining \( e \) and its adjacent non-fragment vertex to the fragment yields another fragment of an MST. This result in the context of fragments has been shown in [46] and we briefly outline below their proof for this property.

Suppose the added edge \( e \) is not in the MST containing the original fragment. Then there is a cycle formed by \( e \) and some subset of the MST edges. At least one edge \( x \neq e \) of this cycle is also an outgoing edge of the fragment, so that \( W(x) \geq W(e) \) where \( W(x) \) is the weight associated with the edge \( x \). Thus, deleting \( x \) from the MST and adding \( e \) forms a new spanning tree which must be minimal if the original tree was minimal. Therefore, the original fragment including the outgoing edge \( e \) is a fragment of the MST.
5.2.2 The Decomposable Algorithm

We now present a self-decomposing version of an MST building algorithm for a graph that exists in parts across a network. The assumptions describing the situation are:

- The complete global connected graph $G$ is represented by $n$ distinct but possibly overlapping subgraphs. Each subgraph is embedded in a database $D_i$ residing at a different computer system (site) in the network. Each subgraph consists of a set of vertices and edges $SG_1(v_i, E_1), SG_2(v_j, E_2) \ldots SG_n(V_n, E_n)$ and $G = \bigcup_i \ SG_i$.

- Each subgraph $SG_i$ may share some vertices with other subgraphs. The set of vertices shared by $SG_i$ and $SG_j$ are: $V_{sh}(i, j) = V_i \cap V_j$.

- The set of all vertices that are shared by at least two subgraphs is: $Shared = \bigcup_{i,j} V_{sh}(i, j)$.

- Each edge $e$ of a subgraph $SG_i$ has a weight $W(e)$ associated with it.

- The weight of a tree in the graph is defined as the sum of the weights of the edges in the tree. Our objective is to find the global tree of minimal weight, that is, the global $MST$.

Algorithm Outlines

1. **Data Structure:** A table called *Links* is maintained at the coordinator site and it stores information about candidate edges for linking various fragments to each other in order to complete the $MST$. This table has one row and one column for each shared vertex in the global graph. Each shared vertex name in the table is a representative of all the fragments, on all the different sites, that contain this shared vertex. Each
entry in the table represents the site number and the weight of a potential edge that can link the row-fragment to the column-fragment. How these values are updated is given in the *Global Computations* paragraph below.

2. **Local Computations** The following steps are executed at each site on its locally resident subgraph. The goal is to create fragments within each subgraph at its local site.

- If the site has $r$ shared vertices then each shared vertex is initialized as a separate fragment. That is, $f_i = v_i, i = 1, 2, \ldots, r$ and $v_i$ is the $i^{th}$ shared vertex.

- Expand fragments $f_i$’s as follows

  - For each fragment $f_i$ find its minimum weight outgoing edge $e_i$ such that $e_i$ does not lead to a vertex already included in some other fragment.

  - From among all the outgoing edges select the one with least weight. That is,

    $$e_{\text{min}} = \arg\min_i W(e_i).$$

  - Add $e_{\text{min}}$ to the fragment that selected this edge as its minimum outgoing edge.

3. **Global Computations** Each site, after completing its local computations, sends a message to the coordinator site informing it that the site is ready for global level coordination. After all the participating sites are ready the coordinator site performs the following steps to generate the global MST. In the final state each site knows the edges in its local subgraph that are included in the global MST. A complete MST is not generated at any single site.
• At each site, find the minimum-weight outgoing edge from each fragment to every other locally resident fragment. Each fragment is identified by the unique shared vertex contained in it. This step generates a tuple <site-number, from-fragment, to-fragment, weight> for each fragment at the site. All these tuples are sent to the coordinator site.

• Update the cell Links(row, column) table with the above tuples received from all the sites as follows.

  – Include all the received tuples from the local sites in a set S where
    row = from-fragment; and
    column = to-fragment

  – From the set S select the tuple t with the minimum weight value.

  – Assign this tuple to Links(row, column).

• Repeat the following steps till the Links table contains only one row and one column:

  – Select in Links the cell with minimum value. This edge links its row-fragment (f_{row}) to its column-fragment (f_{col}) and is now selected to be in the global MST. This step also means that the fragments f_{row} and f_{col} are now merged into a new fragment, called f_{row−col}.

  – Update the Links tuple as follows:

    * Create a new row and a new column in Links for the fragment f_{row−col} and delete the rows and columns for the original two fragments f_{row} and f_{col}. The value for the cells in the new row and column can be determined
as:

- For every shared vertex \( d \) in the table if the minimum weight of outgoing edge from \( d \) to \( f_{row} \) is \( v_i \) and to \( f_{col} \) is \( v_j \) and \( v_i < v_j \) then the minimum weight of outgoing edge from \( d \) \( f_{row-col} \) will be \( v_i \).

4. End Algorithm.

5.2.3 Algorithm Correctness

An algorithm that grows and combines fragments until an \( MST \) for the graph is formed is given by Gallager et al. in [46]. Their algorithm is designed for a parallel processing environment in which one processor can be assigned to work at each vertex of the graph. Since our situation is different, we show below that our algorithm for growing and combining fragments will yield the global \( MST \). The set \( Shared - Vertices \) contains all those vertices of the subgraphs that exist in more than one subgraph. We grow fragments starting from each member of \( Shared - Vertices \) that exists at any site. That is, if \( k \) shared vertices occur in a subgraph at a site then the site will locally generate three distinct fragments each containing one and only one shared vertex. Then we combine these fragments by the global computations described in section 3 above.

We now show that (i) locally generated fragments are parts of the minimum spanning tree and (ii) the edges selected to combine local fragments are also part of the global \( MST \).

Assertion: Each locally generated fragment is a part of the global MST.

Proof: The algorithm ensures that each fragment contains one and only one vertex that is shared with subgraphs at other sites. All the other vertices included in a fragment are
completely local to the subgraph and the site on which the fragment resides. Therefore, in
the global graph there does not exist an edge that connects a vertex in a local fragment to a
vertex in another fragment that resides at some other site. This implies that edges between
vertices of a fragment cannot be replaced by edges existing in some other subgraph residing
at some other site.

Therefore, as long as a fragment is an MST of the local subgraph, it will be a fragment
of the global MST.

Now we show that a fragment is part of the MST of its local subgraph i.e. each added
minimum outgoing edge $e$ is part of the MST.

Suppose the added edge $e$ is not in the MST containing the original fragment. Then
there is a cycle formed by $e$ and some subset of the MST edges. At least one edge $x \neq e$
of this cycle is also an outgoing edge of the fragment, so that $W(x) \geq W(e)$ where $W(x)$ is
the weight associated with the edge $x$. Thus, deleting $x$ from the MST and adding $e$ forms
a new spanning tree which must be minimal if the original tree was minimal. Therefore, the
original fragment including the outgoing edge $e$ is a fragment of the MST.

**Assertion:** Edges selected to combine local fragments are part of the global MST.

**Proof:** Each fragment contains one and only one shared vertex. Therefore, joining
two fragments by an edge is equivalent to bringing two shared vertices into one connected
component of the MST. An edge that connects two shared vertices (and their respective
fragments) may exist on more than one site. Our algorithm examines all possible sites on
which the same two shared vertices (and their fragments) may be connected and selects the
minimum-weight edge from among all the candidate edges on all the sites.

Now, if this does not result in the MST then there must exist an edge on a site that links
two fragments such that it results in a lower weight MST. Suppose \( e_{ij} \) which is least weight minimum outgoing edge from fragment \( f_i \) to fragment \( f_j \) chosen from Links table is not the correct minimum outgoing edge from fragment \( f_i \) to fragment \( f_j \). Then there is another path from \( f_i \) to \( f_j \) with minimum weight less than the \( e_{ij} \) weight. Part of this path is an outgoing edge \( y \) from \( f_i \) with weight is less than \( e_{ij} \) which contradict with the assumption \( e_{ij} \) is the minimum outgoing edge from fragment \( f_i \).

5.2.4 Example

Let us consider the subgraphs in Figure 4.2. The results of local computations can be shown in the following figure Figure 5.1

![Figure 5.1: Local Fragments at Different Sites](image)

The global computations start with the Links table as shown below:
<table>
<thead>
<tr>
<th>$S_{ver tex}/S_{ver tex}$</th>
<th>1</th>
<th>4</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$\infty$</td>
<td>(1,1,8,12)</td>
<td>(2,9,6,6)</td>
</tr>
<tr>
<td>4</td>
<td>(1,1,8,12)</td>
<td>$\infty$</td>
<td>(2,4,5,9)</td>
</tr>
<tr>
<td>6</td>
<td>(2,9,6,6)</td>
<td>(2,4,5,9)</td>
<td>$\infty$</td>
</tr>
</tbody>
</table>

Table 5.2: Link Table Stage 1

From the Table 5.2 we find the minimum value is 6, it means that the fragment represented by a shared vertex 1 at site number 2 and the fragment represented by the shared vertex 6 should be combined. We refer to the new fragment resulted from the combination by (1,6).

The result of updating $Link$ table is the following:

<table>
<thead>
<tr>
<th>$S_{ver tex}/S_{ver tex}$</th>
<th>4</th>
<th>1,6</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>$\infty$</td>
<td>(2,4,5,9)</td>
</tr>
<tr>
<td>1,6</td>
<td>(2,4,5,9)</td>
<td>$\infty$</td>
</tr>
</tbody>
</table>

Table 5.3: Link Table Stage 2

By the same way from Table 5.3 the next minimum weight is 9 then the fragment represented by the shared vertex 4 and the one represented by the shared vertex 1,6 should be combined.

The result of updating the $Link$ table will be Table 5.4:

<table>
<thead>
<tr>
<th>$S_{ver tex}/S_{ver tex}$</th>
<th>4, 1,6</th>
</tr>
</thead>
<tbody>
<tr>
<td>4, 1,6</td>
<td>$\infty$</td>
</tr>
</tbody>
</table>

Table 5.4: Link Table Stage 3
5.2.5 Algorithm Complexity and Analysis

As we discussed before in chapter two Section 2.7. We derive below an expression for the number of messages that need to be exchanged for our algorithm dealing with the implicit set of tuples. Let us say:

1. There are a total number of $m$ vertices in the whole graph.

2. There are $n$ relations, $D_1 \ldots D_n$, residing at $n$ different network sites.

- **Cost Model # 1:**

  As we mentioned before in this cost model we count the number of messages, $N_m$, that must be exchanged among all the participating sites in order to complete the execution of the algorithm. In this case the complexity will be:

  - $n$ exchanged messages to perform local fragments.

  - $n$ exchanged messages to combine the local fragments

Then the total cost in this case will be:

$$2a * n$$  \hspace{1cm} (5.5)

where $a$ is the time taken to exchange a message. The above results shows that the number of messages that need to be exchanged among the sites is not dependent on the the number of the vertices in the local subgraphs. This is significant because it shows that as the sizes of the individual subgraphs grow, the communication complexity of our algorithm would remain unaffected.
- **Cost Model # 2:**

In this model we examine a weighted sum of the number of messages exchanged and the number of local operations performed. For each exchanged message, this algorithm performs one local computation at the responding site, i.e. the total cost will be the cost for exchanged messages plus the cost for performing a local computation at each local site. Therefore, the total cost for this algorithm will be:

\[ 2(a + b) * n \]  \hspace{1cm} (5.6)

Where \( a \) and \( b \) are the weights representing the relative costs of exchanging a message and performing a local computation. Again the above results shows that the number of message that need to be exchanged among the sites is dependent on the the number of local sites.

- **Cost Model # 3:**

In this model we examine a weighted sum of the number of messages exchanged and the number of local operations performed, while discounting the effects of messages and operations that can be executed in parallel, simultaneously at different sites i.e. the cost in this case will be the cost in Cost Model # 2 taking into account the average number of messages that can be exchanged in parallel which is \( n \) in our case. Therefore, the total cost for this algorithm will be:

\[ 2(a + b) * n/n = 2(a + b) \]  \hspace{1cm} (5.7)
Where \( a \) and \( b \) are the weights representing the relative costs of exchanging a message and performing a local computation. From the above results the number of the exchanged messages is not dependent on the number of local sites.

### 5.3 Conclusions

We have demonstrated that it is possible to execute graph operations for databases stored across wide-area networks. These algorithms are decomposable at run time depending on the set of shared vertices among the graph components stored at different network sites.

We examined the complexity of our algorithm from the perspective of cost models that take into account the communication cost across the sites of a wide-area network. It turns out that these graph algorithms can be computed without too much communication overhead. In some situations the cost of extracting a graph from a bigger database, exporting it to a common site and then rebuilding a local database for processing may take more time than what is required by the version of algorithms presented here. Also, this version of algorithms is independent of the manner in which the graphs interface with each other (the sets of shared vertices among them) and also the set of sites that may be available for participation in a particular instance of a computation. These algorithms can performs more efficiently than having to bring all the data at one site while still preserving the privacy and ownership value of individual databases.
Chapter 6

Decomposable Clustering Algorithm - Horizontal Partition

Clustering techniques play an important role in exploratory pattern analysis and unsupervised learning and are very useful in many applications. Most clustering algorithms require intensive computation even for a modest number of data points.

In this chapter we present the decomposable version of the well known $k$-means clustering algorithm. A coordinator site first determines all the databases and sites that should be involved in a clustering task and then communicates to them requests for results of some computations performed locally at each site. Only the results of these local computations are transmitted to the coordinator site, possibly followed by new requests for more local computations, until the global results are generated at the coordinator site.
6.1 Clustering Algorithms and Distributed Databases

The clustering problem aims at classifying a collection of data points into groups such that each point is, in terms of some distance metric, relatively close to other points in its own group and farther from the points in other groups. Clustering algorithms have been studied extensively [6, 7] and the algorithms can be classified primarily as being hierarchical and non-hierarchical. Hierarchical clustering algorithms can be divided into agglomerative and divisive methods. The agglomerative methods start from individual points as sub-clusters and then merge together the most similar sub-clusters at successively higher levels. The divisive methods start by considering the set of all points as a single cluster and at each successive level some cluster is divided into smaller clusters. Non-hierarchical clustering algorithms do not have a hierarchy of clusters and produce only a unique clustering of the points. In most non-hierarchical algorithms all the feature vectors, or data points, are presented to the algorithm once, or a few times. These algorithms tend to produce compact and hyper-spherical or hyper-ellipsoidal shaped clusters, depending on the distance metric used.

The problem we seek to address involves a horizontally partitioned dataset. That is, relational tables defined by an identical set of attributes exist at various sites of a network. Different sets of tuples populate the tables at individual sites. We want to cluster the data points contained in all the tables combined. A simple solution is to move all the databases at some single network site, concatenate the tables into one long table, and then run the clustering algorithm on this table. Another alternative is to perform some local computations at each site and collect only these summaries at some central site. These summaries can
then be used to arrive at a clustering result very close to the one obtained by the former approach of accumulating the databases at a single site.

From the perspective of clustering tuples in distributed databases there is a significant difference between the hierarchical and the non-hierarchical approaches. Hierarchical approaches need much more frequent references to individual points and sub-clusters and are therefore likely to have high communication cost in a distributed implementation. A non-hierarchical algorithm, such as the $k$-means algorithm, is likely to involve less communication overhead. In the following sections we present our formulation of a decomposable version of $k$-means clustering algorithm for horizontally partitioned datasets. This will be accomplished by exchanging a set of messages among the sites to get the result of clustering and some statistical summaries and then combine this results by in some principled fashion by the Coordinator site as depicted in Figure 6.1.

![Diagram](image-url)

Figure 6.1: A Synthesized Global Strategy
6.2 The Decomposable Algorithm

The \textit{k-means} clustering algorithm is very well known and suits many applications because of its efficiency in clustering large numerical and categorical data sets. We adapt this \textit{k-means} algorithm to situations where dataset consists of a number of databases stored at different sites across a wide-area network. Together, these databases represent horizontal partitions of a complete dataset, that is, each database consists of the same set of attributes and databases at individual sites contains different sets of tuples.

The main idea behind our distributed formulation is to perform local clustering operations at each site and then combine these local results to obtain a global result that is as close as possible to what would be obtained if all datasets were to be brought to a single site and made available as a unified table of data points to an execution of a \textit{k-means} algorithm. The closeness between clustering results is measured by the preservation of two principal characteristics of the results of \textit{k-means} algorithm, which are: (i) the clusters formed minimize an error quantity which reflects the average distance between a data point and the center of the cluster to which it belongs; and (ii) The clusters formed by the algorithm are hyper-spherical or hyper-elliptical in shape. Our objective in designing the distributed version of the algorithm is to preserve these two characteristics in the results.

For the \textit{k-means} clustering algorithm we define a quantity called \textit{TotalError} that is defined as:

\[ \text{TotalError} = \sum_{i,j} \text{distance}(p_{ij}, c_j) \]  \hspace{1cm} (6.1)

Where \( p_{ij} \) is the \( i^{th} \) data point in the \( j^{th} \) cluster and \( c_j \) is the center of the \( j^{th} \) cluster.

It is well known [82] that the clusters and cluster centers chosen by a \textit{k-means} algorithm
are located in such a way that they minimize the magnitude of the TotalError quantity as defined above. Our decomposable version is designed to minimize this same error for geographically distributed databases, and thus mimic the behavior of a k-means algorithm run on the implicit \( \mathcal{D} \).

### 6.2.1 Algorithm Outline

If \( D_i \) is the dataset residing on the site \( i \) then: (i) The complete dataset to be clustered \( \mathcal{D} = \bigcup_i D_i \); Each dataset is assumed to be a table with a fixed set of attributes but a different population of tuples; \( C_i \) is the set of cluster centers obtained if the k-means algorithm is run locally at the \( i^{th} \) site; The results of local clustering are transferred to a central site; and the set of all local clusters \( C = \bigcup_i C_i \).

The objective of the algorithm is to determine clusters contained in the dataset \( \mathcal{D} \) by using the information about cluster centers obtained from each individual site. Typically, if the goal is to find \( k \) clusters in \( \mathcal{D} \) then we would create \( p : p > k \) clusters at each individual site and transmit their results to the central site. The larger the \( p \), it is expected that the clusters obtained will be closer to the ones that would have been found in \( \mathcal{D} \).

A coarse outline of our algorithm is as follows:

1. Goal: Cluster the complete dataset \( \mathcal{D} \) into \( k \) clusters.

2. Choose a \( p : p > k \) to be the number of clusters to be created at individual database sites.

**Clustering Databases at Local Sites**
• Send Message to every computer site $D_i$ to cluster the dataset there into $p$ clusters using $k$-means algorithm locally.

• Send back to the central coordinating site the following information about each cluster

1. The cluster centers, $C_i$;

2. The population (Number of points) in each cluster; and

3. The radius within each cluster.

If there are $n$ participating sites then we will obtain $n \times p$ cluster center points at the coordinating site.

**Clustering $C$, the Set of Local Cluster Centers**

Each center of a cluster obtained from a local site (a total of $n \times p$ of them) is treated as a data point and is to be clustered with other similar points in this phase. We have defined clustering error in equation 6.1 as the summation of distance between each point of $D$ and the cluster center to which it belongs. In a distributed environment we don’t have access to all the points of $D$. therefore, we define an estimate, called Estimated Clustering Error (ECE), of the total distance between a data point $c$ and its cluster center (to be found in this phase of the algorithm), weighted by the population associated with $c$. The coordinating site now runs a modified version of the $k$-means algorithm on the globalized cluster centers in set $C$ as follows:

1. Randomly choose $k$ of the $n \times p$ points in set $C$ as candidates for final cluster centers.

2. For each data point $P_i$ having a coordinate $(x_{i1}, x_{i2}, \ldots, x_{im}) \in C$ do
(a) For each cluster $j$ having a coordinate $(c_{j1}, c_{j2}, \ldots, c_{jm})$ do

i. Compute the weighted mid-point $t_j$ between a center $c_j$ and a point $P_i$ as:

$$
 t_j = \left( \frac{(a_i \cdot x_{i1} + b_j \cdot c_{j1})}{(a_i + b_j)}, \frac{(a_i \cdot x_{i2} + b_j \cdot c_{j2})}{(a_i + b_j)}, \ldots, \frac{(a_i \cdot x_{im} + b_j \cdot c_{jm})}{(a_i + b_j)} \right) \tag{6.2}
$$

where $a_i$ is the population of the data point $P_i$, and $b_j$ is the population of the cluster center $j$.

ii. Compute the contribution to the total clustering error if the data point $P_i$ is added to the center $c_j$ as:

$$
 ECE = \text{dist} (\text{center}(j), t_j) \cdot b_j + \text{dist} (p_i, t_j) \cdot a_i \tag{6.3}
$$

iii. Cluster data point $P_i$ to the cluster with minimum $ECE$

The difference between the above and the traditional clustering algorithm lies primarily in the way the membership of a point in a potential cluster is decided. Here we make this decision to minimize the potential contribution a point would make to the total clustering error when examined for its inclusion in all possible candidate clusters. This decision is weighed by the population associated with each point, which is actually a cluster center from a local database. The next phase is to recompute the cluster centers.

**Recompute Cluster Centers**

For each cluster $j$

1. Find the total population of cluster $j$. This is also as $b_j$ described above.
2. Find all the points $P_i$ that belong to this cluster $j$.

3. Recompute the new center of $j$.

$$\text{center}(j) = ((x_{11} * a_1 / b_j + x_{21} * a_2 / b_j + \ldots + x_{m1} * a_m / b_j), (x_{12} * a_1 / b_j + x_{22} * a_2 / b_j$$

$$+ \ldots + x_{m2} * a_m / b_j), \ldots, (x_{1m} * a_1 / b_j + x_{2m} * a_2 / b_j + \ldots + x_{mm} * a_m / b_j)) \quad (6.4)$$

where $a_i$ is the population of $P_i$, and $b_j$ is the population of cluster $j$.

The last two of the above three steps are repeated until the decrease in the estimated cluster error remains below a threshold for a number of iterations. In this phase also there is a deviation from the traditional $k$-means algorithm. We compute the new centers such that they are weighed by the respective populations associated with the points (centers from local sites) included in each cluster.

The population weight related adaptations in the previous two steps can be shown to result in clusters that actually minimize the estimated cluster error. It can also be shown experimentally that the actual cluster error and the estimated cluster error become closer as $p$, the number of centers obtained from the local sites, increases. The results obtained with our algorithm on a constructed dataset are demonstrated below.

### 6.2.2 Example

We consider local databases consisting of points in a 2-dimensional space. The local databases from three sites are shown in Figure 6.2, Figure 6.3, and Figure 6.4. If they were to be collected at a single site, their layout would appear as shown in Figure 6.5.
During the first phase of the algorithm, clustering is performed at each local site using \textit{k-means} algorithm. We obtained a total of fifteen local cluster centers at the central coordinating site. Figure 6.6 shows the locations of these cluster centers. These fifteen points, based on their population and radius information, are used to form global clusters. We illustrate some characteristics of this algorithm when we seek to cluster these 15 points into six clusters. This is only an illustration with the test dataset and normally one would try to first determine the ideal number of clusters for a set of points and then generate those many clusters. The information collected at the central site from the local databases can be summarized as shown in the table below.

First we run a simple \textit{k-means} algorithm, without applying any weighing derived from the populations at the cluster centers, to create six clusters from these fifteen points. The result is shown in Figure 6.7 below. The linear boundaries around the points show the cluster boundaries as determined by the algorithm.
Figure 6.3: Data at Site-2

<table>
<thead>
<tr>
<th>$x$</th>
<th>$yy$</th>
<th>Population</th>
<th>Radius</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.03</td>
<td>52.11</td>
<td>37</td>
<td>1.2</td>
</tr>
<tr>
<td>52.19</td>
<td>36.68</td>
<td>35</td>
<td>1.8</td>
</tr>
<tr>
<td>38.26</td>
<td>9.018</td>
<td>33</td>
<td>1.9</td>
</tr>
<tr>
<td>20.85</td>
<td>24.54</td>
<td>40</td>
<td>2.1</td>
</tr>
<tr>
<td>4.77</td>
<td>4.08</td>
<td>25</td>
<td>2.4</td>
</tr>
<tr>
<td>32.06</td>
<td>17.85</td>
<td>12</td>
<td>2.0</td>
</tr>
<tr>
<td>43.22</td>
<td>12.03</td>
<td>33</td>
<td>2.5</td>
</tr>
<tr>
<td>10.43</td>
<td>7.22</td>
<td>28</td>
<td>2.2</td>
</tr>
<tr>
<td>58.18</td>
<td>37.86</td>
<td>62</td>
<td>2.3</td>
</tr>
<tr>
<td>43.02</td>
<td>26.62</td>
<td>33</td>
<td>2.1</td>
</tr>
<tr>
<td>8.319</td>
<td>61.21</td>
<td>60</td>
<td>2.4</td>
</tr>
<tr>
<td>4.01</td>
<td>9.14</td>
<td>38</td>
<td>2.2</td>
</tr>
<tr>
<td>53.32</td>
<td>40.88</td>
<td>10</td>
<td>2.1</td>
</tr>
<tr>
<td>57.17</td>
<td>58.12</td>
<td>430</td>
<td>9.5</td>
</tr>
<tr>
<td>37.36</td>
<td>24.16</td>
<td>38</td>
<td>2.7</td>
</tr>
</tbody>
</table>

Table 6.1: Example of the Database Exists at the Coordinating Site

Figure 6.8 shows the results of clustering when the weights based on the populations of individual cluster centers are taken into consideration, as described in the algorithm description in the preceding section. It should be noted that the point at the top right corner is now clustered by itself instead of being clustered along with some others in Figure
6.7 (simple \textit{k-means} algorithm). This is done because the single point has a large population and if merged with other points then a larger contribution to the clustering error will be made. The normal \textit{k-means} algorithm, working on a collection of all points brought at a single site, would also result in this cluster being separate from other points because it then minimizes the total error.

\subsection{Consequences of Using Radius and Population}

The difference between our version and the traditional clustering algorithm lies primarily in the way the membership of a point in a potential cluster is decided. In our algorithm version we make this decision to minimize the potential contribution a point would make to the total clustering error when examined for its inclusion in all possible candidate clusters. This decision is weighed by the population and the radius associated with each point, which
is actually a cluster center from a local database.

We have defined clustering error in equation 6.1 as the summation of distance between each point of $\mathcal{D}$ and the cluster center to which it belongs. In a distributed environment we don’t have access to all the points of $\mathcal{D}$, therefore, we define its estimate, called Estimated Clustering Error (ECE), as the estimated total distance between a data point $c$ and its cluster center, weighted by the population associated with $c$.

The advantages of using population weight related adaptations in our version are to minimize the estimated cluster error and to make the estimated cluster error more close to actual cluster error.

The radius is used as follows. When we compute the distance between from a point to some other point, instead of considering the Euclidean distance, we consider this distance in terms of the number of standard deviations away from the originating point. Using population and radius numbers as shown in the table above. The advantage of using radius during the last two phases of the distributed version of the clustering algorithm is to minimize
the estimated cluster error as the following: If the distance is measured in terms of standard
device units then for a highly scattered cluster the contribution to cluster error is reduced
for points farther away from the cluster center. It can, therefore, include more points within
a cluster without adding much to the estimated clustering error.

Using population and radius numbers as shown in the table above, we get the same
clusters as shown in Figure 6.8. However, we increase the radius of the large population
point at the top right corner. We then get a clustering as shown in Figure 6.9 below.

6.2.4 Algorithm Analysis and Comments

As we discussed before in chapter two Section 2.7 the cost of working with implicitly specified
set of tuples can be measured in various ways. One cost model computes the number of
messages that must be exchanged among various sites. Complexity for distributed query
processing in databases has been discussed in [89] and this cost model measures the total
data transferred for performing a local computation. We derive below an expression for the
number of messages that need to be exchanged and the local computations at each local site for our algorithm dealing with the implicit set of tuples. Let us say we have the following:

1. There are $n$ relations, $D_1, D_2, \ldots, D_n$ resides at different network sites.

2. There are $m$ attributes in each relation.

3. The average number of tuples at each site is $r$.

4. The average number of centers obtained from each site is $p$, $p << r$.

- **Cost Model # 1:**

  The messages are exchanged in a sequential manner, that is, one site is asked for its cluster centers, an answer is obtained, and then the request is sent to the next participating database. In this case the complexity will be $n$ exchanged messages. If $a$
Figure 6.8: Clustering Using population Weighted Algorithm

is the time taken to exchange a message then the cost of our algorithm will be:

\[ a \times n \] \hspace{1cm} (6.5)

The above results shows that the number of message that need to be exchanged among the sites is not dependent on the size of the database at each site. This is significant because it shows that as the sizes of the individual databases grow, the communication complexity of our algorithm would remain unaffected.

- **Cost Model # 2:**

In this model we examine a weighted sum of the number of messages exchanged and the number of local operations performed. For each exchanged message, this algorithm performs the following internal computation

- resources for executing \textit{k-means} with \( r \) tuples at each site.
Figure 6.9: If Point u has a higher populations and higher radius

- resources for latter two phases of our algorithm to run with $n \times p$ data points

i.e. the total cost will be the cost for exchanged messages plus the cost for performing a local computation at each local site. Therefore, the total cost for this algorithm will be:

$$a \times n + \text{the internal computation}$$  \hspace{1cm} (6.6)

where $a$ is the weight representing the relative costs of exchanging a message. Here again the above results shows that the number of message that need to be exchanged among the sites is not dependent on the size of the database at each site but the computational cost of internal computation would grow with the database size at each individual site but our decomposable version has an advantage in this regard also over the transport, concatenate, and then cluster alternative.

- **Cost Model # 3:**
The messages can be exchanged in parallel, that is, requests can be sent to all the participating sites at the same time and then continue after all the responses have been received. In this case we will require time for only one exchanged message (though, the message taking the longest time). In this case the complexity will be:

- one exchanged messages,

- Internal Computation

  * resources for executing \( k\)-means with \( r \) tuples at each site.

  * resources for latter two phases of our algorithm to run with \( n \times p \) data points.

Therefore, the total cost for this algorithm will be:

\[
a + \text{the internal computation} \tag{6.7}
\]

where \( a \) is the weight representing the relative costs of exchanging a message.

On the other hand if we move all the data to one site and then run \( k\)-means algorithm the total complexity will be the communication time of moving all the data to one site. The computational resources will be needed to concatenate the data tables plus the complexity of running \( k\)-means algorithm with \( n \times r \) tuples. The intuitive sense is that if the data size at each site, \( r \), is above some threshold then it is advantageous to perform the clustering using the distributed algorithm implementation. Therefore, as the databases become larger, the advantage from using our algorithm is greater.
6.3 Conclusion

In this chapter we have presented a version of the $k$-means clustering algorithm to perform clustering with horizontally partitioned datasets that are geographically distributed. The algorithm seeks to meet the characteristic properties by minimizing the clustering total error achieved by the $k$-means algorithm in the clusters that it forms. Our distributed version of the algorithm achieves the same characteristics by minimizing the clustering error. We use the information about the clusters formed at local sites to determine what the constitution of the global clusters should be. Population and radius of each local cluster is used to minimize the clustering error. It has been demonstrated with example datasets results showing the convergence and closeness properties of our algorithm are also being prepared for reporting in a larger version of this algorithm. We have demonstrated that very close clustering can be performed by transferring only some information from each site and there is no need to transfer entire datasets to a central location in order to perform clustering with the collective dataset.
Chapter 7

Decomposable Clustering Algorithm - Vertical Partition

As we discussed previously, in chapter 6, our approach to solve the clustering problem with distributed databases is to adapt $k$-means algorithm. The algorithm we presented in chapter 6 can work only with data in horizontally partition form. In this chapter we present a decomposable clustering algorithm for a dataset that is partitioned either horizontally or vertically across various sites of a network. We present algorithms that can decompose themselves at run-time into sub-computations to be performed at local database sites. The decomposition of computations must match the distribution of data elements across various sites of a wide-area network and the algorithm should then be able to compose the local results to arrive at global results.

In this decomposable version of the decomposed clustering algorithm a coordinator site first determines all the databases and sites that should be involved in a clustering task and then communicates to them requests for results of some computations performed locally at
each site. Only the results of these local computations are transmitted to the coordinator site, possibly followed by new requests for more local computations, until the global results are generated at the coordinator site.

### 7.1 Decomposition of the k-Means Algorithm

We explain below various aspects of the decomposable version of the well-known *k-means* clustering algorithm. One desired characteristic of our distributed formulation is that it should perform as much computation as possible at each local site and exchange only the minimum amount of information using the minimum number of messages among the participating sites.

As we discussed in chapter 6 for the *k-means* clustering algorithm we define a quantity called *TotalError*

\[
TotalError = \sum_{i,j} distance(p_{ij}, c_j)
\]  

(7.1)

where \(p_{ij}\) is the \(i^{th}\) data point in the \(j^{th}\) cluster and \(c_j\) is the center of the \(j^{th}\) cluster. It is well known [82] that the clusters and cluster centers chosen by a *k-means* algorithm are located in such a way that they minimize the magnitude of the *TotalError* quantity as defined above. Our decomposable version is designed to minimize this same error for geographically distributed databases, and thus mimic the behavior of a *k-means* algorithm run on the implicit \(D\).

If \(D_i\) is the dataset residing on the site \(i\) then: (i) The complete dataset to be clustered \(D\) is implicitly specified by the explicit \(D_1, D_2, \ldots D_n\). If an explicit \(D\) were to be generated from the \(D_i\)s, the process would have been mediated by the attributes shared among the
$D_i$'s. Let us say the set of attributes contained in relation $D_i$ is represented as $A_i$. For a pair of databases $D_i$ and $D_j$ the corresponding sets of attributes $A_i$ and $A_j$ may have a set of shared attributes given by $S_{ij}$ such that

$$S_{ij} = A_i \cap A_j. \quad (7.2)$$

To facilitate pattern discovery in the implicitly described set of tuples of $\mathcal{D}$, we define a set $\mathcal{S}$ that is the union of all the intersection sets defined above. That is,

$$\mathcal{S} = \bigcup_{i,j, i \neq j} S_{ij} \quad (7.3)$$

The set $\mathcal{S}$, in effect, contains all those attributes that occur in more than one $D_i$.

We define a relation called **Shareds** on the attributes in the set $\mathcal{S}$. This relation, **Shareds**, contains tuples corresponding to all possible combinations of values for the attributes in $\mathcal{S}$.

It is assumed that any site initiating a clustering task knows what other sites are participating in the computation and also the sets of attributes present at each of these sites. We now present a decomposable version of the $k$-means clustering algorithm for geographically distributed databases.

### 7.1.1 Decomposable Algorithm

The goal of our decomposable algorithm is to achieve with the distributed collection of $D_i$s the same result that a $k$-means algorithm would have achieved with the $\mathcal{D}$, if it were
to be generated from the $D_i$s. There are some issues that need to be addressed by an implementation of a $k$-means algorithm and we address them in the following sections.

**How Many Clusters?**

Sometimes a user may know the number of clusters that should be generated with the data. If that is not known and needs to be determined by repeated experimentation with data then there are many different ways in which one can determine the number of clusters a dataset should be divided into. We consider one such approach here and compare its implementations for the distributed and non-distributed versions of the $k$-means algorithm. Intuitively, this method involves making a plot of total clustering error as in equation 7.1 on y-axis versus the number of clusters on the x-axis, and then selecting the *most flat* portion of this graph to determine the number of clusters. Intuitively, we select that number (of clusters) for which the clustering error would change the least if the number were to be decreased. Formally, we describe this approach as follows.

Each local site $S_i$, $i = 1, 2, \ldots, n$ executes the following steps to determine the suitable number of clusters for its $D_i$.

1. Using $k$-means algorithm, cluster the local database $D_i$ for $k$ taking values from 2 to some value $s$ chosen by the user.

2. For each value of $k$ compute the *TotalError* as defined in equation 7.1 above. It is the sum of distances for all data points from the centers of clusters to which each point belongs.
3. Define $\Delta T$ as the difference between the maximum and minimum values of $TotalError$

$$\Delta T = \text{Max}_{TotalError} - \text{Min}_{TotalError}$$  \hspace{1cm} (7.4)

4. We now find the longest interval of contiguous $k$ values in which the $TotalError$ quantity remains relatively constant. One way of finding such an interval is given below:

- Define $\delta$ as a very small value
- For counter $j$: 1 to $s$ do
  - $StartInterval[j] = j$
  - $EndInterval[j] = j$
  - $h = j$
  - While ($h < s$) do
    * $Error = \left[ TotalError(j) - TotalError(h) \right] / \Delta T$
    * If $Error < \delta$
      - $EndInterval[j] = h$
      - $h = h + 1$
    * Else $h = s$

- Find the longest interval for which the error is less than $\delta$ by examining $StartInterval[i]$ and $EndInterval[i]$ for all the intervals.

Choose the largest $k$ value from the longest relatively constant error interval as the number of clusters to be generated at the local site $S_i$ from its dataset $D_i$. 
Each local site generates a suitable number of clusters for its own $D_i$ and transmits centers for those clusters to the coordinating site. In general, we select the largest $k$ for which the change in error is minimal. This causes each local site to generate a relatively larger number of clusters. The coordinating site forms final clusters by merging cluster centers obtained from local sites. Therefore, erring on the side of larger number of clusters at the local sites would not affect the global results of the clustering algorithm.

**Clustering at Local Sites**

**Local Computations # 1** The following steps are executed at each site on its locally resident database. The goal is to cluster the local database and return some summaries to the coordinating site.

- Each site $S_i$ uses the method described in the preceding section to determine the number of clusters, $k_i$, that it should generate with the local $D_i$.

- Using *k-means* clustering algorithm, cluster the local database $D_i$ into $k_i$ clusters.

- Send back to the central coordinating site the following information:
  - The set of local cluster centers $\{C_{ij} : \text{it is the center of the } j^{th} \text{ cluster at the } i^{th} \text{ site}\}$.
  - $r_{ij}$ is the radius of the local cluster centered at $C_{ij}$ and is defined as the maximum of the distances between the cluster center and any of the points included in the cluster.
Globalizing Local Cluster Centers

We now describe how the join of the cluster centers from various local sites is computed at the coordinating site to get the globalized cluster centers. Let us consider an example in which Site 1 has a database containing attributes $A$ and $B$ and Site 2 has a database containing attributes $B$ and $C$. Together, the two sites describe an implicit database containing attributes $A$, $B$, and $C$. Figure 7.1 shows the data points of Site 1 in the $A - B$ plane and Figure 7.2 shows the data points of Site 2 on the $B - C$ plane. After the local clustering phase Site 1 returns cluster centers that lie in the $A - B$ plane and Site 2 returns cluster centers that lie in the $B - C$ plane. Each local site determines cluster centers that lie in its own plane. Figure 7.3 shows the join of the points in the two planes, resulting in the points in the 3-D space. Candidates for these global cluster centers can be generated by performing a join of the cluster center points from the two distinct planes. Some globalized cluster centers may not have many data points in implicit $D$ around them because, as explained in section 2.2 above, the set $Exclusions$ may have eliminated many tuples from $D$. Figure 7.4 shows the reduced set of points (form those in Figure 7.3 due to the conditions in the $Exclusions$ set.

Let us consider the example of two sites sending the following local cluster centers after executing $k$-means algorithms at their respective sites. The problem now is to determine when two values of a shared attribute, in this case $B$, may be considered identical for the purpose of performing a join of the two sets of local cluster centers. That is, should a value of 3.5 for $B$ from Site 1 be considered the same as the value 3.2 for $B$ determined by Site 2 as part of a cluster center coordinate? Cluster centers in a $k$-means algorithm adjust with
every iteration and may be somewhat different for different sets of tuples representing the same underlying process. Therefore, we need to match the values of shared attributes only approximately for determining the globalized cluster centers. Once the globalized cluster centers are processed and adjusted in future iterations the impact of approximation here would be eliminated.

<table>
<thead>
<tr>
<th>Cluster Centers from Site 1</th>
<th>Cluster Centers from Site 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>2.0</td>
<td>3.5</td>
</tr>
<tr>
<td>18.0</td>
<td>6.7</td>
</tr>
</tbody>
</table>
A formal approach that models the population and radius around each cluster center may be adopted to determine the margin around each cluster center but since we are generating only the candidates we adopt a simpler and quicker way to generate the globalized cluster centers. For each value $x$ of a shared attribute, we create a $x \pm \varepsilon$ window around it and whenever two windows overlap we consider the values to be identical for the purpose of generating join. For example, in the table shown above Site 1 has a cluster center at $(A=2.0, B=3.5)$ and Site 2 has a cluster center at $(B=3.2, C=19.0)$. If we replace values of $B$
throughout with windows $x \pm 0.5$ then $3.2 \pm 0.5$ and $3.5 \pm 0.5$ overlap, and we can generate a global cluster center at $(A=2.0, B=3.5, C=19.0)$ by choosing any of the two values as the candidate value for $B$.

The coordinating site executes the following procedure to determine the globalized cluster centers from the local centers:

- Select a value for $\varepsilon$.

- Perform a cross-product of the local cluster centers with $\varepsilon$ window built around each shared attribute.

For the example shown above the following globalized cluster centers will be generated:

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
</tr>
<tr>
<td>2.0</td>
<td>3.5</td>
<td>19.0</td>
</tr>
<tr>
<td>18.0</td>
<td>6.7</td>
<td>12.5</td>
</tr>
<tr>
<td>18.0</td>
<td>6.7</td>
<td>30.4</td>
</tr>
<tr>
<td>5.0</td>
<td>17.9</td>
<td>8.2</td>
</tr>
</tbody>
</table>

The procedure for **Global Computations # 1** will be the following. At the coordinating site execute the following steps

- Compute the join of all the local cluster centers to generate the set $C$ (the globalized cluster centers).

**Populations around Globalized Cluster Centers**

If we take tuples from smaller dimensional spaces of local $D_i$s and *Join* them, it would produce all possible combinations of local tuples, as mediated by the common values of the shared
attributes and illustrated by Figure 7.3. Some of these points should not be considered by our algorithm due to the conditions included in the Exclusions set, as explained in section 2.2 and illustrated in Figure 7.4. Therefore, all the globalized cluster center candidates may not have many points around them. We need to compute populations of data points, in implicit \( \mathcal{D} \), around each globalized cluster center, taking into account the exclusions dictated by the set Exclusions. This will help us decide which globalized cluster center candidates to keep and which ones to discard. We have presented in [11] a way of counting tuples that match some conditions in an implicit \( \mathcal{D} \) using some counts taken from explicit \( \mathcal{D}_s \)s that constitute the \( \mathcal{D} \). This count of tuples can also be done only for those tuples that satisfy certain conditions [11] as imposed by the Exclusions set. We illustrate this issue with the following example. Let us say site \( S_1 \) has a local cluster center at local point \( p_1 \) with radius \( r_1 \); site \( S_2 \) has a local cluster center at point \( p_2 \) with cluster radius \( r_2 \); and site \( S_3 \) has a local cluster at point \( p_3 \) with a cluster radius of \( r_3 \). At the coordinating site the points \( p_1, p_2, \) and \( p_3 \), when joined, produce a globalized cluster center \( P \). To count the population around \( P \) in the implicit space \( \mathcal{D} \), we consider a point of \( \mathcal{D} \) as belonging to the cluster centered at \( P \) as long as its distance in the projected space of \( S_1 \) is less than \( r_1 \); its distance from \( P \) in the projected space of \( S_2 \) is less than \( r_2 \); and similarly for \( S_3 \).

**Decomposable Algorithm for Population Computation:** For each globalized cluster center \( c \) we find population \( N_c \) of data points around it in \( \mathcal{D} \). For an implicitly stated set of tuples we have decomposed the counting process in such a way that each decomposed part can be sent as a request to an explicit database site and the responses composed to reconstruct the counts. The decomposition for obtaining the count \( N_c \) for each globalized
cluster center candidate \( c \in C \) is as follows:

\[
N_c = \sum_{tuples \text{ - } in \text{- } Shared} \prod_{t=1}^{n} (N(D_t)_{tuple \text{- } in \text{- } Shared})
\]  

(7.5)

Where \( n \) is the number of participating database sites (\( D_i \)s),

tuple-in-shared is a tuple selected by the coordinating site from the relation \( Shareds \) and sent as part of request to each local site;

and \( (N(D_t)_{tuple \text{- } in \text{- } Shared}) \) is the count of those tuples in \( D_t \) that meet the following three conditions:

1. The values of shared attributes of \( D_t \) in the local tuple are the same as in the tuple tuple-in-shared selected from the relation \( Shareds \).

2. The attribute values in the tuple tuple-in-shared do not violate any of the conditions in the \textit{Exclusions} set; and

3. the Euclidean distance between the local tuple at \( D_t \) and the local cluster center whose globalized version is \( c \), is less than the radius associated with the local cluster center.

For each sum-of-products term in the above expression the coordinating site sends a message to the corresponding local site to obtain the count of tuples satisfying the conditions of tuple-in-shared. A number of such count requests may be combined in a single message to reduce the number of messages exchanged.

The product, in the above expression, is performed for the counts obtained from all the \( n \) \( N(D_t) \)s (sites) for each tuple from \( Shareds \); and the summation is performed over the product values for all the tuples \( sh \)’s in the relation \( Shareds \). The coordinating site stores
the relation *Shareds* and sends out messages to individual sites to obtain various $N(D_i)$’s and it is assumed that each local site retains a copy of the set *Exclusions*.

The expression for $N_c$, therefore, simulates the effect of a *Join* operation on all the $n \, D_i$s without explicitly enumerating the tuples.

We can thus compute the population around each globalized cluster center candidate and then retain only those centers around which the points’ population is above certain threshold.

The second major **local computation** to be performed at individual sites is, therefore, the following. At each local site the following steps are executed on its resident database to help compute the population of points near each globalized center $c \in \mathcal{C}$ using the following steps:

- For each point $c \in \mathcal{C}$ do
  - For each tuple $sh \in Shared$ do
    - For each $D_i$ do
      * Compute $(N(D_i)_{tupl-in-shared})$ the number of tuples at $D_i$, as explained above and send it to the coordinating site.

- The coordinating site then computes $N_c$ as the summation of multiplication of all $N(D_i)_{tupl-in-shared}$’s which represent the population of points around $c$.

**Clustering of Globalized Cluster Centers**

We define $\mathcal{C}'$ to be the set obtained by subtracting from $\mathcal{C}$ the set of all those globalized local centers that do not have sufficient population of points around them. This set $\mathcal{C}'$ of
globalized cluster centers and the population of points in $\mathcal{D}$ around each one of them is now available at the coordinating site. This site considers these globalized versions of the local cluster centers as mere data points that are weighted by the populations of points in $\mathcal{D}$ surrounding each of them. The coordinating site then goes on to cluster these globalized local centers in such a way that an estimate of the total clustering error is minimized.

We have defined clustering error in equation 7.1 as the summation of distance between each point of $\mathcal{D}$ and the cluster center to which it belongs. In a distributed environment we don’t have access to all the points of $\mathcal{D}$, therefore, we define its estimate, called Estimated Clustering Error (ECE), as the estimated total distance between a data point $c$ and its cluster center (to be found in this phase of the algorithm), weighted by the population associated with $c$. The coordinating site now runs a modified version of the $k$-means algorithm on the globalized cluster centers in set $\mathcal{C}'$ as follows:

1. Randomly choose $k$ points in set $\mathcal{C}'$ as the candidates for final cluster centers.

2. For each data point $P_i \in \mathcal{C}'$ having a coordinate $(x_{i1}, x_{i2}, \ldots, x_{im})$ do

   (a) For each final cluster $j$ having a coordinate $(c_{j1}, c_{j2}, \ldots, c_{jm})$ do

       i. Compute the weighted mid-point $t_j$ between a final cluster center $c_j$ and a point $P_i$, weighted by populations around $P_i$s, as:

       $$t_j = \left(\frac{(a_i * x_{i1} + b_j * c_{j1})}{(a_i + b_j)}, \frac{(a_i * x_{i2} + b_j * c_{j2})}{(a_i + b_j)}, \ldots, \frac{(a_i * x_{im} + b_j * c_{jm})}{(a_i + b_j)}\right)$$

       (7.6)

       where $a_i$ is the population corresponding to point $P_i$, and $b_j$ is the popula-
tion of the cluster center $j$, and

$$b_j = \sum_{c \in \text{cluster}(j)} N_c$$  \hspace{1cm} (7.7)

ii. Compute the estimated contribution to the total clustering error if the data point $P_i$ is added to the final cluster $j$ as:

$$ECE = \text{dist}(\text{center}(j), t_j) \cdot b_j + \text{dist}(p_i, t_j) \cdot a_i$$  \hspace{1cm} (7.8)

iii. Include data point $P_i$ in that final cluster which results in minimum value for the estimated clustering error.

The difference between the above and the traditional clustering algorithm lies primarily in the way the membership of a point in a potential cluster is decided. Here we make this decision to minimize the potential contribution a point would make to the total clustering error when examined for its inclusion in all possible candidate clusters. This decision is weighed by the population associated with each point, which is actually a cluster center from a local database. The next phase is to recompute the final cluster centers, as per the iterations of the $k$-means algorithm. This part is performed entirely at the coordinating site and does not require any communication with the local sites.

**Recompute Final Cluster Centers**

For each cluster $j$

1. Find the total population of cluster $j$. This is the same as $b_j$ described above.

2. Find all the points $P_i$ that belong to this cluster $j$. 

3. Recompute the new center of cluster $j$. In this phase we have a deviation from the traditional $k$-means algorithm. We compute the new centers such that each point is weighed by the population of data points in $D$ that is associated with it.

$$center(j) = ((x_{11} * a_1 / b_j + x_{21} * a_2 / b_j + \ldots + x_{m1} * a_m / b_j), (x_{12} * a_1 / b_j + x_{22} * a_2 / b_j + \ldots + x_{m2} * a_m / b_j), \ldots, (x_{1m} * a_1 / b_j + x_{2m} * a_2 / b_j + \ldots + x_{mm} * a_m / b_j))$$  \hspace{1cm} (7.9)

where $a_i$ is the populations of $P_i$, and $b_j$ is the population of cluster $j$.

The last two of the above three steps are repeated until the decrease in the estimated cluster error remains below a threshold for a number of iterations.

In this phase also there is a deviation from the traditional $k$-means algorithm. We compute the new centers such that they are weighed by the respective populations associated with the points (centers from local sites) included in each cluster. The population weight related adaptations in the previous two steps can be shown to result in clusters that actually minimize the estimated cluster error.

### 7.1.2 Example # 1

We consider a test situation consisting of a global database containing points in a 3-dimensional (attributes A, B, and C) space and two local databases that are projections of the global database on two different 2-Dimensional planes. The local databases $D_1$ and $D_2$ from two sites are shown in Figure 7.1, and Figure 7.2. The database $D_1$ at site $S_1$ is the projection of global database on $A - B$ plane and the database $D_2$ at site $S_2$ is the projection of global database on $B - C$ plane. In an actual situation only the two local databases would
exist and the global database would only be implicit. For the test we have made the global database $D$ explicit in order to compare the performance of our algorithm. In this example the local cluster centers obtained from site $S_1$ are:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>10.5</td>
<td>7.5</td>
</tr>
<tr>
<td>10.5</td>
<td>5.5</td>
</tr>
<tr>
<td>6.0</td>
<td>5.0</td>
</tr>
<tr>
<td>3.67</td>
<td>22.67</td>
</tr>
<tr>
<td>5.0</td>
<td>24.0</td>
</tr>
<tr>
<td>3.27</td>
<td>5.36</td>
</tr>
<tr>
<td>22.0</td>
<td>28.6</td>
</tr>
<tr>
<td>23.22</td>
<td>25.56</td>
</tr>
</tbody>
</table>

and the local cluster centers obtained from site $S_2$ are:

<table>
<thead>
<tr>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.0</td>
<td>3.5</td>
</tr>
<tr>
<td>7.2</td>
<td>3.6</td>
</tr>
<tr>
<td>26.0</td>
<td>15.0</td>
</tr>
<tr>
<td>28.71</td>
<td>13.57</td>
</tr>
<tr>
<td>5.5</td>
<td>24.5</td>
</tr>
<tr>
<td>7.29</td>
<td>23.86</td>
</tr>
<tr>
<td>23.0</td>
<td>6.0</td>
</tr>
<tr>
<td>24.33</td>
<td>7.33</td>
</tr>
</tbody>
</table>

The join of the local cluster centers is used to obtain the candidates for the globalized cluster centers. After computing populations around these points, taking conditions in Exclusions into account, we get the following set of points as members of the set $C'$.
The coordinating site then clusters these points into four clusters, taking into account the population around each point in $\mathcal{D}$ and minimizing the weighted clustering error. The cluster centers thus obtained are:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>10.51</td>
<td>6.3163</td>
<td>24.238777</td>
</tr>
<tr>
<td>22.489</td>
<td>27.386</td>
<td>14.542</td>
</tr>
<tr>
<td>4.425</td>
<td>23.424</td>
<td>6.3103333</td>
</tr>
<tr>
<td>8.475</td>
<td>6.375</td>
<td>3.5549998</td>
</tr>
</tbody>
</table>

We then make the dataset $\mathcal{D}$ explicit and run the $k$-means algorithm to determine the four cluster centers that would have been obtained if we had moved all the data to a single site and performed a *Join* before running the algorithm. The resulting cluster centers are as follows:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>10.469</td>
<td>6.489</td>
<td>24.04</td>
</tr>
<tr>
<td>22.84</td>
<td>26.94</td>
<td>14.5</td>
</tr>
<tr>
<td>4.5</td>
<td>23.433</td>
<td>6.4</td>
</tr>
<tr>
<td>5.866</td>
<td>5.2833</td>
<td>3.65</td>
</tr>
</tbody>
</table>

The two sets of cluster centers are very close. The extent of divergence between the two sets of cluster centers can be estimated by the value of *clustering error* that the cluster centers result in. We ran the tests varying the number of clusters and measured the *clustering error*
for each case. The plots in Figures 7.5 and 7.6 show the total clustering error for cluster centers determined by our decomposable algorithm and by direct use of \( \mathcal{D} \). It can be seen that the difference between the error quantities is very small and follows the same pattern. The plots also show the estimated clustering error that is computed by the coordinating site to guide itself towards the final cluster centers. This quantity reduces faster than the actual clustering error but follows the same trend, and thus, can guide towards the minimum error cluster centers. Two sets of plots in different figures show the error quantities from two different runs that used different points as center seeds for the \( k\text{-means} \) algorithm.

![Figure 7.5: TotalError versus Number of Clusters in Distributed and traditional algorithms](image)

### 7.1.3 Example # 2

The case of a horizontally partitioned global database \( \mathcal{D} \) can be looked at as a special case of vertical partitioning. In horizontal partitioning an identical set of attributes is used at each
Figure 7.6: TotalError versus Number of Clusters in Distributed and traditional algorithms

local site. All attributes can, therefore, be considered as shared attributes and the above algorithm then applied. We show results of the decomposable algorithm with an example of a horizontally partitioned database. We run the same data in chapter 6 with our algorithm. The layout of points from the three local databases are shown in Figures 7.7, 7.8, and Figure 7.9 respectively. If the three databases were to be collected at a single site, the collective layout of their points would be as shown in Figure 7.10.

During the first phase of the algorithm, clustering is performed at local sites using \textit{k-means} algorithm. We obtained a total of fifteen local cluster centers at the central coordinating site. Figure 13 shows the locations of these cluster centers. Globalization of local cluster centers is not needed here because all attributes are shared by each database. These fifteen points, based on their population information, are used to form global clusters.

Just for illustration, we first run the \textit{k-means} algorithm on the set of local cluster centers,
without applying any weighing derived from the populations at the cluster centers, to create six clusters. The result is shown in Figure 14 below. The linear boundaries around the points show the cluster boundaries as determined by the algorithm.
Figure 7.9: Dataset at site-3

Figure 7.10: Data points at all three sites

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>Population</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.03</td>
<td>52.11</td>
<td>37</td>
</tr>
<tr>
<td>52.19</td>
<td>36.68</td>
<td>35</td>
</tr>
<tr>
<td>38.26</td>
<td>9.018</td>
<td>33</td>
</tr>
<tr>
<td>20.85</td>
<td>24.54</td>
<td>40</td>
</tr>
<tr>
<td>4.77</td>
<td>4.08</td>
<td>25</td>
</tr>
<tr>
<td>32.06</td>
<td>17.85</td>
<td>12</td>
</tr>
<tr>
<td>43.22</td>
<td>12.03</td>
<td>33</td>
</tr>
<tr>
<td>10.43</td>
<td>7.22</td>
<td>28</td>
</tr>
<tr>
<td>58.18</td>
<td>37.86</td>
<td>62</td>
</tr>
<tr>
<td>43.02</td>
<td>26.62</td>
<td>33</td>
</tr>
<tr>
<td>8.319</td>
<td>61.21</td>
<td>60</td>
</tr>
<tr>
<td>4.01</td>
<td>9.14</td>
<td>38</td>
</tr>
<tr>
<td>53.32</td>
<td>40.88</td>
<td>10</td>
</tr>
<tr>
<td>57.17</td>
<td>58.12</td>
<td>430</td>
</tr>
<tr>
<td>37.36</td>
<td>24.16</td>
<td>38</td>
</tr>
</tbody>
</table>
Figure 7.11: The Local Cluster Centers

Table 7.1: Example of the Database Exists at one Site

Figure 7.12: Clustering Using k-means Algorithms

Figure 7.13 shows the results of clustering when the weights based on the populations of individual cluster centers are taken into account to minimize the clustering error as described in 7.1 above. It should be noted that the point at the top right corner is now clustered by itself instead of being clustered along with some others in Figure 7.12 (simple k-means algorithm). This occurs because the single point has a large population and if merged with other points then a larger contribution is made to the clustering error. The normal k-means algorithm,
working on a collection of all points brought at a single site, results in this cluster being separate from other points because it then minimizes the total error.

![Population - Weighted and population Variance Weighted Algorithm Results](image)

Figure 7.13: Clustering Using population Weighted Algorithm

### 7.1.4 Analysis and Comments

The cost of working with implicitly specified set of tuples can be measured in various ways. One cost model computes the number of messages that must be exchanged among various sites. Complexity for distributed query processing in databases has been discussed in [89] and this cost model measures the total data transferred for answering a query. In our case the amount of data transferred is very little (statistical summaries) but the number of messages to be exchanged may grow rapidly. We derive below an expression for the number of messages that need to be exchanged for our clustering algorithm dealing with the implicit set of tuples. Let us say:

- There are $n$ relations, $D_1 \ldots D_n$, residing at $n$ different network sites.

- There are $k$ attributes in set $\mathcal{S}$. Each attribute in this set appears at more than one
site.

- There are $m$ distinct attributes in all the sets ($\bigcup_{i=1}^{n} A_i$) combined.

- There are $l$ possible discrete values for each attribute in set $S$.

- The number of globalized cluster centers in $C'$ is $p$.

The number of tuples in relation $Shareds$ is $l^k$ because it contains all possible combinations of values for its attributes.

### 7.1.5 Complexity

As we discussed before in chapter two Section 2.7 the cost of working with implicitly specified set of tuples can be measured in various ways. One cost model computes the number of messages that must be exchanged among various sites. We derive below an expression for the number of messages that need to be exchanged and the local computations at each local site for our algorithm dealing with the implicit set of tuples.

- **Cost Model # 1:**

  In this cost model we count the number of messages, $N_m$, that must be exchanged among all the participating sites in order to complete the execution of the algorithm. That is, one site is asked for its cluster centers, an answer is obtained, and then the request is sent to the next participating database. Each product term in the expression for counting tuples in equation 7.5 above requires an exchange of $n$ messages between the coordinating site and the participating sites. The product steps are repeated for
each tuple in \textit{Shareds} and therefore, the total number of messages to be exchanged for determining the count of tuples in \( D \) is \( n \times l^k \). In this case the complexity will be:

- \( n \) exchanged messages to perform local clustering.

- \( n \times l^k \) exchanged messages to compute the population for every \( c \in C' \).

Then the total number of exchanged messages will be \( n + n \times p \times l^k = n(1 + p \times l^k) \) and the total cost will be

\[
a \times n + a \times n \times p \times l^k = a \times n(1 + p \times l^k)
\]  \hspace{1cm} (7.10)

Where \( a \) is the time taken to exchange a message. The above results shows that the number of messages that need to be exchanged is dependent on the number of shared sites, the number of globalized cluster centers, and the number of tuples in the shared relation and is not dependent on the size of the database at each site. This is significant because it shows that as the sizes of the individual databases grow, the communication complexity of our algorithm would remain unaffected.

- **Cost Model \# 2:**

In this model we examine a weighted sum of the number of messages exchanged and the number of local operations performed. For each exchanged message, this algorithm performs a local operation at the responding site. Therefore, the total cost for the algorithm will be

\[
(a + b) \times n(1 + p \times l^k)
\]  \hspace{1cm} (7.11)
Where \( a \) and \( b \) are the weights representing the relative costs of exchanging a message and performing a local operation. Here again the above results shows that the number of message that need to be exchanged among the sites is not dependent on the size of the database at each site but the local computational cost at each site would grow with the database size at each individual site but our decomposable version has an advantage in this regard also over the transport, join, and then cluster the whole database.

- **Cost Model # 3:**

In this model we examine a weighted sum of the number of messages exchanged and the number of local operations performed, while discounting the effects of messages and operations that can be executed in parallel, simultaneously at different sites. Therefore, the total cost for the algorithm will be

\[
(a + b) \times \frac{n(1 + p \times l^k)}{n} = (a + b)(1 + p \times l^k)
\]

Where \( a \) and \( b \) are the weights representing the relative costs of exchanging a message and performing a local operation.

### 7.2 Advantages of Decomposable Algorithm

The above analysis of complexity shows that the number of messages that need to be exchanged among the sites is not dependent on the size of the database at each site. The communication complexity, in the case of vertically partitioned data, is dependent primarily on the number and manner in which the attributes are shared among the participating sites.
This is significant because it shows that as the sizes of the individual databases grow, the communication complexity of the algorithm would remain unaffected. Computational cost of local computations would grow with the database size at each individual site but our decomposable version has an advantage in this regard also over the transport, join, and then cluster alternative. If a k-means algorithm runs \( q \) iterations for finding \( k \) cluster centers and it has \( m \) data points then it must compute \( k \times j \times q \) distances.

If each local database \( D_i \) has \( m \) tuples, then in the worst case the join of \( n \) local databases would produce a relation containing order of \( m^n \) tuples. There is additional cost of order of \( (n \times m) \) comparisons for creating the Join. When the k-means algorithm is run with this explicitly created \( D \), we would need to compute \( q \times k \times m^n \) distances. In our decomposable version, each of the \( n \) sites would be computing only \( q \times k \times m \) distances. Thus, there is tremendous saving in the computational cost when the decomposable version is executed instead of moving the data, creating a Join and then running the clustering algorithm. Also, for the communication cost, the number of partial results that need to be transmitted is far fewer that the messages that may have to be transmitted if entire databases are collected at some central site.

Another important gain of decomposable version is that it preserves the security of the data by not requiring any data tuples to be placed on a communication network. It also preserves the integrity of individual databases because no site needs to update or write into any of the participating databases. All the queries are strictly reading queries.
7.3 Conclusion

In this chapter we have presented a decomposable version of *k*-means clustering algorithm to perform with vertically and horizontally partitioned datasets that are geographically distributed. The algorithm succeeds in obtaining results very close to those that would be achieved by moving all the databases to one site, joining them, and then executing the *k*-means clustering algorithm. Our distributed version of the algorithm succeeds in doing so by minimizing the total clustering error, a characteristic property of the *k*-means algorithm. We use the information about the clusters formed at local sites to determine the approximate locations of the possible global cluster centers. Information about the centers and an algorithm to count populations of points around cluster centers in an implicitly specified relation are used by the central coordinating site to minimize a close estimate of the total clustering error. We have demonstrated that the convergence of our version and the original *k*-means algorithm are to centers that are very closely placed; signified by a very small difference in the total clustering error. Our version achieves these very close results at a very great savings in the total communication cost and also preserves the security and integrity of the individual databases.

Most strategies for mining distributed databases depend on transferring either the data or final results from one site to a coordinating site. This may not work very well for clustering vertically partitioned datasets. Our algorithm has decomposed the *k*-means algorithm in such a way that no data needs to be transferred and better accuracy is achieved by exchanging results of a number of localized computations instead of just the final results from each site.
Chapter 8

Conclusion and Future Work

8.1 Conclusion

We have shown that graph and clustering algorithms can be adapted for distributed database environments. We have demonstrated that such algorithms involving counts of tuples, sum of products, ... etc can be computed in a distributed knowledge environment without moving all database to a single site. We have shown how the computations can be decomposed and demonstrated how computations can be performed by either sending one summary per message or by sending the whole set of summaries in one message. We have shown the following:

- It is possible to very efficiently find paths across a global graph formed by overlapping subgraphs stored on possibly sites of a wide-area network. These algorithms are decomposable at run time depending on the collection of sites with which a site can connect and communicate and also the set of shared vertices among their datasets. We have analyzed the complexity of our algorithms from the perspective of three differ-
ent cost models that take into account the communication cost across the sites of a communication network. It turns out that these algorithms can be executed with very low communication overhead. In almost all situations the cost of exporting datasets to a common site and then rebuilding the global graph at a local site for processing will take more communication energy and time than what is needed by the algorithms presented here. Also, these algorithms are independent of the manner in which the subgraphs may interface and overlap with each other (via the sets of shared vertices among them) and also the set of sites that may be available for participation in a particular instance of a computation.

- We have demonstrated that it is possible to design decomposable algorithms for computing shortest paths spanning tree and minimum spanning tree for databases stored across wide-area networks. These algorithms are decomposable at run time depending on the set of shared vertices among the graph components stored at different network sites. We examined the complexity of our algorithms from the perspective of cost models that take into account the communication cost across the sites of a wide-area network. It turns out that these graph algorithms can be computed without too much communication overhead. In some situations the cost of extracting a graph from a bigger database, exporting it to a common site and then rebuilding a local database for processing may take more time than what is required by the version of algorithms presented here. Also, these version of algorithms are independent of the manner in which the graphs interface with each other (the sets of shared vertices among them) and also the set of sites that may be available for participation in a particular instance
of a computation.

- It is possible to design a version of the \textit{k-means} clustering algorithm to perform clustering with horizontally partitioned datasets that are geographically distributed. The algorithm seeks to meet the characteristic properties achieved by the \textit{k-means} algorithm in the clusters that it forms. Our distributed version of the algorithm achieves the same characteristics by minimizing the clustering error. We use the information about the clusters formed at local sites to determine what the constitution of the global clusters should be. Population and radius of each local cluster is used to minimize the clustering error. It has been demonstrated with example datasets results showing the convergence and closeness properties of our algorithm are. We have demonstrated that very close clustering can be performed by transferring only some information from each site and there is no need to transfer entire datasets to a central location in order to perform clustering with the collective dataset.

- It is possible to design a decomposed clustering algorithm to perform clustering with vertically partitioned datasets that are geographically distributed. The algorithm seeks to meet very close results which can be achieved by moving, all data to one site, joining the relations, and applying any clustering algorithm. Our distributed version of the algorithm achieves the same characteristics by minimizing the clustering error. We use the information about the clusters formed at local sites to determine what the constitution of the global clusters should be. Population of each local cluster is used to minimize the clustering error. It has been demonstrated with example datasets and analytical results showing the convergence and closeness properties of our algorithm.
We have demonstrated that very close clustering can be performed by transferring only some information from each site and there is no need to transfer entire datasets to a central location in order to perform clustering with the collective dataset.

A very important contribution of these results is that some graph operations and clustering tasks can be performed in a number of databases residing at different network sites without having to move them to a single site the communication cost of such decomposable algorithms turns out to be a very low. These algorithms can prove extremely useful for many applications with databases at different sites of a network and still obtain too close results to the one would have been obtained if we had moved all these databases to a single site. Also these algorithms also preserve the privacy and security of the data at individual sites by requiring transmission of only minimal information to other sites.

8.2 Future Research

The following research problems are some of the extended work for our research

- consider a distributed system of sensors which collect data over a period of time. The sensors are laid in a grid like fashion and each of them has a neighbor on all four sides except the ones on the border. Assume a light ray say from a truck to have passed along a path which is constituted by these sensors. A couple of basic assumptions to this test are that the sensors are stationary and not on the move and that the time-stamps recorded by the sensors increase in a particular direction. At different time-stamps, depending on the sensing range of the sensors, the details of the light ray are recorded in each of them. These details include the Coordinates and the time-stamps. If needed
the light intensity could also be recorded. These details are present in each of the sensors through which the light has gone through. The problem is how we can mine the data collected by these sensors to produce meaningful results? Examples of the information we looking for are:

1. If there was any light ray which passed through any direction during a given time interval?
2. How many light rays passed through in a particular direction during a given time interval?

The main idea of the solution is to define Local Hypotheses and Global Hypotheses as the following:

- **Local Hypothesis:** A Local Hypothesis $LHs$ is a set of 3 or more points which lie in a line in any direction and which lie in ascending time-stamps fashion in that direction. e.g. if the direction is North (i.e at an angle of 0), then if the points $t_1$, $t_5$ and $t_9$ lie in the same line and in the same direction, they would make a Local Hypothesis. This can be applied to points lying in a straight line in any direction. The angle could be anything from 0 - 360°. For coding purposes, we have assumes the angles 0 - 90° to be the same as 180° - 270°. Angles 90° – 180° are the same as 270° – 360° and are represented as 0-90°.

- **Global Hypothesis:** Once the $LHs$ for all the sites are formed, these sites send their $LHs$ to all their neighboring sites, the list of which is maintained by each site. Once this is done, the current site will compute Global Hypotheses from
these data. A Global Hypothesis is formed when \( LHs \) from two or more sensors are taken and merged together to form a point which starts from the first sensor and ends in the last sensor and passes through the current sensor. The basis for merging the 2 \( LHs \) is whether the points lie in a line in the given direction and whether they lie in ascending time-stamps fashion.

Some of useful applications for this work would be in the following areas

- The army, where sensors can be distributed throughout an area which is required to be under supervision for enemy movements.

- The Department of Meteorology. It is possible to find out information about the area which has been devastated by the natural calamity.

- Determine ocean currents.

- Weather (predicting about Heat Frontiers, Ocean currents).

- Consider \( n \) databases located at \( n \) different network sites that together constitute the dataset \( \mathcal{D} \). The data at each site is modeled by a relation represented as a set of tuples. The set of attributes contained in a relation \( D_i \) is represented by \( X_i \). For any pair of relations \( D_i \) and \( D_j \) the corresponding sets \( X_i \) and \( X_j \) may have a set of shared attributes. The dataset \( \mathcal{D} \) with which the computation is to be performed is a subset of a set of tuples generated by a join operation performed on all the relations. \( D_1, D_2, \ldots, D_m \). However, the tuples of \( \mathcal{D} \) cannot be made explicit at any one site because entire \( D_i \)'s can not be moved to other sites. The problem is to compute maximum or
minimum value for \( f \) where

\[
f = F(x_1, x_2, \ldots, x_m)
\] (8.1)

where \( x_1, x_2, \ldots, x_m \) are attributes in the global databases \( D \)

- We have performed tests, running the algorithms on a various distributed databases. However most of the data that we have worked on has been numeric. Scope for further research exists in databases having non-numeric data like images and databases having attributes of different types like strings. Our algorithms consider the databases are complete i.e. there are no missing attribute values (no NULL values). Further research could involve methods to handle this situation.

- Another interesting area that can be further explored is the use of pattern discovery algorithms on data security and privacy of the user. Also researched will be the morel and ethical issues of discovering patterns in databases relating to particular demographics.
Bibliography


[12] Raj Bhatnagar, Barrington Young *Computations in Distributed Databases* proceedings of the ADCOM99 Conference held in Rookee, India, pp. 32-38. 1999.


[40] IEEE Conference Record of the Symposium on Feature Extraction and Selection in Pattern Recognition, Argonne, 1970.

[42] Fox, M.S. An organizational view of distributed systems IEEE trans on systems, man and cybernetics 11, 70-80.


[64] Ahmed Khedr, Raj Bhatnagar A Decomposable Algorithm for Minimum Spanning Tree submitted to AAMAS.


