An Efficient and Effective Computation of 2-Dimensional Depth Contours

by

Ivy Olga Kwok

B.Sc. (Hons), The University of British Columbia, 1996

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

Master of Science

in

THE FACULTY OF GRADUATE STUDIES

(Department of Computer Science)

we accept this thesis as conforming
to the required standard

The University of British Columbia

October 1999

In presenting this thesis in partial fulfilment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the head of my department or by his or her representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Department of Computer Science

The University of British Columbia
Vancouver, Canada

Date Oct 14, 1999

DE-6 (2/88)

ABSTRACT

Depth contour computation is a useful aid to outlier detection. For two-dimensional datasets, Rousseeuw and Ruts' ISODEPTH has been the state-of-the-art algorithm. Further contributions involve improving its efficiency and effectiveness, and resolving the issue of higher-dimensional datasets. This thesis details our research in these two areas. It presents a Fast Depth Contour Algorithm – FDC, which permits the selection of useful data points for processing and the removal of data point positioning limitations. Different selection methods result in the creation of FDC-basic and FDC-enhanced; and for buffer-space-compatibility, three further variants (FDC-M1, FDC-M2, and FDC-M3) of FDC-basic have been developed. FDC-basic capitalizes on the normal location of outiers and the relevancy of the vertices of convex hulls in depth contour computation. By computing with only the useful data points, it proves to run 150 times faster than ISODEPTH when computing the first 21 depth contours of a 6500-point dataset. FDC-enhanced takes only a quarter of the time required by FDC-basic to compute the first 151 depth contours of a 100,000-point dataset because it maintains a constantly adjusted sorted list of the useful data points to reduce redundant computation. FDC-M1 keeps dividing the dataset into subsets and discarding the useless data points until the union of the subsets fits the buffer size. FDC-M2 further uses a periodically adjusted inner convex hull to filter useless data points. FDC-M3 saves buffer space by starting the computation with a minimum dataset, which is enlarged with additional useful data points for each succeeding computation. To tackle datasets of higher dimensions, we have pioneered a promising algorithm in 3-dimensional computation, FDC-3D, which can be improved

with the use of sorted lists. Algorithm Fast Depth Contour not only provides an efficient and effective tool for immediate use, but also suggests a direction for future studies.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGEMENTS

I wish to acknowledge my gratitude toward Dr. Raymond Ng for initiating me into this research and guiding me till its completion. Without his insight into the importance of depth contour computation and his innovative ideas toward its improvement, this thesis would not have been possible.

I would also like to thank Dr. George Tsikins for taking much of his precious time to read through my thesis and making many valuable suggestions thereto.

IVY OLGA KWOK

The University of British Columbia

October 1999

Chapter One: INTRODUCTION

Since the introduction of the relational model in the early 1970's, many corporate enterprises have used database management systems (DBMS's) to store their business data. A database management system is designed to manage a large amount of information. It consists of a database and a set of programs to simplify and facilitate access to the data. Besides providing a convenient and efficient means of storing, retrieving and updating data, a database management system ensures the safety of the information stored and enforces integrity and concurrency controls. Relational database management systems store data in the form of related tables. These tables then form a relational database, which can be spread across several tables and viewed in many different ways. Relational database management systems are powerful because they require few assumptions about how data is related or how it will be extracted from the database.

For over 20 years, corporate databases have been growing exponentially in size. A huge amount of data is collected in these databases. Traditionally, the data have been analyzed to answer simple queries like "How many customers have bought product *X*?" "Which products sell the best?" and "How many transactions involve amounts greater than *m* dollars?" Yet, these data often contain more information than simple answers to specific queries. They can answer complex questions such as "What are the characteristics of customers who are most likely to buy product *X*?" "Which products are more frequently sold in combination?" "Which products are usually included in transactions with amounts greater than *m* dollars?" Valuable patterns that reveal the

customer characteristics are hidden in the database. However, the traditional database systems do not support searches for these interesting patterns. Therefore, we need more sophisticated analytical tools.

Data Mining

Data mining, or *knowledge discovery in large databases*, is the non-trivial extraction of implicit, previously unknown, and potentially useful information from large databases, thus, providing answers to non-specific but interesting queries. It has emerged from a number of disciplines, including statistics and machine learning, and has been made possible by advances in computer science and data storage. Since the 1990's, data mining has become a very popular research area in database. Initially, researches were interested in mining relational databases. Later on, studies are also conducted in mining temporal databases, such as those involving stock market data, and spatial databases, such as GIS's (Geographic Information Systems).

Not only has data mining excited the academic community, but it has also attracted a lot of corporate interest. Many companies have used data mining tools to help improve their business. Data mining tools can handle high volumes of data, and help both well-trained specialists and non-experts to answer business questions. One common use of data mining is database marketing. The identification of customer preferences helps to target the market and to personalize advertisements, which efficiently reduce advertising costs and attract customers.

Data mining tasks fall into four general categories: (a) dependency detection, (b) class identification, (c) class description, and (d) exception/outlier detection. The first three categories involve finding similar patterns that apply to the majority of objects in the database. On the other hand, the fourth category focuses on the identification of the *outliers*, the minority set, which is often ignored or discarded as noise. Most research works in data mining focus on finding patterns, rules, and trends in the majority set. Such works include concept generalizations [1, 2, 3], association rules [4, 5, 6, 7], sequential patterns [8, 9], classification [10], and data clustering [11, 12]. Some existing works have touched upon the outliers [11, 12, 13, 14]. Yet, they accept the outliers only as noise in the database and their main focus is still the majority set. However, identifying and analyzing the exceptions can be equally interesting as this aspect of data mining can reveal useful hidden information other than the frequent trends exposed in the data. For example, the identification of exceptional stock transactions and the detection of fraudulence uses of credit cards rely on this aspect of data mining [15].

Outlier Detection

In [16], Hawkins defined an outlier as "an observation that deviates so much from other observations as to arouse suspicions that it was generated by a different mechanism". His definition differentiates outliers from the noise in the database.

Traditionally, outlier detection follows a statistical approach [16, 17, 18]. In the field of statistics, more than one hundred discordancy/outlier tests have been developed. These tests are based on data distribution, knowledge of the distribution parameters, and

the number and the type of outliers. However, these tests are subject to two serious restrictions. First, because most of the tests involve only a single attribute, they cannot handle datasets with multiple attributes. Second, since they are all distribution-based and the distribution of an attribute is seldom given, extensive testing is necessary just to find out the appropriate distribution.

Arning, Agrawal, and Raghavan have developed a linear method of deviation detection [19]. Their algorithm searches a dataset for implicit redundancies, and extracts data objects called *sequential exceptions* that maximize the reduction in Kolmogorov complexity. However, their notion of outliers is very different form the aforementioned statistical definitions of outliers.

Besides the statistical approach, there is a distance-based outlier detection method developed by Knorr and Ng [20]. Distance-based outlier detection overcomes the restrictions inherent in the statistical approach. It can handle datasets that have multiple attributes following any data distribution. However, like any distance-based algorithm, distance-based outlier detection requires the existence of a well-defined metric distance function, which is not always available for every dataset.

To minimize the restrictions experienced in outlier detection, *depth-based* approaches have been developed. The *depth* of a data object relative to the dataset measures how deep the data object lies in the dataset. With a depth-based approach, each data object is assigned a depth and organized into layers based on the assigned depth. The deeper layers are more robust with respect to outliers than the shallower layers. Depth-based approaches avoid distribution-fitting problem, handle multi-dimensional

datasets, and do not require a metric distance function. Also, they are independent of the chosen coordinate system because the location of depth is affine-invariant, that is, the depth of a data object remains the same if the dataset is scaled, rotated, or translated.

Depth Contour

Tukey has suggested a simple procedure (known as "peeling" or "shelling") to locate suspected outliers in a dataset [21]. Peeling involves stripping away the convex hull of the dataset, then removing the convex hull of the remainder, and continuing until only a certain percentage of points remain. The number of convex hulls that have to be stripped from D before a point p is exposed determines the depth of p in the dataset D. Point depths contribute towards the construction of *depth contours*. When points of the same depth are connected, they produce one depth contour of the dataset. Figure 1-1 shows the depth contours of a dataset D consisting of data points $\{A, \ldots, Q\}$using the peeling definition. Points $\{A, \ldots, G\}$ make up the outermost convex hull and have depth 0; points $\{H, \ldots, M\}$ form the next convex hull and have depth 1; $\{N, O, P, Q\}$ constitute the innermost convex hull and have depth 2. As the example in Figure 1-1 demonstrates, peeling is straightforward; but it tends to move into the regions of high point density too quickly.

In 1975, Tukey defined a more robust notion of depth – *half-space depth* [18, 22, 23]. The half-space depth of a point p in a dataset is the minimal number of data points contained in a half-space the boundary plane of which passes through point p. The depth of a point p can also be viewed as the minimum number of data points that have to be

removed to expose p. For example, in a univariate dataset, $X = \{x_1, x_2, \ldots, x_n\}$, the depth of a point x_i is the minimum of the number of data points to its left and the number of data points to its right. Figure 1-2 shows the depth contours of the same dataset D using the half-space depth definition. By this new definition, points $\{A, \ldots, G\}$ still have depth 0; but points $\{H, \ldots, M\}$ now have depth 2, and points $\{N, \ldots, Q\}$ have depth 3. The stepping up in depth of these latter points is due to the fact that by the new definition their exposure requires the removal of at least two or at least three data points.

A comparison of Figure 1-1 and Figure 1-2 also reveals that while all vertices of the depth contours are actual data points under the peeling definition, they need not be so under the half-space depth definition. Indeed, a depth contour may not consist of any data points. For example, the vertex between H and I in the 2-depth contour in Figure 1-2 is not an actual data point; and there are no actual data points at all in the 1-depth, 4-depth, 5-depth and 6-depth contours.

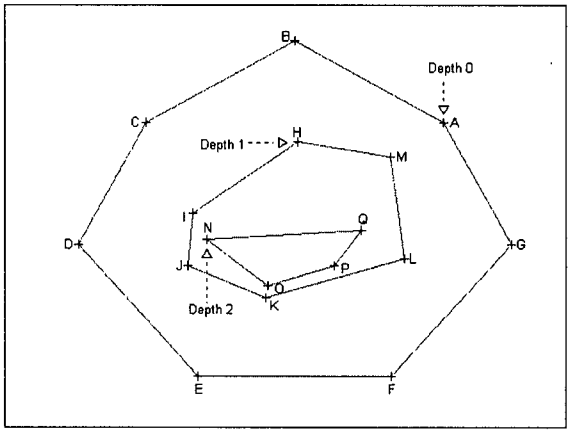


Figure 1-1 Peeling

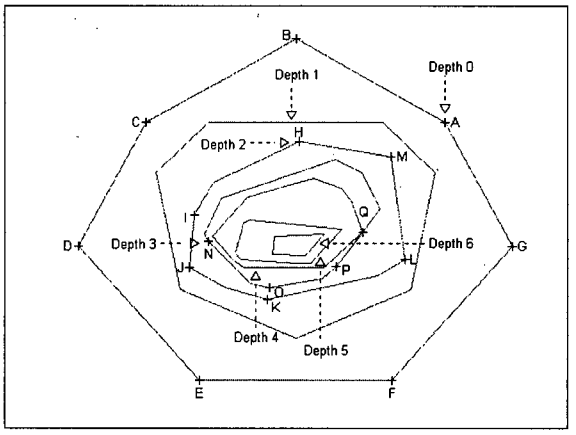


Figure 1-2 Half-space Depth

The k-depth contour, DC_k, of an n-point dataset is a sub-region in which every point has depth greater than or equal to k. It separates the data points with depth $< k$ and those with depth $\geq k$. The boundary points on DC_k have depth exactly equal to k, and the interior points have depth at least k. The outermost depth contour, 0-depth contour, is always the convex hull of the dataset. The k-depth contour is the intersection of all half-spaces which contain at least $n - k$ data points. All depth contours are convex and nested, that is, the k-depth contour is contained in the (k-1)-depth contour.

Depth contours help visualize the dataset. They are very useful in identifying the outliers and the "core" of the dataset. The outliers are usually located in the shallower depth contours of the dataset. On the contrary, the core data points are located in the deeper depth contours.

Motivation

Although depth contours are useful tools in detecting outliers, there are very few algorithms that compute depth contours. In [24], Rousseeuw and Ruts developed an algorithm, *ISODEPTH*, to compute the k-depth contour of a 2-dimensional dataset in three steps. ISODEPTH first checks if all n data points in the dataset are in general position, that is, no two points are the same and no three points are collinear. Then, it finds all half-spaces which contain $n - k$ data points[1]. Finally, it intersects the half-spaces and returns the depth contour.

[1] The definition of depth in ISODEPTH is off by one point from ours. We will discuss the algorithm and explain the difference in Chapter 2.

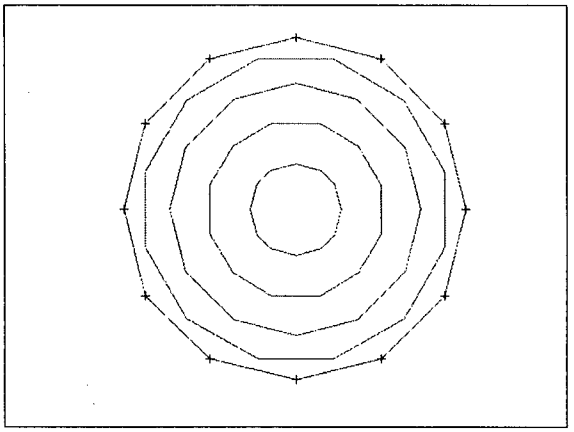


Figure 1-3 Data points in general position

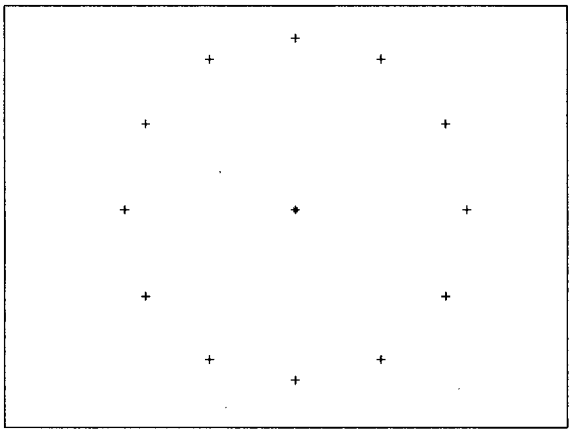


Figure 1-4 Data points not in general position

However, ISODEPTH has two drawbacks. The first one is that it imposes a harsh restriction on the position of the data points. All data points must be in general position, which is a condition that many datasets do not satisfy. Figure 1-3 shows 12 data points and their depth contours as generated by ISODEPTH. However, when we add one point to the center, as in Figure 1-4, ISODEPTH fails to compute the depth contours because the data points are no longer in general position. The removal of duplicate points and collinear points can be very time consuming.

The second drawback is that regardless of the value of k, ISODEPTH processes all data points in the dataset. However, when depth contours are used to find the outliers, the k-value is usually very small and most of the data points do not contribute to the depth contours. The last example in Table 1-1 shows that a very small percentage of the data points in a dataset can indeed make up the required depth contours. There are eight datasets of different sizes in the table; and, for each dataset, we have computed its first 151 depth contours (151 being a relatively high k-value for outlier detection.). We have included in Table 1-1 both the actual number of data points contributing to the depth contours and the percentage of those data points. The first of the eight datasets contains

two groups of original data taken from the National Hockey League (NHL) records, namely, the number of hockey games played and the plus-minus statistics of 855 NHL players in the year 1995-96. The remaining seven datasets are expanded datasets simulating the statistical distribution in the first dataset.

Dataset Size	Number of Data Points in the First 151 Depth Contours	% of Data Points in the First 151 Depth Contours
855	*521*	*60.93%*
500	450	90.00%
750	553	73.73%
1,000	629	62.90%
3,000	921	30.70%
5,000	1039	20.78%
10,000	1221	12.21%
100,000	1854	1.85%

Table 1-1 Number of data points in the first 151 depth contours of different datasets

Contribution

Our study comprising this thesis seeks to improve on the work of Rousseeuw and Ruts by removing some of the limitations of ISODEPTH, to approach from new perspectives the problem of developing an effective and efficient algorithm for 2-dimensional depth contour computation, and to probe into the possibility of multi-dimensional applications. The product of this thesis is our Algorithm *FDC* (*Fast Depth Contours*), which consists of six variants.

Unlike ISODEPTH, which generates only the k-depth contour in the computation, Algorithm FDC generates the first k^{th} depth contours of a dataset, thereby providing a more thorough view of the dataset to facilitate the identification of the outliers.

Algorithm FDC does not only relax the data point positioning criteria, but it also optimizes the time and space requirements. With the construction of the appropriate convex hulls, Algorithm FDC need not process the entire dataset but only a subset of the data points.

We implement a basic version (*FDC-basic*) and an enhanced version (*FDC-enhanced*) of Algorithm FDC for 2-dimensional datasets. FDC-basic is the first phase in the development of Algorithm FDC. It focuses on restricting the computation to a selected subset of data points. As mentioned earlier, when the k-value is small, most of the data points do not contribute to the requested depth contours. FDC-basic carefully works out subsets of data points to process. We claim that the depth contours generated from the chosen subsets are the same as those generated from the entire dataset.

Then, we work on improving the efficiency of our Algorithm FDC. Before computing any depth contour, FDC-basics must check the relevancy of all the half-spaces of the data points in the selected subset to see whether a half-space is required for the computation of the depth contour in question or not. Knowing that a half-space is contributive to the computation of only one depth contour, we make FDC-enhanced "remember" all those half-spaces that have already been used. This new step helps reduce redundant computation. In addition, we also introduce the use of sorted lists in FDC-enhanced to speed up the detection of half-spaces for future depth contour computation. These two steps combined raise the efficiency of our Algorithm FDC.

Furthermore, we realize that, like many other data mining applications, our Algorithm FDC may have to deal with disk-resident datasets that are too large for the

available buffer. Therefore, we work on the development of three variants of Algorithm FDC-basic, namely, FDC-M1, FDC-M2 and FDC-M3 to handle situations in which the buffer space is not sufficient to accommodate the entire dataset. These three variants of FDC-basic combine different merging methods with a divide-and-conquer approach to reduce the size of the dataset to fit the buffer space available for computing the requisite depth contours. All three variants are effective and show varying degrees of improvement in efficiency.

Finally, we work on extending Algorithm FDC to a 3-dimensional space. In designing Algorithm FDC-3D, we adhere to the same principal concept as is adopted for our Algorithm FDC-basic: let the algorithm choose the best subset of data points to work on, and terminate itself on completing the first k depth contours. We have completed the groundwork for FDC-3D, and will need further study to improve its performance. However, our work so far confirms our belief that the above concept can indeed apply to a 3-dimensional space. Algorithm FDC-3D is a milestone towards higher dimensional depth contour computation. Difficult to visualize as they may be, multi-dimensional depth contours will prove to be an important tool in detecting outliers.

Outline of the Thesis

This thesis is organized in the following manner. Chapter 1 gives the general background of our study, and touches on a few crucial historical developments. Chapter 2 explains in detail the concept of depth contours and provides a summary of ISODEPTH. Chapter 3 presents the basic version of Algorithm FDC and the logic behind its working. Chapter 4

discusses the enhanced version of Algorithm FDC. Chapter 5 suggests certain modifications to Algorithm FDC for working with a limited computer buffer space. Chapter 6 brings Algorithm FDC into a 3-dimensional data environment. Chapter 7 compares the performance of the various versions of Algorithm FDC discussed in the thesis. Chapter 8 concludes the thesis with a note on future study.

Chapter Two: DEPTH CONTOURS & ISODEPTH

In the previous chapter, we briefly discuss about the evolution of depth contour computation as a tool for data mining. We culminate the chapter with comments on an algorithm for computing depth contours proposed by Rousseeuw and Ruts, and point out that we intend to introduce our new depth contour algorithm, Algorithm FDC. For a better understanding of our problem, we will first define a few terms frequently used in the ensuing discussion, and provide a summary of Rousseeuw and Ruts' paper "Computing Depth Contours of Bivariate Point Clouds" [24] emphasizing how ISODEPTH computes a depth contour of a 2-dimensional dataset.

Background

Even though all the definitions here refer to 2-dimensional situations only, they apply equally well to cases of higher dimensions when suitably amended where necessary.

- **Data point:** A 2-tuple (p_x, p_y), where p_x is the x-coordinate and p_y is the y-coordinate.
- **Dataset:** A collection of data points.
- **Convex Set:** A dataset D that completely contains every straight line segment $<p, q>$, where p and q are in D.
- **Convex Hull of *D*:** The smallest convex set that contains the dataset D. (Intuitively, it can be viewed as the polygon formed by stretching a rubber band around the data points.)

- **Half-plane:** The part of the xy-plane that lies on either side of the boundary line.
- **Depth of *p*:** The minimal number of data points contained in a half-plane the boundary line of which passes through point *p*.
- ***k*-Depth Contour:** The closed boundary between all data points with depth < *k* and those with depth >= *k*, where *k* is a whole number.
- ***e*-Divider of *n* points:** A directed straight line *L*, or any finite line segment of *L*, with at most *e* points to its right and at most $(n - e)$ points to its left.
- **"Outside" of *L*:** The right-hand side of the directed line *L*.
- **"Inside" of *L*:** The left-hand side of the directed line *L*.
- **Inside Region of *L*:** The sub-region of the convex hull of a dataset that is to the left-hand side of the directed line *L*. It is the intersection of the convex hull and the half-plane to the inside of *L*.
- ***e*-Intersected Inside Region:** The intersection of all the inside regions of the *e*-dividers.

Depth contours form closed boundaries of points that have the same depth. Points on the *k*-depth contour, DC_k, have depth equal to *k*, those outside DC_k have depth less than *k*, and those inside have depth greater than *k*. Intuitively, every edge of the *k*-depth contour in a 2-dimensional data space has at most *k* data points to its outside.

The 0-depth contour, DC_0, of the dataset is the convex hull of the dataset. The convex hull is the smallest convex set that contains the dataset. Since no data points are outside the convex hull, for every point *p* on the boundary of the convex hull, there exists

a half-plane, whose boundary line passes through p, which contains no other data points. By definition, these boundary points have depth 0. All the interior points of DC_0 have depth greater than 0 because at least one of the boundary data points needs to be removed to expose an interior point. The subsequent depth contours are constructed by intersecting the appropriate inside regions. Thus, the k-depth contour is the intersection of all the inside regions of the k-dividers, which is the k-intersected inside region.

The depth contours form a collection of nested convex sets. All inside regions are convex; the intersection of convex regions is convex. Thus, all depth contours are convex. Because points outside DC_k have depth $< k$ and those contained in DC_{k+1} have depth strictly greater than k, DC_{k+1} is completely contained in DC_k. Different depth contours never intersects; but that is not to say that DC_k and DC_{k+1} cannot touch each other. This scenario is possible when there are collinear points in the dataset. Figure 2-1 shows an example where the 0-depth contour touches the 1-depth contour at Point A. Point A has depth 0, but it is on both the 0-depth and the 1-depth contours.

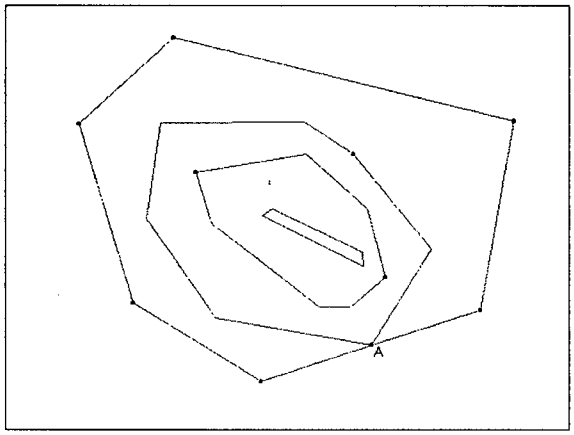


Figure 2-1 0-Depth Contour touches 1-Depth Contour

ISODEPTH

Among the very few studies conducted on depth contours, Rousseeuw and Ruts' work – ISODEPTH – is the state-of-the-art algorithm, and the following is a brief summary of it. In ISODEPTH, the definition of depth is off by one point from ours; that is, instead of calling the outermost depth contour 0-depth contour as we do, ISODEPTH calls it 1-depth contour. To facilitate comparison with the original text, we have adopted this slightly different definition of Rousseeuw and Ruts' when preparing the summary.

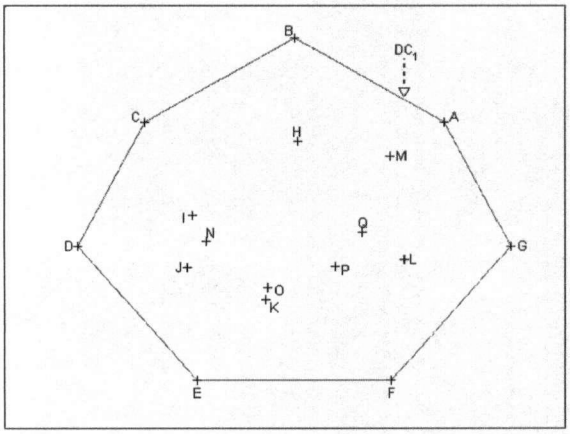


Figure 2-2 1-Depth Contour generated by ISODEPTH

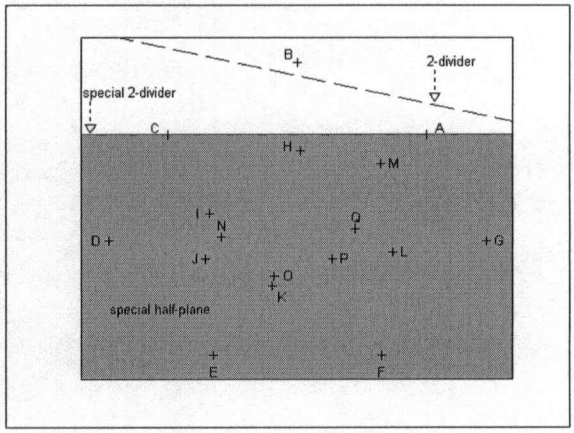


Figure 2-3 2-Divider, Special 2-Divider and its Special Half-plane

ISODEPTH defines the directed line L as a k-divider of n data points where L has at most $k - 1$, instead of k, data points to its outside and at most $n - k$ to its inside. L is a *special* k-divider if it contains at least two data points. The inside half-plane bounded by a special k-divider is the *special half-plane*, which contains at least $n + 1 - k$ data points. The k-depth contour is the intersection of all special half-planes.

ISODEPTH computes the k-depth contour of an n-point dataset in three steps. In the first step, ISODEPTH checks whether the dataset D is in general position, that is, no

two points are the same and no three points are collinear. The data points are sorted first by their x-coordinates then by their y-coordinates. If two data points have the same x- and y-coordinates, ISODEPTH halts because two data points coincide. If three points have the same x-coordinates, it halts too because three data points lie on a vertical line. Then, for each pair of data points, p_i and p_j, ISODEPTH computes the angle between the line directed from p_i to p_j and the horizontal axis. n data points result in n^2 pairs and, hence, n^2 angles. These angles are sorted and stored in array *ANGLE*. If two angles are the same and the corresponding lines share a common point, ISODEPTH halts because three data points are collinear.

In the second step, ISODEPTH detects all special k-dividers using the *circular sequence* technique. The circular sequence is a periodic sequence of the $n(n-1)$ permutations of n numbers. Given a dataset $D = \{p_1, p_2, \ldots, p_n\}$, where all data points are in general position and a directed line L that is not perpendicular to any line through two data points in D, the orthogonal projection of D onto L gives a permutation of the set of indices $N = \{1, 2, \ldots, n\}$. As L rotates counterclockwise, it defines the circular sequence of permutations of N. The complete circular sequence can be obtained by rotating L over only half a circle because the remaining permutations are exactly the reverse of the previous permutations.

The difference between two successive permutations in the circular sequence is an exchange of positions between two adjacent numbers, for example, $\{1, \ldots, h, i, j, k, \ldots n\}$ and $\{1, \ldots, h, j, i, k, \ldots, n\}$. The exchange takes place when the line passing through p_i and p_j is perpendicular to L, in which case p_i and p_j project to the same spot on L. Consider the example shown in Figure 2-4. Let α be the angle between L and the

horizontal axis when L is perpendicular to the line passing through p_i and p_j. Suppose, before L reaches the angle α, the projection of p_i on L precedes that of p_j, that is, i precedes j in the permutation. When L and the horizontal axis form the angle α, the two projections coincide and the exchange takes place. After L passes the angle α, the projections of p_i and p_j change order and i follows j in the new permutation.

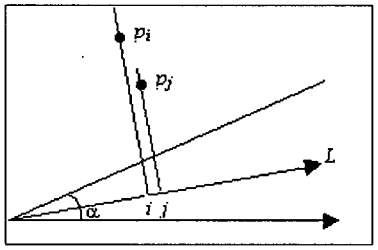


Figure 2-4a i precedes j

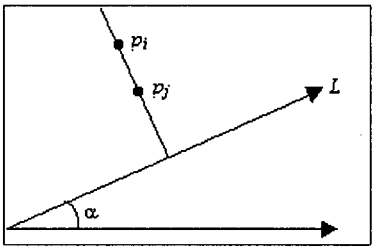


Figure 2-4b i coincides with j

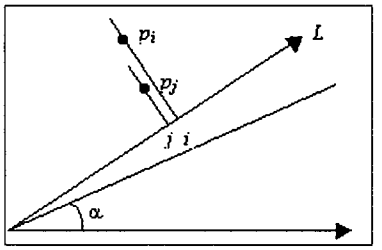


Figure 2-4c i follows j

ISODEPTH constructs the circular sequence and, in doing so, detects the special k-dividers. If i and j exchange positions and their positions are k and $k + 1$ or $n - k$ and $n - k + 1$, the line passing through p_i and p_j has $k - 1$ data points to either its left side or its right side. Thus, either the line directed from p_i to p_j or the one directed from p_j to p_i is a special k-divider.

Since the data points are already sorted by their x-coordinates in the first step, ISODEPTH readily gets the first permutation $\{1, 2, \ldots, n\}$ from the projection on the horizontal axis. ISODEPTH then rotates the projection line L from 0 to *ANGLE*(1). During the rotation, if the line passing through data points p_i and p_j is perpendicular to L, the positions of i and j will exchange places in the next permutation. In other words, an exchange occurs if there exists a line which is orthogonal to the line passing through any two data points and whose angle is in the range [0, *ANGLE*(1)]. Since the angles of the orthogonal lines can be computed from array *ANGLE*, an exchange, if there is one, can be

easily detected by going through *ANGLE*. A special k-divider is found if the exchange takes place at positions k and $k + 1$ or $n - k$ and $n - k + 1$. ISODEPTH continues checking for exchanges that occur between *ANGLE*(1) and *ANGLE*(2), between *ANGLE*(2) and *ANGLE*(3), and so on, till *ANGLE*[$^{n(n-1)}/_2$] and π.

Once all the special k-dividers are found, ISODEPTH computes the intersection of all the special half-planes in the third step. It first sorts the special k-dividers by their angles. It then walks along the dividers to trace the intersection. When it completes the entire cycle, ISDOEPTH gets all the vertices of the k-depth contour. If another depth contour is needed, ISODEPTH repeats the second and the third steps.

Complexity Analysis

In the first step, ISODEPTH sorts n data points and n^2 angles. With a standard sorting sub-routine, this step yields time complexity $O(n^2 \log n)$. In the second step, since the angles in *ANGLE* are sorted, ISODEPTH can search for all the special k-dividers by going through *ANGLE* at most twice. Thus, the second step yields time complexity $O(n^2)$. Because the number of special k-dividers is proved to be bounded by $O(n\sqrt{k})$ and k is at most $O(n)$, the number of special half-planes is at most $O(n^{1.5})$. Since the intersection of N half-planes requires $O(N \log N)$ time, to compute the intersection of the special half-planes, the third step yield time complexity $O(n^{1.5} \log n)$. Therefore, the overall time complexity for computing one depth contour with ISODEPTH is $O(n^2 \log n)$. If contours from depth 0 to depth k are required, the second and the third steps will be repeated k+1 times, and ISODEPTH yields complexity $O(n^2 \log n + kn^2 + kn^{1.5} \log n)$.

ISODEPTH uses a number of arrays in the computation. Arrays X and Y, of length n, store the x- and the y-coordinates of the data points. *ANGLE*, *IND1* and *IND2*, of length n^2, store the angles of all lines passing through two data points and the indices of the corresponding data points. *NCIRQ* and *NRANK*, of length n, store the permutation and the positions of the data points in *NCIRQ*. *ALPHA*, *D*, *KAND1* and *KAND2*, of length at most $O(n^{1.5})$, store the equations of the special k-dividers and the indices of the corresponding data points. Therefore, ISODEPTH requires $O(n^2)$ space for the computation.

Chapter Three: FAST DEPTH CONTOUR ALGORITHM – FDC

We have analyzed the performance of ISODEPTH, pointed out its shortcomings in relation to, at least, quadratic time and space complexities, and concluded that ISODEPTH is not feasible for large datasets. To remedy these shortcomings, we now present our Fast Depth Contour Algorithm – FDC. The FDC Algorithm presented in this chapter is the basic version of our algorithm and is called FDC-basic. We will simply call FDC-basic FDC in this chapter.

Our major contention is that FDC does not have to process all data points to compute depth contours. Since depth contours are often used to differentiate between outliers and the core of the dataset, in many situations, only the first few depth contours are needed. The majority of the dataset does not contribute to the shallow depth contours. In one of our NHL test cases, we computed the first 151 depth contours of a generated dataset that has 100,000 data points. Only 1854 data points, that is, fewer than 2% of the total, belong to these depth contours. If we can minimize the work processing the "useless" data points, the performance can improve drastically.

Algorithm FDC

As pointed out at the beginning of the chapter, FDC aims at processing only a selected subset of data points. The dataset is divided into two disjoint subsets: the *outer* point set consisting of data points that are outside the current depth contour and the *inner* point set comprising the remaining data points. Initially, the outer point set is empty and the inner

point set is the dataset. When FDC starts generating a depth contour, it considers only the outer point set and the inner convex hull (or the data points forming the convex hull of the inner point set) while ignoring the data points that are inside the convex hull. Certain data points are then removed from the inner point set and added to the outer point set as FDC moves inward to build up the depth contours at deeper layers.

Let us review a few more terms before we present the pseudo code of Algorithm FDC. We use $<p, q>$ to denote is a straight line segment directed from data point p to data point q. The right side of $<p, q>$ is its outside; the left side of $<p, q>$ is its inside. $<p, q>$ is an e-divider of an n-point dataset D if the directed line supporting $<p, q>$ divides D such that the subset to the right of the line has at most e data points and the subset to the left has at most $(n - e)$ data points. The inside region of $<p, q>$, denoted as $IR(<p, q>)$, is the intersection of the convex hull of D and the half-plane to the inside of $<p, q>$. If $<p, q>$ is an e-divider, $IR(<p, q>)$ is an e-inside region. Given a set of e-inside regions, their common intersection is the e-intersected inside region.

Algorithm FDC(D, k)

Input: D = the dataset and k = an integer.

Output: Contours of depth from 0 to k.

1 CH = vertices of the convex hull of D;
2 Output CH as the vertices of the 0-depth contour;
3 If $k == 0$, done;
4 Initialize $T = \varnothing$;
5 For ($d = 1$; $d \leq k$;) { /* new peel */
  5.1 If CH is empty, break; /* Done and Stop */
  5.2 $T = T \cup CH$; $D = D - CH$;
  5.3 CH = vertices of the convex hull of (the updated) D;
  5.4 If $|CH| == 3$, continue; /* degenerate case of having a triangle as the convex hull */
  5.5 For ($e = d$; $e < {}^{|CH|}/_{2}$;) { /* Otherwise: the general case */
    5.5.1 For all points $p \in T$ {

5.5.1.1 Find all points q in T (if any) so that $<p, q>$ is an e-divider of the points in T;

5.5.1.2 If every point in CH is contained in $IR(<p, q>)$,

$IR[p] = IR(<p, q>)$; $CT[p] = \varnothing$;

5.5.1.3 Else call Procedure Expand(p, q, CH, T, e) to compute $IR[p]$ and $CT[p]$;

} /*end for */

5.5.2 Output the boundary of the region $(\cap_{p \in T} IR[p])$as the d-depth contour;

5.5.3 If $d == k$, break; /*Done and Stop */

5.5.4 /* Otherwise: */ $d = d + 1$; $e = e + 1$;

5.5.5 If $\cup_{p \in T} CT[p] \neq \varnothing$ {

5.5.5.1 $T = T \cup (\cup_{p \in T} CT[p])$; $D = D - (\cup_{p \in T} CT[p])$;

5.5.5.2 CH = vertices of the convex hull of (the update) D;

} /* end if */

} /* end for */

} /* end for */

If some points in the inner convex hull CH are not contained in the inside region of an e-divider $<p, q>$, we have to expand $IR(<p, q>)$ with the points outside the inside region. The expanded series, $<p_1, \ldots, p_n>$ (where $p_1 = p$, $p_n = q$, and $p_i \neq p_j$ for all $i \neq j$), is a set of line segments $<p_i, p_{i+1}>$ for $1 \leq i \leq (n - 1)$. The expanded inside region, $IR(\ldots, <p_i, p_{i+1}>, \ldots)$, is the intersection of the inside regions of the line segments in the expanded series that are e-dividers. The intersection of some e-inside region and at least one expanded e-inside region is an expanded e-intersected inside region.

Procedure Expand(p, q, CH, T, e)

Input: $<p, q>$ = an e-divider of points in T, and CH contains point(s) outside $IR(<p, q>)$.

Output: $IR[p]$ and $CT[p]$.

1 Enumerate (in counter-clockwise fashion) all the points in CH that are outside $<p, q>$ as $r_1, \ldots, r_n$;

2 Among them, find r_i that maximizes the angle between the segment $<r_i, p>$ and $<p, q>$;

3 Among them, find r_j that maximizes the angle between the segment $<r_j, q>$ and $<q, p>$;

/* Without loss of generality, assume that $i \leq j$. For convenience, rename the expanded series $<p, r_i, \ldots, r_j, q>$ as $<p_0, p_1, \ldots, p_{w-1}, p_w>$, where $w = j - i + 2$ */

4 Initialize $IR[p]$ = convex hull of T and $CT[p] = \varnothing$. /* convex hull of $T = DC_0$ */

5 For ($u = 1$; $u \leq w$; u++) { /* iterate over each segment */

5.1 If the line $<p_{u-1}, p_u>$ is not an e-divider of the points in $T \cup \{q_i, \ldots, q_j\}$, continue;

5.2 Else { /* expand the inside region */

5.2.1 $IR[p] = IR[p] \cap IR(<p_{u-1}, p_u>)$;

5.2.2 Add p_{u-1} and p_u to $CT[p]$;

} /* end else */

} /* end for */

6 Return $IR[p]$ and $CT[p]$;

Demonstration

To make our algorithm easier to understand, we are going to construct the depth contours of a sample dataset. Figure 3-1 shows a dataset of 17 data points. We are going to compute the first 3 depth contours of the dataset step by step. We call the outer point set T, the inner point set D, and the vertices of the inner convex hull CH.

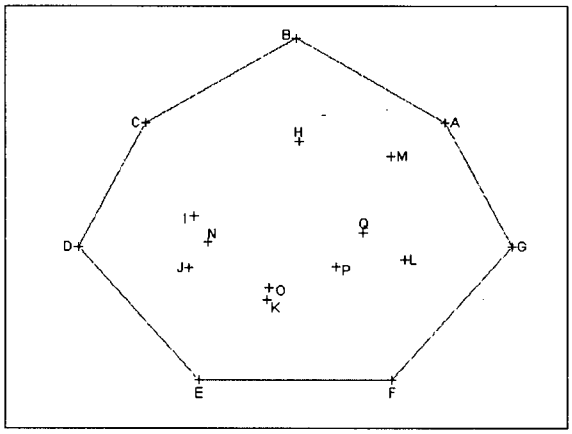


Figure 3-1 0-Depth Contour of the 17-point dataset

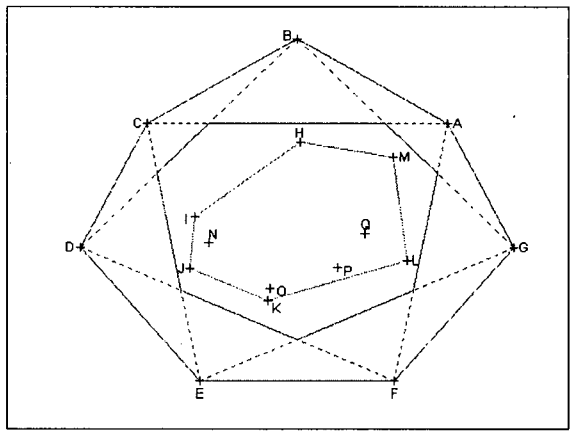


Figure 3-2 1-Dividers , 1-Intersected Inside Region and 1st Inner Convex Hull

In Step 1 of Algorithm FDC, we find the convex hull of D to be the polygon with vertices $CH = \{A, \ldots G\}$. The convex hull is also the 0-depth contour, DC_0, of D. Then,

we initialize the outer point set T in Step 4 and proceed to Step 5, the heart of FDC. Data points A to G are removed from D and added to T in Step 5.2, and the updated D becomes $\{H, \ldots, Q\}$ with the first inner convex hull $CH = \{H, \ldots, M\}$.

We are now ready to compute the 1-depth contour. In the first iteration of the for-loop in Step 5.5, we go through all the data points in T successively to find their corresponding 1-divider. We start with data point A, which has a 1-divider $<A, C>$. The inside region of $<A, C>$, $IR(<A, C>)$, is the polygon with vertices A, C, ..., G. Because $IR(<A, C>)$ contains all the points in CH, $IR[A]$ equals $IR(<A, C>)$ following Step 5.5.1.2. Indeed, all the inside regions of the 1-dividers of T contain CH. We compute the 1-intersected inside region in Step 5.5.2 and output the boundary of the region as the 1-depth contour, DC_1.

The 1-dividers and the 1-intersected inside region, whose boundary is the 1-depth contour DC_1, are shown in Figure 3-2. From the figure, we see that every edge of DC_1 has exactly one of the data points $\{A, \ldots, G\}$ to its outside. Any point p that is inside DC_0 but outside DC_1 or on the boundary of DC_1 has depth 1. In order to expose p, at least one of the points $\{A, \ldots, G\}$ has to be removed. The points inside DC_1 have depth greater than 1. Any line passing through a point inside DC_1 separates at least two of the points $\{A, \ldots, G\}$ from the rest of the set.

In our example, all the points in CH are contained in the 1-depth contour; therefore we need not evoke Steps 5.5.5.1 and 5.5.5.2 to update T, D or CH before computing the 2-depth contour. Instead, we move on to the second iteration of Step 5.5 to find the 2-dividers of the points in T. In this instance, we find that unlike the 1-

intersected inside region, the 2-intersected inside region does not contain all the points in *CH* (Figure 3-3). For example, <*A*, *D*> is a 2-divider of *T*. However, *IR*(<*A*, *D*>) does not contain point *H*. We, therefore, call Procedure Expand in Step 5.5.1.3 of Algorithm FDC to expand the 2-inside regions that do not contain all the points in *CH*.

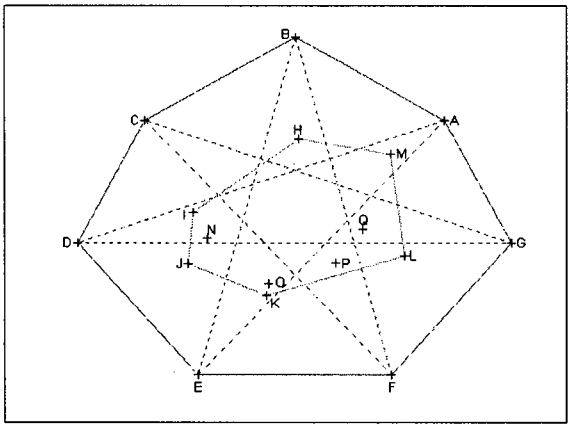


Figure 3-3 2-Dividers, 2-Intersected Inside Region and 2nd Inner Convex Hull

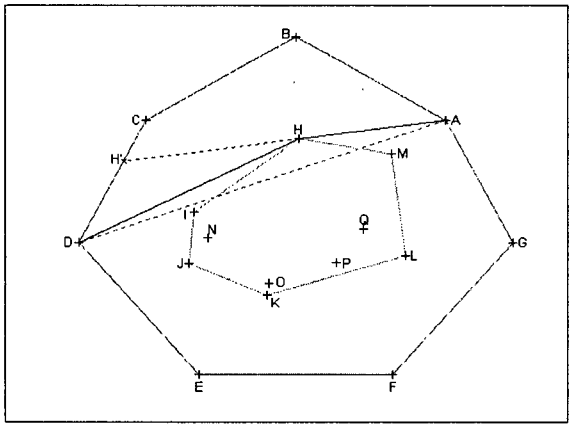


Figure 3-4 Expanded *IR*(<*A*, *H*>, <*H*, *D*>)

In Procedure Expand, we compute the expanded inside region, *IR*[*A*], and obtain the list of the outside points, *CT*[*A*], for data point *A*. As *H* is the only point in *CH* that is outside *IR*(<*A*, *D*>), we define the expanded series as <*A*, *H*, *D*> by the end of Step 3 under Procedure Expand, initialize *IR*[*A*] with DC_0, and set *CT*[*A*] to null in Step 4. In the first iteration of Step 5, we discover that <*A*, *H*> is a 2-divider with data points *B* and *C* to its outside. Therefore, we subscribe *IR*[*A*] to the intersection of DC_0 and *IR*(<*A*, *H*>), which is the polygon with vertices *A*, *H'*, *D*, ..., *G* as shown in Figure 3-4; and add points *A* and *H* to *CT*[*A*]. Then, in the second iteration of Step 5, we discover that <*H*, *D*> is a 2-divider, too. The expanded inside region *IR*[*A*] is *IR*(<*A*, *H*>, <*H*, *D*>), which is the polygon with vertices *A*, *H*, *D*, ..., *G*, and *CT*[*A*] has points *A*, *H*, and *D*.

In a similar fashion, we go on to inspect data point B, which has a 2-divider $<B, E>$. We find that data points I and J are to the outside of $IR(<B, E>)$. We, therefore, expand $<B, E>$ in Step 5.5.1.3 of FDC. Then, in Step 2 of Procedure Expand, we find that data point I maximizes the angle $\angle r_i BE$, and in Step 3, data point J maximizes the angle $\angle r_j EB$. As neither is I contained in the region bounded by B, J and E, nor is J contained in the region bounded by B, I and E, the expanded series has to be $<B, I, J, E>$ (Figure 3-5). Since $<B, I>$, $<I, J>$ and $<J, E>$ are all 2-dividers, the expanded insider region $IR[B]$ has vertices $B, I, J, E, \ldots, A$, and $CT[B]$ has points B, I, J, and E.

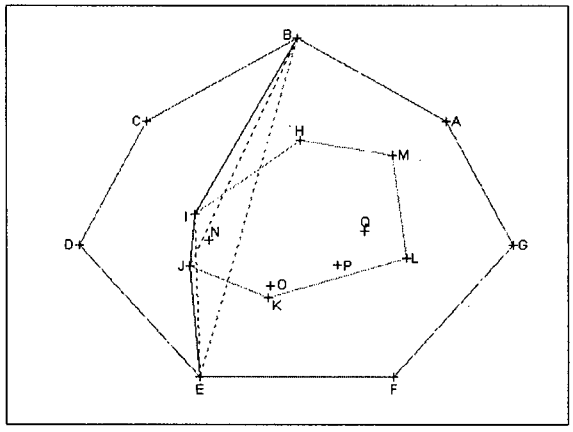


Figure 3-5 Expanded $IR(<B, I>, <I, J>, <J, E>)$

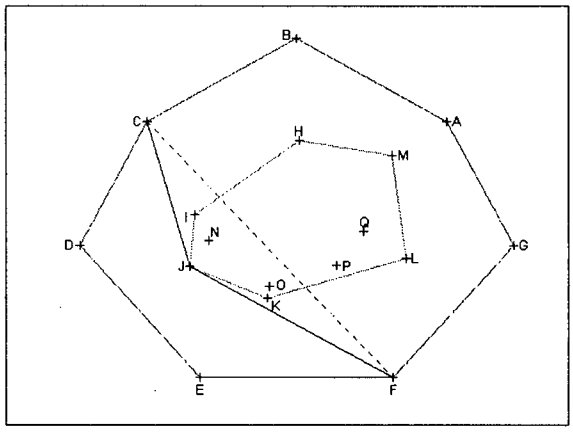


Figure 3-6 Expanded $IR(<C, J>, <J, F>)$

Having considered points A and B, we will move on to consider data point C with a view to expanding the 2-divider $<C, F>$ and determining the expanded inside region $IR[C]$. Figure 3-6 shows that data points I, J and K are outside $IR(<C, F>)$. Point J also maximizes both the angles $\angle r_i CF$ and $\angle r_j FC$. Points I and K are, therefore, contained in the region bounded by C, J and F, and cannot be included in the expanded series $<C, J, F>$. $IR[C]$ is the polygon with vertices $C, I, F, \ldots, B$ and $CT[C]$ has points C, J, and F.

We find the remaining inside regions in a similar way. Figure 3-7 shows all the (expanded) 2-dividers and the intersection of their inside regions. This expanded 2-intersected inside region is the 2-depth contour. Since the *CT* lists are not empty this time, we update *T*, *D*, and *CH* in Steps 5.5.5.1 and 5.5.5.2 of FDC. *T* now has points {*A*, *B*, ..., *G*, *H*, ..., *M*} while both *D* and *CH* contains points {*N*, *O*, *P*, *Q*}.

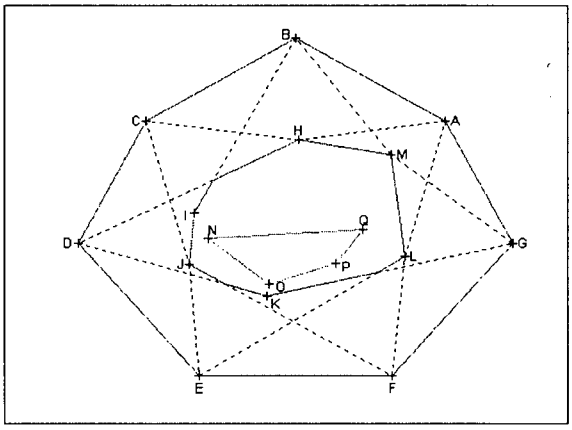


Figure 3-7 2-Dividers, Expanded 2-Intersected Inside Region and 3rd Inner Convex Hull

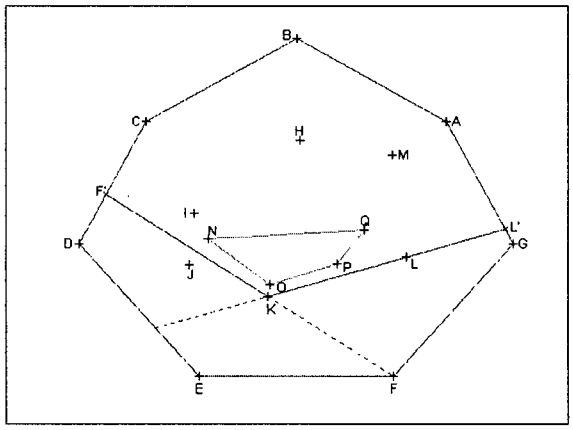
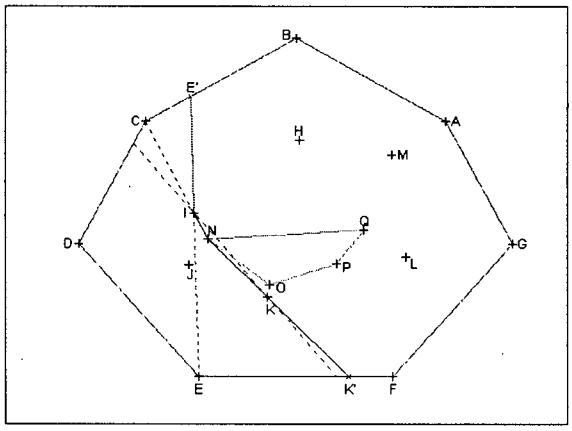


Figure 3-8 $IR[K] = IR(<K, F>) \cup IR(<K, L>)$

To find the 3-dividers, we go back to Step 5.5 of FDC for the third time. The procedures are the same as before. However, we should point out that while the data points on the outermost depth contour always have unique *e*-dividers, the data points inside the outermost depth contour may have multiple *e*-dividers. For example, data points *I*, *J* and *K* have two 3-dividers. Figure 3-8 shows the two 3-dividers for *K*, Figure 3-9 shows for *I*, and Figure 3-10 shows those for *J*.

Finding the 3-inside region for *K* is simple. Both $IR(<K, F>)$ and $IR(<K, L>)$ contain all the points in *CH*. Thus, $IR[K]$ is the intersection of the above two inside regions, which is the polygon with vertices *F'*, *K*, *L'*, *A*, *B*, and *C*.

Finding $IR[I]$ is more interesting. As shown in Figure 3-9, both $<I, E>$ and $<I, K>$ are 3-dividers for I. But since $IR(<I, E>)$ contains all the points in CH while $IR(<I, K>)$ does not contain data point N, we need to first expand $IR(<I, K>)$ by calling Procedure Expand in Step 5.5.1.3. The expanded series of $<I, K>$ is $<I, N, K>$, where both $<I, N>$ and $<N, K>$ are 3-dividers. Therefore, $IR[I]$ is the intersection of $IR(<I, E>)$ and $IR(<I, N>, <N, K>)$, which is the polygon with vertices E', I, N, K', F, G, A, and B.



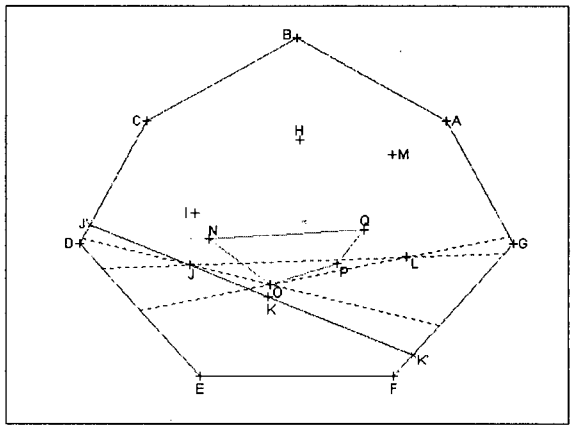


Figure 3-9 $IR[I] = IR(<I,E>) \cup IR(<I,N>,<N,K>)$ **Figure 3-10** $IR[J] = IR(<J, K>)$

Point J also has two 3-dividers, namely $<J, K>$ and $<J, L>$. We need to expand $IR(<J, L>)$ because it does not contain data points O and P. Since P is contained in the region bounded by J, O, and L, the expanded series is $<J, O, L>$. However, data points D, E, F, and K are outside $<J, O>$, and data points E, F, G, and K are outside $<O, L>$. Therefore, $<J, O>$ and $<O, L>$ are 3-dividers. And since we are only interested in the 3-dividers at this moment, $<J, O>$ and $<O, L>$ are ignored and, thus, the expanded inside region of $<J, L>$ is the convex hull of the dataset (see Step 5.1 in Procedure Expand). As a result, $IR[J]$ is $IR(<J, K>)$, which is the polygon with vertices J', K', G, A, B, and C.

Figure 3-11 shows all the (expanded) 3-dividers. The 3-depth contour is the expanded 3-intersected inside region. We will continue with the Algorithm FDC to compute the rest of the depth contours. FDC will stop when it computes the 7-depth contour of the dataset because the 7-intersected inside region is empty. Figure 3-12 shows all the depth contours of the dataset, from the 0-depth contour to the 6-depth contour.

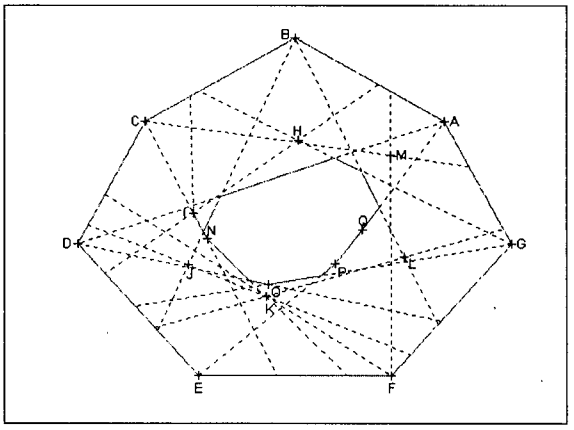


Figure 3-11 3-Dividers and Expanded 3-Intersected Inside Region

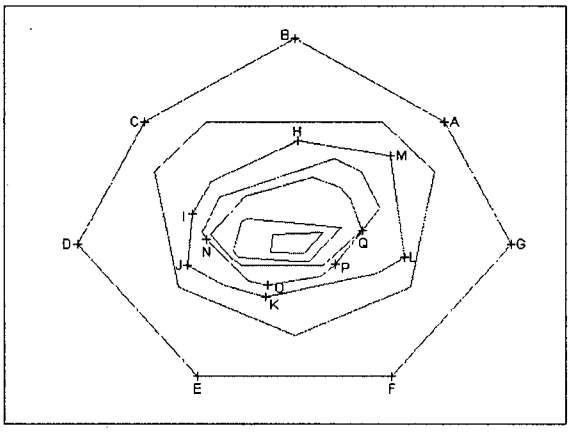


Figure 3-12 0-Depth to 6-Depth Contours of the dataset

The above example explains the normal operations of our Algorithm FDC. In the exceptional case where e reaches half the size of T in the for-loop in Step 5.5, continuous iteration without modifying T will simply wrap around and reverse the previously found dividers. Therefore, in this case, the execution returns to the beginning of Step 5 to update T, D and CH before it moves on to the next iteration.

For this exceptional case, consider the example shown in Figure 3-13. The 0-depth, the 1-depth, and the 2-depth contours of 12 data points $\{A, \ldots, L\}$ are shown. Points $\{A, \ldots, G\}$ are in the outer point set T and points $\{H, \ldots, L\}$ are in the inner convex hull CH. Let us say we have just finished computing the 2-depth contour in Step 5.5.2 and incremented e to 3 in Step 5.5.4. Since the 2-depth contour contains all the

points in *CH*, *T* remains unchanged, and has 7 points {*A*, ..., *G*}. As *e* reaches half the size of *T*, we exit the for-loop in Step 5.5 and return to Step 5.1 to update *T*, *D* and *CH*. But, what will happen if we continue with *T* = {*A*, ..., *G*} in the third iteration? Can we still find the 3-depth contour without modifying *T*, *D* and *CH*?

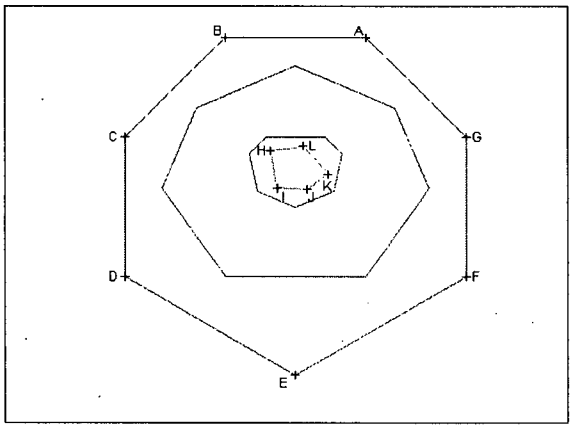


Figure 3-13 0-Depth to 2-Depth Contours and Inner Convex Hull

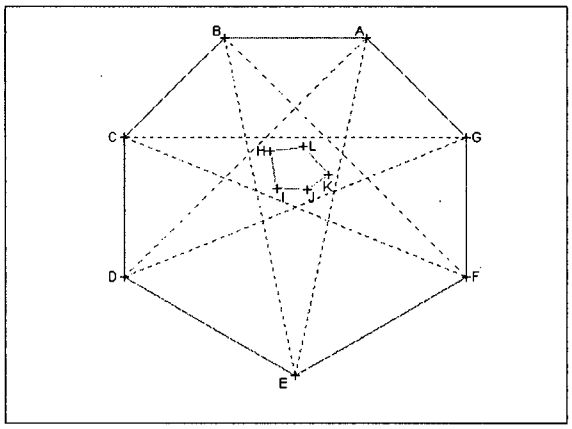


Figure 3-14 3-Dividers and Inner Convex Hull

If we continue with *T* = {*A*, ..., *G*}, we will find the 3-dividers <*A*, *E*>, <*B*, *F*>, ..., <*G*, *D*>, but these 3-dividers are the previously found 2-dividers with the direction reversed. For example, <*A*, *E*> is a 3-divider and <*E*, *A*> is a 2-divider. Since the 2-intersected inside region of *T* contains all the points in *CH*, it follows that all the 2-inside regions of *T* contain all the points in *CH*. If the inside region of a line *L* contains all points in *CH*, the outside region of *L* does not contain any point in *CH*. In other words, if *IR*(<*E*, *A*>) contains all points in *CH*, the outside region of <*E*, *A*> does not contain any point in *CH*. However, the outside region of <*E*, *A*> is the same as the inside region of <*A*, *E*>; therefore, *IR*(<*A*, *E*>) does not contain any point in *CH*, and it has to be expanded.

In fact, if we continue with $T = \{A, \ldots, G\}$, all inside regions of the 3-dividers will be expanded. This will be an inefficient process, as we already know that *CH* is not contained in these inside regions. We know that T is not sufficient for the computation of the 3-depth contour; hence, we should not continue with T. We should merge T and *CH* before the next iteration. Figure 3-15 shows all the depth contours of the dataset in this example.

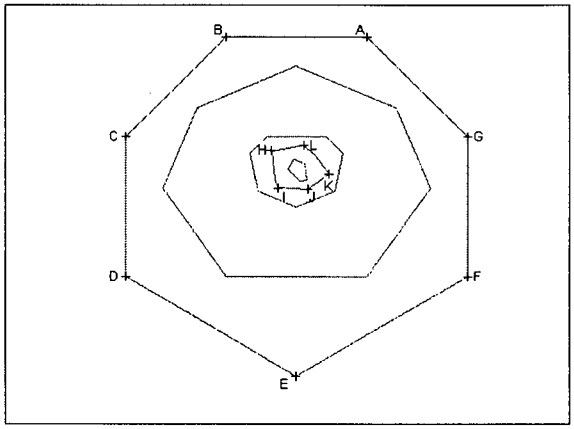


Figure 3-15 0-Depth to 4-Depth Contours of the Dataset

Correctness of FDC

In the above examples, we have demonstrated how Algorithm FDC correctly computes the depth contours of two sample datasets. We are now going to prove that FDC can correctly compute the depth contours of any 2-dimensional dataset, and during the proof, we will study the structural relationships that exist among the depth contours.

Recall that at any stage of Algorithm FDC, the data points in the dataset are classified into two subsets: the outer point set T, and the inner point set D. Among the data points in D, the data points on the convex hull of D are also in the inner convex hull

CH. In what follows, let us call the sets T and CH at the *beginning* of i^{th} iteration of the for-loop in Step 5.5 T_i and CH_i respectively. If we refer to the first example, T_1 and T_2 = $\{A, B, \ldots, G\}$, CH_1 and CH_2 = $\{H, I, \ldots, M\}$, T_3 = $\{A, B, \ldots, M\}$, and CH_3 = $\{N, O, P, Q\}$; and for $i > 3$, T_i = $\{A, B, \ldots, Q\}$ and $CH_i = \varnothing$.

Let us consider T_i and T_{i+1}. $T_{i+1} = T_i$ if and only if the i-intersected inside region contains all the points in the inner convex hull CH_i. For example, in our first example, T_2 = T_1 because the 1-intersected inside region contains CH_1. It is obvious that T_i is sufficient for the computation of DC_i. Since all the points in D are inside CH and CH is inside the i-intersected inside region, all the points in D have depth greater than i and, thus, are not needed for the computation of DC_i.

On the other hand, $T_{i+1} \subset T_i$ if and only if the i-intersected inside region does not contain CH_i. Although many points in D may be outside the i-intersected inside region, we consider only those points that are in CH_i. For the purpose of illustration, we will make use of the example in Figure 3-5 again. In Figure 3-5, data points I, J, and N are all outside the inside region of the 2-divider $<B, E>$. We consider only I and J, but not N, during the expansion of $IR(<B, E>)$ because N is not in CH_2. The points in CH_i are immediately behind the i-intersected inside region. We have to remove CH_i before we can expose the remaining points in D. Therefore, if there are two points p and q, where p is in CH_i and q is not, and both are outside the i-intersected inside region, the depth of q must be greater than the depth of p, which is greater than or equal to i. In other words, the depth of q must be greater than i. For this reason, we need to consider only the points in CH_i in Procedure Expand in order to include all the necessary points.

Our observation confirms that there is a one-to-one correspondence between the series T_1, T_2, ..., T_k and the depth contours DC_0, DC_1, ..., DC_k. To compute DC_i, we need T_i and, where expansion is necessary, certain points in CH_i. We also note that T_{i+1} is exactly T_i and the expanded points (if any). Thus, T_{i+1} provides us with sufficient data points for the computation of DC_i. Namely, DC_i is $\bigcap_{p \in T_{i+1}} IR[p]$. The following theorem summarizes the above.

Theorem (Correctness of Algorithm FDC):

The list of contours of depth 0, 1, 2, ... is the list: T_1, $\bigcap_{p \in T_2} IR[p]$, $\bigcap_{p \in T_3} IR[p]$, ...

Complexity Analysis

We will now divide Algorithm FDC into 4 functional sections and analyze the complexity of each one. The four sections are: 1) computing and maintaining convex hulls, 2) finding initial e-dividers, 3) expanding e-dividers, and 4) intersecting half-planes. And for the complexity analyses, let n be the size of the dataset, k be the maximum number of depth contours desired, t be the size of T_k, and c be the maximum cardinality of the first k CH_i.

1. Computing and Maintaining Convex Hull:

   In Step 1, we compute the convex hull of an n-point dataset from scratch. With standard convex hull computation, Step 1 yields complexity $O(n \log n)$. In Steps 5.3 and 5.5.5.2, we compute the convex hull of the updated dataset. The maintenance of a convex hull of n points after each removal of a data point is only

$O(\log^2 n)$. As there are at most $O(t)$ deletions, the complexity of the maintenance is $O(t \log^2 n)$. Thus, the total complexity of computing and maintaining the convex hulls is $O(n \log n + t \log^2 n)$

2. Finding Initial *e*-Dividers:

   In each iteration of Step 5.5.1.1, we go through t dividers $<p, q>$ to find the *e*-dividers among t points. It requires $O(t^2)$ complexity to find the *e*-dividers for one point. We need to find the *e*-dividers for t points in each iteration of Step 5.5.1, which is executed k times. Thus, the total complexity for finding the initial *e*-dividers is $O(kt^3)$.

3. Expanding *e*-Inside Region:

   In Step 5.5.1.3, we call Procedure Expand to expand an inside region if some point in the inner convex hull is outside the inside region. In Step 5 of Procedure Expand, we check if each line of the expanded series is an *e*-divider. The complexity of finding the *e*-value of a line among t points is $O(t)$. Since there are at most $O(c)$ lines in the expanded series, the complexity of expanding an *e*-divider is $O(ct)$. Procedure Expand is called at most $O(kt)$ times. Thus, the total complexity of expanding the *e*-inside regions is $O(ckt^2)$.

4. Intersecting Half-Planes:

   In each execution of Step 5.5.2, we intersect t half-planes to find the depth contours. The complexity of intersecting t half-planes is $O(t \log t)$. Step 5.5.2 is executed at most k times. Thus, the total complexity of intersecting half-planes is $O(kt \log t)$.

From the above analyses, we know that the overall complexity of FDC is $O(n \log n + t \log^2 n + kt^3 + ckt^2 + kt \log t)$, and we recall that the complexity of ISODEPTH is $O(n^2 \log n + kn^2 + kn^{1.5} \log n)$. Since complexity is a key issue in comparing these two algorithms, the following important question arises: what are the relative magnitudes of c, k, t, and n? Remember that both algorithms set out to identify outliers in large datasets, and that we do not expect outliers to be located in the deep contours. Thus, it is legitimate to assume that k is not going to be large. Although the magnitude of t depends on n and k, in most cases, t is much smaller than n. We have mentioned a test case at the beginning of this chapter: among 100,000 generated data points, fewer than 2% of the data points are to the outside of the 151-depth contour. Furthermore, c is relatively small and is insignificant when compared with t or n. Therefore, we have reasons to believe that FDC will outperform ISODEPTH in finding outliers when n is large and k is not. We will present some experimental results in Chapter 7.

Chapter Four: ENHANCED FDC – FDC-EHNAHCED

In Chapter 3, we have proved that FDC-basic is capable of correctly computing the depth contours of any 2-dimensional dataset without having to process all data points. We have also explained how FDC-basic significantly improves the performance of depth contour computation through its systematic removal and retrieval of data points and the imposition of restrictions on computation so that only selected subsets of data points are worked on. In this chapter, we will present an enhanced version of FDC – FDC-enhanced – that is capable of further reducing the computational time by optimizing FDC-basic.

Sorted Lists

In FDC-basic, the overall complexity is $O(n \log n + t \log^2 n + kt^3 + ckt^2 + kt \log t)$, and Step 5.5.1.1 alone has $O(kt^3)$ time complexity. This step for finding the initial e-dividers dominates the algorithm. Thus, in FDC-enhanced, we aim at optimizing this step. In FDC-basic, the process of finding the initial e-dividers in T_{i+1} is independent of the corresponding process involved in finding the initial e-divider in T_i. We have learned from the previous chapter that T_{i+1} and T_i are closely related because T_{i+1} contains all the data points in T_i. Now we will make use of this close relationship to reduce redundancies in our computation. Specifically, we will carry out an incremental process that works on only those points that are newly added to T_{i+1}.

In FDC-enhanced, which we are going to present below, we will associate a sorted list, $SL[p]$, with every point p in T_i. The sorted list $SL[p]$ stores all points q in T_i,

where $q \neq p$, sorted according to the e-values of the lines passing through p and q. Once the sorted lists for all points p are set up, the e-dividers for point p can easily be read from $SL[p]$ in Step 5.5.1.1.

While finding the initial e-dividers becomes easier now, constructing and maintaining the sorted lists are still challenging tasks. Before we enter the for-loop in Step 5, we have to create the initial sorted lists for all points p in T_1; and then, we must modify the sorted lists systemically whenever we add new data points to T_{i+1}.

Construction of Initial Sorted Lists

The construction of the initial sorted lists is relatively straightforward. Recall that T_1 is the convex hull of the dataset, and each sorted list is a list of points in T_1 sorted in the counter-clockwise direction. Note also that we will use the same example as shown in Chapter 3 to demonstrate the construction of the initial sorted lists and their maintenance. Table 4-1 below shows the initial sorted lists corresponding to $T_1 = \{A, B, \ldots, G\}$. For any point p, the line $<p, SL[p][e]>$ has exactly e points, namely the points in $SL[p][0]$, $SL[p][1]$, ..., $SL[p][e-1]$ to the outside, and the points in $SL[p][e+1]$, $SL[p][e+2]$, ... to the inside. In other words, the line $<p, SL[p][e]>$ is an e-divider. For example, $SL[A][0] = B$ forms a 0-divider (with no point to the outside of $<A, B>$); $SL[A][1] = C$ forms a 1-divider (with B to its outside); $SL[A][2] = D$ forms a 2-divider (with B and C to its outside), and so on.

In this example, since $T_1 = T_2$, the next set of e-dividers can be read from the existing sorted lists. As a result, we can get both the set of 1-dividers $\{<A, C>, <B, D>,$

..., <*G*, *B*>} and the set of 2-dividers {<*A*, *D*>, <*B*, *E*>, ..., <*G*, *C*>} from the initial sorted lists.

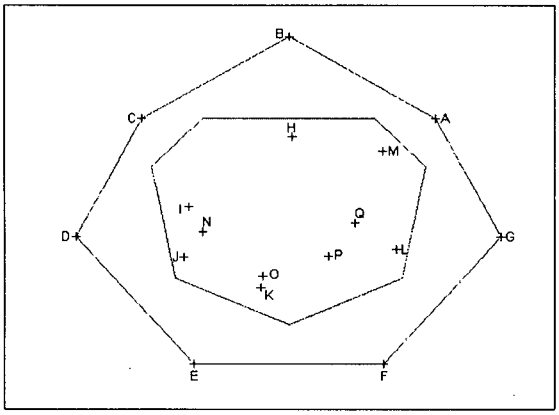


Figure 4-1 0-Depth and 1-Depth Contours

e \ Sorted List For	*A*	*B*	*C*	*D*	*E*	*F*	*G*
0	*B*	*C*	*D*	*E*	*F*	*G*	*A*
1	*C*	*D*	*E*	*F*	*G*	*A*	*B*
2	*D*	*E*	F	*G*	*A*	*B*	*C*
3	*E*	*F*	*G*	*A*	*B*	*C*	*D*
4	*F*	*G*	*A*	*B*	*C*	*D*	*E*
5	*G*	*A*	*B*	*C*	*D*	*E*	*F*

Table 4-1 Initial sorted lists for points {*A*, ..., *G*}

In fact, in all cases, we need not modify any sorted list until we add new data points to T_{i+1}. When new points are added to T_{i+1}, we update the sorted lists in two steps: 1) we insert new points to the existing lists, and 2) we create new lists for the new points. In our example, since T_3 has additional points $CT = \{H, I, ..., M\}$, we must insert points H to M to $SL[A]$, $SL[B]$, ..., $SL[G]$, and create new sorted lists $SL[H]$, $SL[I]$, ..., $SL[M]$.

Before we continue with our example, we want to expand on a few characteristics of the sorted list and its elements. First, since an interior point can have multiple *e*-dividers, an element in a sorted list for an interior point can consist of more than one point; therefore, each element by itself can be a list. Second, the sorted list for an interior point need not to be a complete list. For example, the sorted list for an interior point p that is added during the expansion of an e-divider will have the first e^{th} elements missing because p has depth $\geq e$ and, thus, no divider <p, q> can have an e-value less than e. We will look at some more examples later. Last, based on the above clarification, we shall

expand what we mean by "inserting a point q to $SL[p]$". When we "insert a point q to $SL[p]$ at position i", we insert a list with a single point q before the current i^{th} element of $SL[p]$ so that $SL[p][i] = \{q\}$, and the original i^{th} as well as the subsequent elements of $SL[p]$ are pushed down. When we "insert a point q to $SL[p][i]$", we insert a single point q to the list $SL[p][i]$, and the positions of the elements in $SL[p]$ remain unchanged.

Incremental Maintenance of Sorted Lists

We will now go back to our discussion about the addition of new points and the creation of new lists. In general, the addition of a new point q to an existing list $SL[p]$, where $<p, q>$ is an i-divider, involves the insertion q to $SL[p]$ at position i, and the *re-arrangement* of the points in $SL[p]$ that are no longer in their correct positions.

The need for re-arrangement is not always apparent. For instance, the insertion of q to $SL[p]$ does not apparently affect the positions of the points in $SL[p][j]$ where $j < i$. However, the addition of q to T may have change the e-values of some of these points, as when q is indeed to the outside of certain dividers $<p, r>$ where r is a point in $SL[p][j]$. Thus, there exists a real but undetected need for re-arrangement, and the affected points must be pushed down so that all points will be in line with their e-values.

In other instances, as in $SL[p][j]$ where $j \geq i$, the insertion of q to $SL[p]$ obviously requires all points in $SL[p][j]$ to be pushed down. Yet, in these instances, the e-values of some of the points may not have been affected by the addition of q, that is, q is indeed to the inside of certain dividers $<p, r>$ where r is a point in $SL[p][j]$. For this reason, the unaffected points should be pushed up back to their original positions.

The only occasion when the re-arrangement of point positions can be readily predicted relates to the maintenance of the sorted list for a boundary point. Points in the sorted list for a boundary point p always remain in their correction positions even after the insertion of a new point q because here p has a single e-divider for all values of e. All the dividers for p follow a specific sequence – if we rotate a line directed from p in the counterclockwise direction, we will encounter the 0-divider, the 1-divider, ... in that order. Therefore, the insertion of q into $SL[p]$ at position i will affect all the points in $SL[p][j]$ where $j \geq i$ and leave all the points in $SL[p][j]$ intact where $j < i$.

Unfortunately, this unique case does not apply to an interior point. An interior point may have multiple e-dividers, and each e-dividers may have a different set of points to its outside; hence, one e-divider may need expansion while another may not. Re-arrangement of point positions, therefore, requires careful handling.

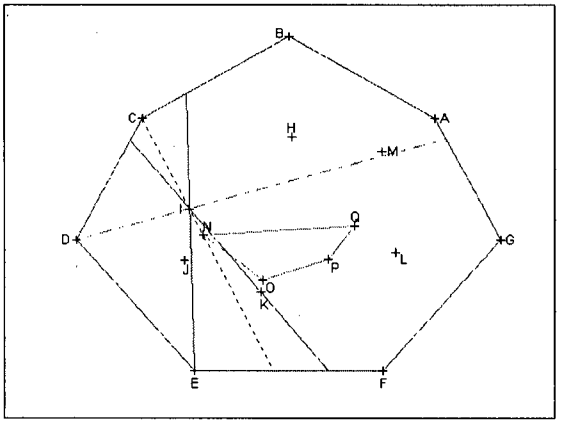


Figure 4-2 3-Dividers <*I*, *E*> and <*I*, *N*>; 4-Divider <*I*, *K*>; 5-Dividers <*I*, *D*> and <*I*, *F*>

e	*SL[I]*				
0	--		--		--
1	--		--		--
2	*J*		*J*		*J*
3	*E, K*		*N*		*E, N*
4	*F*		*E, K*		*K*
5	*D, L*	→	*F*	→	*D, F*
6	*G, M*		*D, L*		*L*
7	*A*		*G, M*		*G, M*
8	*C, H*		*A*		*A*
9	*B*		*C, H*		*C, H*
10	--		*B*		*B*

Table 4-2 Inserting *N* into *SL[I]*

Now, consider the example in Figure 4-2 that illustrates how FDC-enhanced handles the insertion of data point N into the sorted list for data point I. Point I has two 3-dividers: <I, E> and <I, K>. IR(<I, K>) does not contain data point N. Thus, <I, K> is

expanded to <*I*, *N*, *K*>. When we insert *N* to *SL*[*I*] at position 3, we push both *E* and *K* down to the 4th position. However, *N* is to the inside of <*I*, *E*> and does not affect the *e*-value of <*I*, *E*>. Therefore, *E* should remain in the 3rd position. Similarly, 5-divider <*I*, *D*> is pushed down to the 6th position. But, since *N* is to the inside of <*I*, *D*>, *D* should remain in the 5th position.

Inserting new points *q* into the existing sorted lists, and re-arranging the positions of the points, if necessary, complete one step in the maintenance of the sorted lists. The other step is the creation of a new listed list for each new point. In fact, these two steps – the creation of *SL*[*q*] and the insertion of *q* into the existing lists – are closely related because we know that for any pair of data points *a* and *b*, the position of *b* in *SL*[*a*] is the opposite of the position of *a* in *SL*[*b*]. Therefore, in the example in Figure 4-1, *B* is the first element in *SL*[*A*] and *A* is the last element in *SL*[*B*]. For the creation of *SL*[*q*], if we know the position of *q* in *SL*[*p*], we will also know the position of *p* in *SL*[*q*]. In other words, the new sorted list *SL*[*q*] can be created simultaneously with the insertions of *q* into the existing lists. We capitalize on this idea to speed up FDC-enhanced.

We can now look at one more way to achieve greater efficiency. As the *e*-value of a divider can only increase when new points are added, a divider will never be used in the computation of the first *k*-depth contours once its *e*-value exceeds *k*. Therefore, to optimize the maintenance procedure, the sorted lists need to keep only the first *k* dividers. A new point *q* need not be inserted to *SL*[*p*] if <*p*, *q*> has an *e*-value greater than *k*. On the other hand, the *e*-value of an *e*-divider will not change after the generation of DC_e because the inside region of the divider contains all the points in *CH*; and thus, all the points in the inner point set. Since an *e*-divider does not contribute to the construction of

depth contours whose depths are greater than e, the sorted lists can remove the dividers that are no longer in use.

In light of these findings, Step 5.5.5.1 in the original Algorithm FDC is replaced by the following maintenance procedure.

```
5.5.5   If ⋃_{p∈T} CT[p] ≠ ∅ {
  5.5.5.1   For all new points q ∈ CT {
    5.5.5.1.1   SL[q] = ∅;
    5.5.5.1.2   For all points p ∈ T {
      5.5.5.1.2.1   i = e-value of <p, q>, j = e-value of <q, p>;
      5.5.5.1.2.2   If i ≤ k, insert q to SL[p] at position i;
      5.5.5.1.2.3   For ( l = e, l < i, l++) {
        5.5.5.1.2.3.1   For all points r in SL[p][l],
                        If q is outside <p, r>, move r to SL[p][l+1];
      }  /* end for */
      5.5.5.1.2.4   For (l = i+1, l < k, l++ ) {
        5.5.5.1.2.4.1   For all points r in SL[p][l],
                        If q is inside <p, r>, move r to SL[p][l-1];
      }  /* end for */
      5.5.5.1.2.5   If j ≤ k, insert p to SL[q][j];
    }  /* end for */
    5.5.5.1.3   Remove q from D and Add q to T;
  }  /* end for */
  5.5.5.2   CH = vertices of the convex hull of (the updated) D;
}  /* end if */
```

Demonstration

Let us compute the first 5 depth contours of the dataset shown in Figure 4-1 using the enhanced algorithm. The first two depth contours are already computed and the sorted lists are shown as Table 4-1. We are now at the second iteration of Step 5.5.5 where T_2 =

$\{A, B, \ldots, G\}$ and $CT = \{H, I, \ldots, M\}$. Since CT is not empty, we proceed to Step 5.5.5.1 and update the sorted lists following the maintenance procedure.

In the first iteration of Step 5.5.5.1, we insert data point H to all the existing sorted lists and create $SL[H]$ by first initializing $SL[H]$ with an empty list. We start with point A in Step 5.5.5.1.2. $<A, H>$ is a 2-divider and $<H, A>$ is a 4-divider as shown in Figure 4-3.

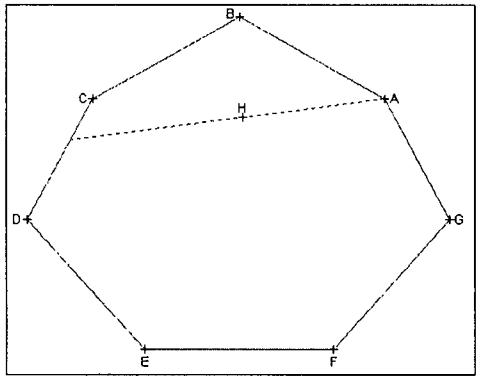


Figure 4-3 2-Divider <A, H>; 4-Divider <H, A>

e	*SL[A]*				*SL[H]*		
2	*D*	→	*H*		--	→	--
3	*E*		*D*		--		--
4	*F*		*E*		--		*A*
5	*G*		*F*		--		--

Table 4-3 Inserting *H* into *SL[A]* and *A* into *SL[H]*

Because the e-value of $<A, H>$ is less than k (5), we insert H to $SL[A]$ at position 2 in Step 5.5.5.1.2.2. Since $SL[A][2]$ is already the first element in the sorted list, we skip Step 5.5.5.1.2.3 and proceed to Step 5.5.5.1.2.4. H is outside $<A, D>$, $<A, E>$ and $<A, F>$. No points need to be moved. We insert A to $SL[H][4]$ in Step 5.5.5.1.2.5.

	SL[A]	*SL[B]*	*SL[C]*	*SL[D]*	*SL[E]*	*SL[F]*	*SL[G]*	*SL[H]*
2	*H*	*E*	*F*	*G*	*A*	*B*	*H*	*C, D*
3	*D*	*H*	*G*	*A*	*H*	*H*	*C*	*B, E, F*
4	*E*	*F*	*H*	*H*	*B*	*C*	*D*	*A, G*
5	*F*	*G*	*A*	*B*	*C*	*D*	*E*	--

Table 4-4 Sorted lists for $T = \{A, \ldots, H\}$

We insert H to $SL[B]$, ..., $SL[G]$ and update $SL[H]$ using the same steps. At the end of the first iteration, we remove H from D and add H to T in Step 5.5.5.1.3. The sorted lists $SL[A]$, $SL[B]$, ..., $SL[H]$ are as shown in Table 4-4.

Then we insert I to the existing sorted lists and create $SL[I]$. We first insert I to $SL[A]$, then to $SL[B]$, and so on. Finally we insert I to $SL[H]$ as shown in Table 4-5. Since $<H, I>$ is a 3-divider, I is inserted to $SL[H]$ at position 3. B, E and F are pushed down to the 4th position; A and G are pushed down to the 5th position. In Step 5.5.5.1.2.3, the points above the newly added point I, which corresponds to $SL[H][2]$ in this case, are checked. $SL[H][2]$ has points C and D. Because I is to the inside of both $<H, C>$ and $<H, D>$, C and D need not be re-arranged. In Step 5.5.5.1.2.4, the points after I, which are points in $SL[H][4]$ and $SL[H][5]$, are checked. Since I is to the inside of $<H, B>$, B is pushed up from $SL[H][4]$ to $SL[H][3]$. From Figure 4-4, we see that $<H, B>$ is indeed a 3-divider and should remain in $SL[H][3]$. Table 4-6 shows the sorted lists for all points after the addition of I.

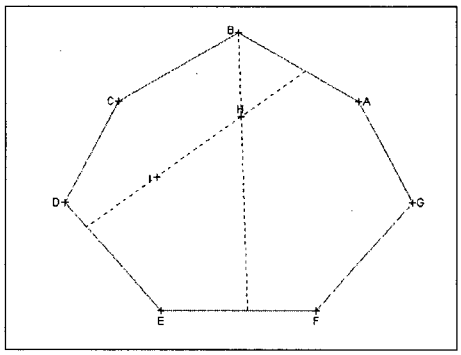


Figure 4-4 3-Dividers $<H, B>$ and $<H, I>$

e	*SL[H]*				
2	*C, D*		*C, D*		*C, D*
3	*B,E,F*	→	*I*	→	*B, I*
4	*A, G*		*B,E,F*		*E, F*
5	--		*A, G*		*A, G*

Table 4-5 Inserting *I* into *SL[H]*

e	*SL[A]*	*SL[B]*	*SL[C]*	*SL[D]*	*SL[E]*	*SL[F]*	*SL[G]*	*SL[H]*	*SL[I]*
2	*H*	*I*	*I*	*G*	*A*	*B*	*H*	*C, D*	*E, F*
3	*D*	*E*	*F*	*I*	*H*	*H*	*C*	*B, I*	*A, G*
4	*I*	*H*	*G*	*A*	*B*	*C*	*I*	*E, F*	*D, H*
5	*E*	*G*	*H*	*H*	*I*	*I*	*D*	*A, G*	*B, C*

Table 4-6 Sorted lists for $T = \{A, \ldots, I\}$

Next we insert J to the existing sorted lists and create $SL[J]$. When H and I were being inserted to T earlier, the size of T was small and the maximum depth of the dividers was less than $k = 5$. Now when J is inserted to T, T has 10 data points and some dividers have depth greater than 5. Figure 4-5 shows all the dividers passing through J.

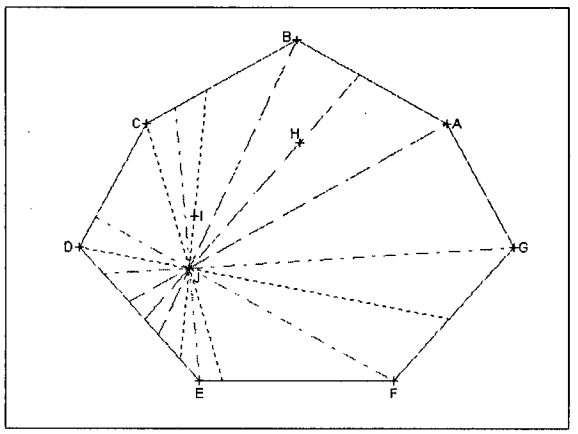


Figure 4-5 All dividers for *J*

<*A*, *J*> is a 5-divider and <*J*, *A*> is a 3-divider. Both dividers are inserted to the sorted lists *SL*[*A*] and *SL*[*J*] respectively. Similarly, <*B*, *J*>, <*J*, *B*>, <*H*, *J*>, and <*J*, *H*> are inserted. <*C*, *J*> is a 2-divider but <*J*, *C*> is a 6-divider. So *J* is insert to *SL*[*C*] at position 2, but *C* is not inserted to *SL*[*J*]. Likewise, *J* is inserted to *SL*[*D*] and *SL*[*I*], but *D* and *I* are not added to *SL*[*J*]. Conversely, <*E*, *J*> is a 6-divider, but <*J*, *E*> is a 2-divider. So *J* is not inserted to *SL*[*E*] while *E* is inserted to *SL*[*J*][2]. For similar reasons,

J is not inserted to $SL[F]$ and $SL[G]$, but F and G are inserted to $SL[J][2]$. Table 4-7 below shows the sorted lists after the insertion of J. Note that both points D and C are pushed up in $SL[I]$ because they are outside $<I, J>$.

e	*SL[A]*	*SL[B]*	*SL[C]*	*SL[D]*	*SL[E]*	*SL[F]*	*SL[G]*	*SL[H]*	*SL[I]*	*SL[J]*
2	*H*	*I*	*J*	*J*	*A*	*B*	*H*	*C, D*	*J*	*E,F,G*
3	*D*	*J*	*I*	*G*	*H*	*H*	*C*	*B, I*	*E, F*	*A*
4	*I*	*E*	*F*	*I*	*B*	*C*	*I*	*J*	*A,D,G*	*H*
5	*J*	*H*	*G*	*A*	*I*	*I*	*D*	*E, F*	*H, C*	*B*

Table 4-7 Sorted lists for $T = \{A, ..., J\}$

The rest of the maintenance procedure follows the same pattern. Before we present the finalized sorted lists, we will show an example in which we push a point down in Step 5.5.5.1.2.3. We are going to insert data point K to $SL[J]$. As shown in Figure 4-6, $<J, K>$ is a 3-divider. K is inserted to $SL[J]$ at 3. E, F, and G are the points in $SL[J][2]$, which are above K. Yet, $<J, G>$ is indeed a 3-divider. K is found to be to the outside of $<J, G>$. Therefore, G is pushed down to the 3rd position.

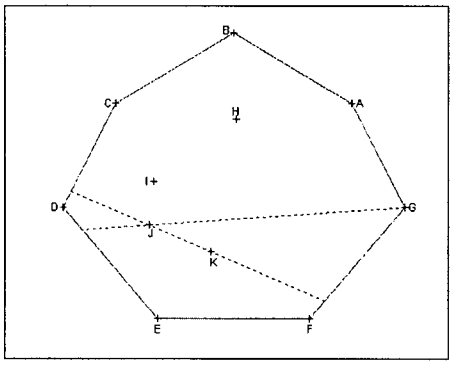


Figure 4-6 3-Dividers $<J, G>$ and $<J, K>$

e	*SL[J]*				
2	*E,F,G*		*E,F,G*		*E, F*
3	*A*	→	*K*	→	*G, K*
4	*H*		*A*		*A*
5	*B*		*H*		*H*

Table 4-8 Inserting K into $SL[J]$

After the last iteration of Step 5.5.5.1, we compute the convex hull of the updated D in Step 5.5.5.2 as in FDC-basic. At the end of Step 5.5.5, we obtain the following sorted lists for all points in $T = \{A, \ldots, M\}$.

e	$SL[A]$	$SL[B]$	$SL[C]$	$SL[D]$	$SL[E]$	$SL[F]$	$SL[G]$	$SL[H]$	$SL[I]$	$SL[J]$	$SL[K]$	$SL[L]$	$SL[M]$
2	*H*	*I*	*J*	*K*	*L*	*L*	*M*	*C,D*	*J*	*E,F*	*G*	*A,M*	*B,H*
3	*D*	*J*	*I*	*J*	*A*	*M*	*H*	*I*	*E,K*	*K,L*	*F.L*	*B,G*	*C*
4	*I*	*E*	*K*	*L*	*M*	*B*	*C*	*J*	*F*	*G*	*A*	*H*	*D*
5	*J*	*K*	*F*	*G*	*K*	*H*	*I*	*B,E*	*D,L*	*M*	*H,M*	*C*	*A,I*

Table 4-9 Sorted lists for $T = \{A, \ldots, M\}$

We use the same maintenance procedure to update the sorted lists when we merge CH with T in Step 5.2. We treat the data points in CH as the new points in CT. For each point q in CH, we insert q into the existing sorted list and create a new sorted list $SL[q]$.

Complexity Analysis

How much faster is FDC-enhanced compared with FDC-basic in computing the first k depth contours of an n-point dataset? In FDC-basic, finding the initial e-dividers for each data point in Step 5.5.1.1 requires time complexity $O(t^2)$, where t is the number of data points in the final outer point set T. Since Step 5.5.1.1 is executed kt times, the total complexity is $O(kt^3)$. In FDC-enhanced, the initial e-dividers for each point p are the lines directed from p to the points in $SL[p][e]$. Therefore, the time required to find the initial e-dividers for a data point is simply the time required to read the points from the e^{th} element in a sorted list, which is $O(1)$. Thus, the total complexity for Step 5.5.1.1 is $O(kt)$.

However, FDC-enhanced does introduce an extra procedure to maintain the sorted lists in Step 5.5.5.1. When a new point q is inserted into a sorted list $SL[p]$, the computation of the e-value of $<p, q>$ in Step 5.5.5.1.2.1 requires $O(t)$ time, the insertion of q to $SL[p]$ in Step 5.5.5.1.2.2 and that of p to $SL[q]$ in Step 5.5.5.1.2.5 require $O(1)$ time, and the re-arrangement of points in $SL[p]$ in Steps 2.6.5.1.2.3 and 2.6.5.1.2.3 requires $O(k)$ time. Thus, the complexity of one iteration of Step 5.5.5.1.2 is $O(t + k)$. Since a new point is inserted into at most t sorted lists and the total number of new points is at most t, the total complexity of Step 5.5.5.1 is $O(t^3 + kt^2)$.

FDC-enhanced requires a total complexity $O(t^3 + kt^2 + kt)$ to find the initial e-dividers while continuously updating the sorted lists, whereas FDC-basic requires $O(kt^3)$ merely to find the initial e-dividers. When k is small, FDC-enhanced may not excel FDC-basic. However, when k gets larger, the improvement becomes obvious. We will see more comparisons between FDC-basic and FDC-enhanced in Chapter 7.

Chapter Five: FDC WITH BUFFER SPACE CONSTRAINTS

We have explained and showed with examples why our Algorithm FDC can work faster in computing depth contours. However, both the basic version and the enhanced version of Algorithm FDC are main-memory algorithms; and, so far, we have assumed that the computer system has sufficient buffer space to handle the entire computation. On the other hand, we know that data mining applications almost always deal with a large volume of data. What can we do if the buffer space can accommodate only a portion of the dataset? In anticipation of this buffer space problem, we have come up with three different solutions based on three different divide-and-conquer techniques. The central idea is to divide the original dataset D into smaller subsets, find the outer point set of each subset S_i, and compute the depth contours of the union, S, of all the outer point sets. We call the three variants of FDC-basic that we have developed FDC-M1, FDC-M2, and FDC-M3. Although these three variants have been developed with FDC-basic, they can be enhanced to work with FDC-enhanced.

Computation with Limited Buffer Space

We begin our introduction of the three variants with a more precise definition of the problem: Given a dataset D of n points of which t points are outside the k-depth contour, compute the first k+1st depth contours of D within the constraint of a buffer space which can accommodate only m data points, where $t < m < n$.

While the buffer space need not be large enough to hold all n data points, it must nevertheless be large enough to accommodate at least all t data points. Otherwise, we cannot correctly compute the depth contours.

We observe that for any point p in a subset S_i, if p is inside the k-depth contour of S_i, p must also be inside the k-depth contour of the original dataset. This is because the depth of p can only increase as more points are added to S_i. Since S_i is a subset of the original dataset, if p has depth $> k$ in S_i, p must also have a depth $> k$ in the original dataset. In other words, to compute the first k^{th} depth contours of the original dataset, we need only the k+1st outer point sets of the subsets, for these outer point sets contain all the data points with depths $\leq k$.

Among the three variants of FDC-basic mentioned above, FDC-M1 is the basic method, FDC-M2 is enhanced with *batch* merging, and FDC-M3 is further enhanced with *full merging*. All three methods start with the division of the original dataset into smaller subsets. For the sake of simplicity[2], we divide up the data points by their physical order: we put the first m points into subset S_1, the next m points into subset S_2, and carry on this division up to the last subset S_l, where $l = \lceil n/m \rceil$. Figure 5-1 shows the first 2 depth contours of a sample dataset D of 60 data points. These depth contours are generated by FDC-basic on the basis that the buffer size is large enough for the computation. Figure 5-2 demonstrates how FDC-M1, FDC-M2, and FDC-M3 divide the same dataset into two subsets S_1 and S_2 for computing the first 2 depth contours when the buffer space can accommodate the computation of only 30 data points.

[2] Although a different way of dividing up the data points may affect the performance, the resulting depth contours will remain the same. We will further investigate the use of different divisions in future.

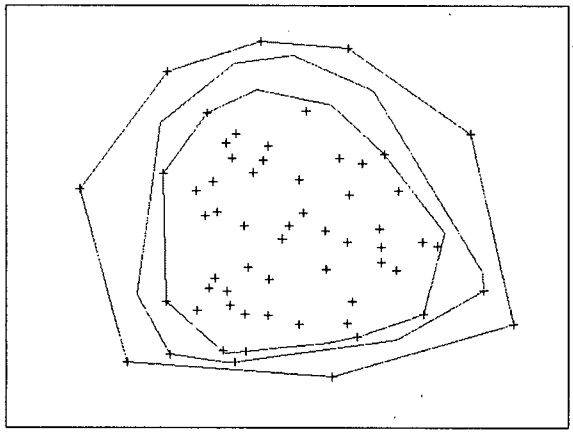


Figure 5-1 0-Depth to 2-Depth Contours of a 60-point dataset

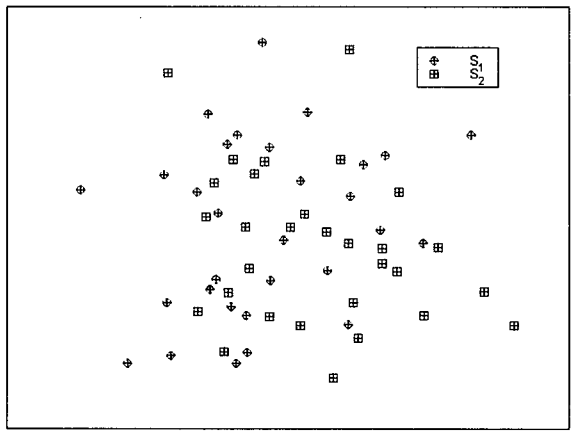


Figure 5-2 S_1 and S_2

FDC-M1: Basic

Algorithm FDC-M1(D, k, m)

Input: D = the dataset, k = an integer, and m is the maximum buffer space size.

Output: Contours of depth from 0 to k.

1 $S = D$;

2 While ($|S| > m$) {

2.1 Divide the dataset S into smaller subsets $S_1, \ldots, S_l$;

2.2 Initial S with $\varnothing$;

2.3 For ($i = 1; i < l; i$++) {

2.3.1 Call FDC-basic to compute the k+1st outer point set of S_i;

2.3.2 $S = S \cup$ the k+1st outer point set of S_i;

} /* end for */

} /* end while */

3 Compute the first k depth contours of S using FDC-basic;

We now consider the sample dataset shown above for the purpose of demonstrating the operation FDC-M1. In Step 2.1, we obtain S_1 and S_2. Then we find the k+1st outer point sets of S_1 and S_2 in the first and the second iteration of Step 2.3.1. Step 2.3.1 in FDC-M1 is quite similar to the process in FDC-basic for computing the first

k depth contours of S_i. However, Step 2.3.1 of FDC-M1 stops short of actually working out the depth contours by skipping Step 2.6.2 of FDC-basic – intersecting the inside regions; and returns, instead, the outer point set at the end of the computation. Figure 5-3 shows the 2-depth contours and the outer point sets of S_1 and S_2 respectively. S_1 has 17 outer data points and S_2 has 16. Thus, S has 33 points altogether as shown in Figure 5-4.

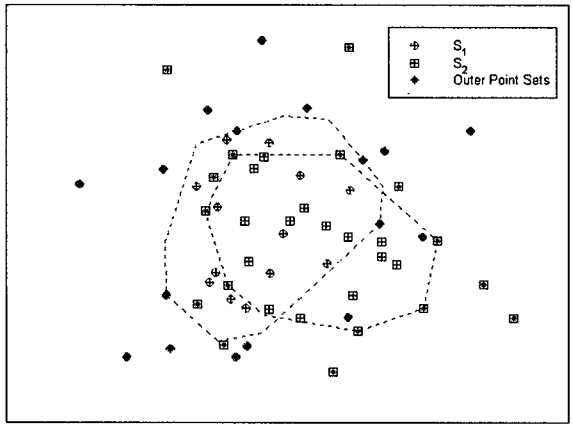


Figure 5-3 2-Depth Contours and Outer Point Sets of S_1 and S_2

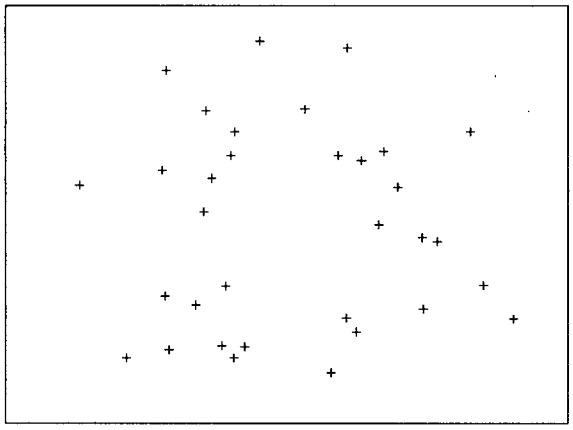


Figure 5-4 Updated S has 33 data points

As S still exceeds our buffer limit of 30 data points, further pruning of S is necessary. We, therefore, repeat Step 2. Here again, while other methods of division will yield the same result, for the sake of simplicity, we divide up the 33 data points by their original physical order, putting the first 30 points of D into S_1 and the remaining 3 points into S_2 as shown in Figure 5-5. Of the 30 points in S_1, 18 points have depths ≤ 2; all three points in S_2 have depth 0[3]. The 18 points in S_1 and the 3 points in S_2 make up S, which now has 21 points in total and does not exceed the buffer limit.

[3] A 3-point dataset is a special case because all of its data points are on the outermost convex hull and automatically have depth 0; we need not even compute the outer point set with Algorithm FDC in this special case.

Then, we compute the depth contours of these 21 points in Step 3. Figure 5-7 shows the first two depth contours of S. These depth contours are the same as the ones shown in Figure 5-1 computed by FDC-basic.

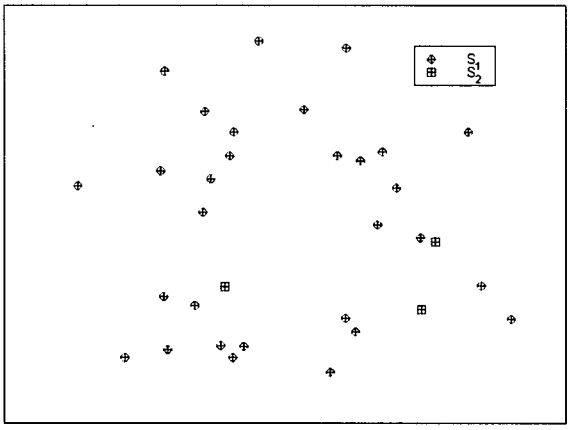


Figure 5-5 S_1 and S_2

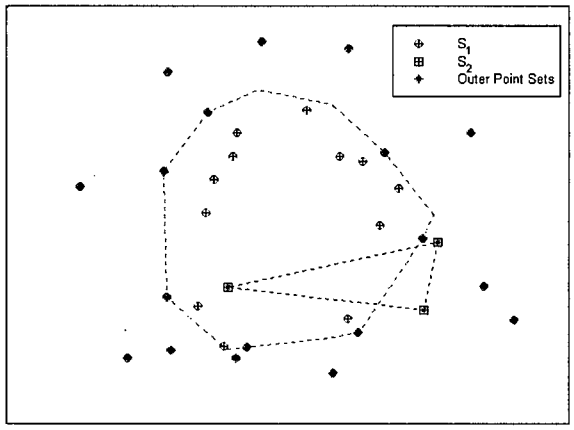


Figure 5-6 2-Depth Contours and Outer Point Sets of S_1 and S_2

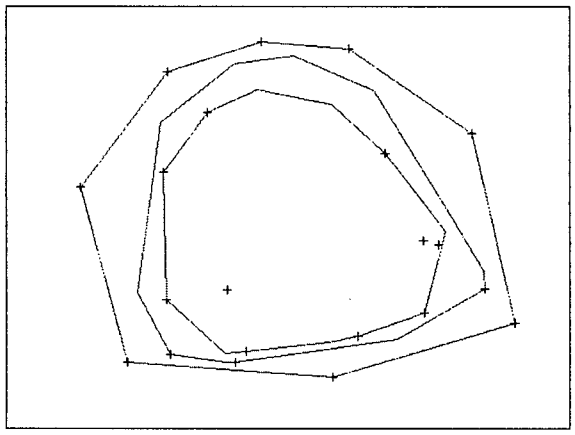


Figure 5-7 0-Depth to 2-Depth Contours of S with 21 data points

FDC-M2: Batch Merging

FDC-M1 treats all data points in the outer point sets equally. S is the union of all the outer point sets. However, certain of these points will definitely not appear in any of the desired depth contours, namely, those points that are inside the k+1st inner convex hull of

any subset. FDC-M2 reduces the number of data points to be added to S by first filtering the data points in the outer point sets. It uses the k+1st inner convex hull of S_1 as the controlling convex hull, *CCH*, to determine which data points should be added to S and which ones should be excluded. Data points inside *CCH* have depths greater than k. Thus, all data points in the outer point sets that are inside *CCH* are too deep to be added to S.

Algorithm FDC-M2(D, k, m)

Input: D = the dataset, k = an integer, and m is the maximum buffer space size.

Output: Contours of depth from 0 to k.

1 Divide the dataset D into smaller sets $S_1, \ldots, S_l$;

2 Call FDC-basic to compute the k+1st inner convex hull and the k+1st outer point set of S_1;

3 *CCH* = the k+1st inner convex hull of S_1; S = the k+1st outer point set of S_1;

4 For ($j = 0; j < k; j$++) Initialize ST_j with $\varnothing$;

5 For ($i = 2; i < l; i$++) {

  5.1 For ($j = 0; j \le k; j$++) {

    5.1.1 Call FDC-basic to compute the k+1st outer point set of S_1;

    5.1.2 $ST_j = ST_j \cup$ the data points of depth j in S_i;

  } /* end for */

} /* end for */

6 For ($j = 0, j \le k; j$++) {

  6.1 For every point p in ST_j {

    6.1.1 If p is inside *CCH*, continue;

    6.1.2 /* Otherwise */ $S = S \cup \{p\}$;

    6.1.3 If ($|S| \ge m$) {

      6.1.3.1 Call FDC-basic to compute the k+1st inner convex hull and the k+1st outer point set of S;

      6.1.3.2 *CCH* = the k+1st inner convex hull of S; S = the k+1st outer point set of S;

    } /* end if */

  } /* end for */

} /* end for */

7 Compute the first k depth contours of S using FDC-basic;

Because we have picked the inner convex hull of S_1 as the controlling *CH*, the outer point set of S_1 becomes the initial *S*. The data points in the other outer point sets are organized by depth in Step 4.1.1. In Step 5, these data points will be added to *S* provided they are outside *CCH*. Data points with the smallest depth will be added to *S* first followed by data points with the second smallest depth, and so on. Thus, FDC-M2 ensures that the "more useful" points – those points outside and farthest away from *CCH* – will always be added to *S* first. As for those points inside *CCH*, FDC-M2 stows them away as "useless" points right away.

The unique step taken by FDC-M2, however, is *Batch Merging*, which distinguishes it from the other variants of our Algorithm FDC; and it is because of this unique step that FDC-M2 can become more efficient than FDC-M1.

After adding each new data point to *S*, FDC-M2 finds out whether *S* is within the set buffer limit *m* or not. Once the size of *S* reaches *m*, FDC-M2 will compute the $k+1^{st}$ outer point set of *S* as well as the $k+1^{st}$ inner convex hull of *S*. Then, FDC-M2 will re-set *S* to the outer point set and *CCH* to the inner convex hull (Step 5.1.3.1). Thus, a new initial *S* is in place. Because of the addition new points to *S*, the updated *CCH* will be enlarged; and, therefore, more "useless" data points can be discarded or filtered.

With *S* as the new initial *S*, FDC-M2 now continues to search the remaining data points in the outer point sets to select the next batch of data points for merging with *S*. Batch merging coupled with the continual updating of *CCH* increases the efficiency of FDC-M2. The cycle is complete when all the "useful" points in the other outer point sets have been added to *S*.

The final step in FDC-M2 involves the computation of the requisite depth contours, which is exactly the same as the final step in FDC-M1.

We will use the same sample dataset in Figure 5-1 to show how FDC-M2 computes the first two depth contours. As does FDC-M1, FDC-M2 divides the dataset into S_1 and S_2 as shown in Figure 5-2. However, instead of setting S to the union of the outer point sets of S_1 and S_2, FDC-M2 sets the initial S to the outer point set of S_1 and divides up the data points in the outer point set of S_2 by depth into ST_0, ST_1, and ST_2. Figure 5-8 shows the initial S, the controlling CH, ST_0, ST_1 and ST_2. Initial S has 17 points. ST_0, ST_1 and ST_2 have 6, 6 and 4 points respectively.

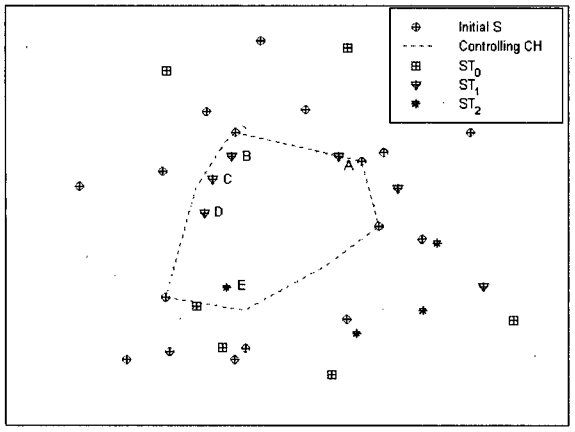


Figure 5-8 Initial *S*, Controlling *CH*, and Outer Point Set of S_2

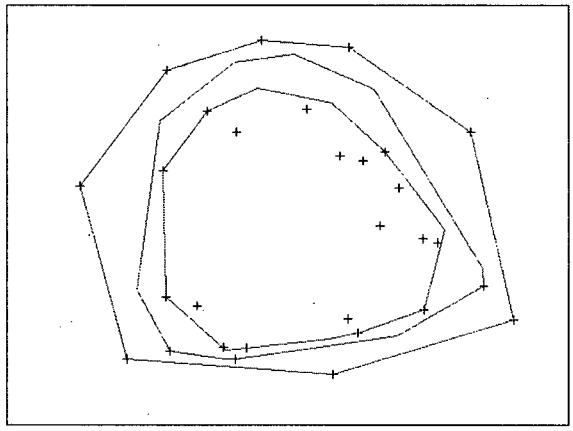


Figure 5-9 0-Depth to 2-Depth Contours of *S* with 28 data points

In the first iteration of Step 7, we find that all the points in ST_0 are outside CCH; therefore, we add all these points to S, which now has 23 points. Then we check the points in ST_1. As data points A, B, C and D are inside CCH, only two points in ST_1 are added to S. By the end of the second iteration, S has 25 points. In the next iteration, we find that data point E is inside CCH; so all the points in ST_3 but E are added to S. After all the useful data points are added to it, S has only 28 data points, which does not exceed

the buffer limit. Finally, we compute the first two depth contours of S (Figure 5-9), which again match those in Figure 5-1 and Figure 5-7.

FDC-M3: Full Merging

When FDC-M2 carries out batch merging in the intermediate steps, it does not simply pick all the data points that have a depth $\leq k$ in each subset for addition to S as FDC-M1 does. It discards such outer points as are within *CCH*, so as to reduce the number of data points to be added to S for computing the required depth contours – the final computation.

However, both FDC-M1 and FDC-M2 use the information about the depth of points only during the intermediate steps. Such information helps FDC-M1 and FCD-M2 to determine which points are to be added to S; but once a point has been added to S, both algorithms ignore the depth of that point on the basis that the point can assume new depths in the expanding S. However, with FDC-M3, the depth of a point in any subset becomes the basis for relating that point to other points within the same subset. When FDC-M3 has worked out the depth of a point p, it associates p with its *next convex hull* – $nextCH(p)$ – the set of points of immediate behind p. The next convex hull of a point p of depth i is the inner convex hull CH_{i+1}, that is, $nextCH(p) = CH_{i+1}$; and the next convex hulls of all the points of the same depth are the same. FDC-M3 places special emphasis on this relationship between a point and its next convex hull because it intends to use this relationship in the final computation; and for this reason, it retains all such relational information and all information about the depth of points.

How can such information help FDC-M3 minimize the number of data points required to compute the depth contours? The answer lies in the fact that a different concept is adopted in designing FDC-M3. In designing the previous two variants, we try to obtain an S that contains all the data points of the desired depths before starting the final computation. This means that S will look like the complete original dataset without the points of depths greater than k. In designing FDC-M3, we want to start with the smallest S that contains only those points required for computing the 0-depth contour. This concept of working on an "incomplete" S avoids the need to maintain a mixed bag of data points of different depths, some of which are not yet due to be worked on. It is only when FDC-M3 has completed the computation of the 0-depth contour that it will merge the next minimum set of data points with the data points inside the 0-depth contour to form another S. At that point, the union of the all the next convex hulls of the points comprising the 0-depth contour will give the next minimum set. As FDC-M3 tries to complete the "incomplete" S in this manner while working on the final computation, the depth and relational information it generates in the intermediate step becomes a tremendous help.

Before we present our Algorithm FDC-M3 and demonstrate its operation with the help of an example, we will explain how FDC-M3 makes use of the Full Merging approach. To achieve its goal of minimizing the number of data points to be included in the final computation, FDC-M3 retrieves only the points of 0-depth from all the subsets and merges them to create the initial S for computing the 0-depth contour of the entire original dataset. This is the first round of merging performed by FDC-M3, and it aims at merging all the requisite data points in one go instead of merging them only a batch at a

time. For the next round of full merging, it will retrieve from all the subsets the next convex hulls of only those data points that are on the 0-depth contour, which is identical with the convex hull of the initial S. All such next convex hulls will then be merged in one go with the 0-depth data points that are not on the convex hull of the initial S. The resultant dataset becomes the next S for another round of final computation—the computation of the 1-depth contour of the entire original dataset. Full Merging to form an "incomplete" S distinguishes FDC-M3 from FDC-M2.

Algorithm FDC-M3(D, k, m)

Input: D = the dataset, k = an integer, and m is the maximum buffer space size.

Output: Contours of depth from 0 to k.

1 Divide the dataset D into smaller sets $S_1, \ldots, S_l$;

2 Initialize S with $\varnothing$;

3 For ($i = 1; i < l; i$++) {

3.1 Call FDC-basic to compute the first k inner convex hulls and the k+1st outer point set of S_i;

3.2 For ($j = 0; j < k; j$++) {

3.2.1 $ST_{i,j}$ = the data points of depth j in S_i;

3.2.2 $nextCH_{i,j}$ = the j+1st inner convex hull of S_i;

3.2.3 For every data point p in $ST_{i,j}$,

3.2.3.1 $nextCH(p) = nextCH_{i,j}$;

} /* end for */

} /* end for */

4 $S = \cup_{1 \le i \le l} ST_{i,0}$; /* initial S */

5 Run FDC-basic with the following change in Step 5.2 and Step 5.5.5 (when we update S and T):

5.1 For every point p to be moved from S to T {

5.1.1 $T = T \cup \{p\}$;

5.1.2 $S = S - p \cup nextCH(p)$;

5.1.3 If ($|S| > m$) {

5.1.3.1 i = current depth contour; $j = k - i + 1$;

5.1.3.2 Call FDC-basic to compute the j+1st outer point set of S;

5.1.3.3 Reset S to the j+1st outer point set;

} /* end if */

} /* end for */

We now consider again the sample dataset in Figure 5-1, which is divided into S_1 and S_2 as shown in Figure 5-2. For each of these subsets, we compute the outer point sets and the next convex hulls of all the data points in the outer point set. Figures 5-10 and 5-11 show the outer point sets and the different next convex hulls for S_1 and S_2, respectively.

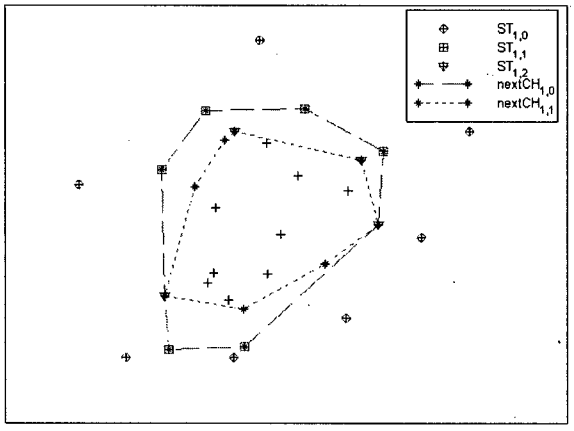


Figure 5-10 Outer Point Set and Next Convex Hulls of S_1

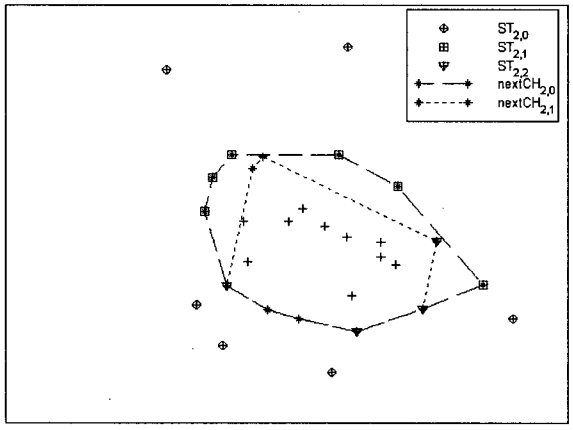


Figure 5-11 Outer Point Set and Next Convex Hulls of S_2

Because $ST_{1,0}$ contains the outermost data points of S_1, and $ST_{2,0}$ contains those of S_2, their union includes all of the outermost data points of the original dataset and is sufficient for the initial S (13 data points). The convex hull of the initial S is the 0-depth contour, and it gives us the first outer point set T (8 data points). As all the points in T have come from either $ST_{1,0}$ or $ST_{2,0}$, their next convex hulls are either $nextCH_{1,0}$ or $nextCH_{2,0}$; and when we remove the points in T from S and add their next convex hulls to S, we obtain the updated $S = \{S - T \cup nextCH_{1,0} \cup nextCH_{2,0}\}$ (24 data points), and re-compute the inner convex hull CH. Figure 5-12 shows the 0-depth contour, the inner convex hull, the initial S, and the additional points from the next convex hulls $nextCH_{1,0}$ and $nextCH_{2,0}$ after the update.

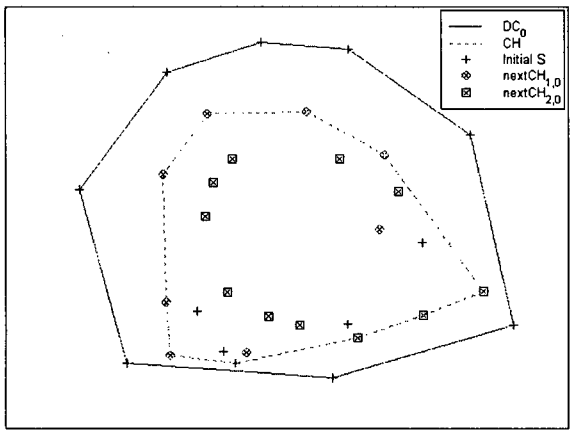


Figure 5-12 0-Depth Contour and 1st Inner Convex Hull of *S*

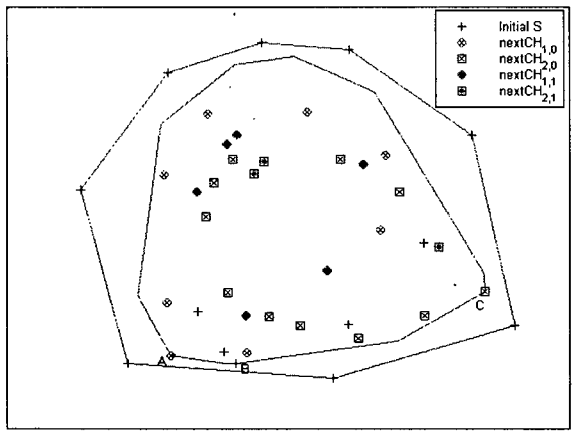


Figure 5-13 0-Depth and 1-Depth Contours of *S*

We then compute the 1-depth contour with T and CH. As data points A, B and C are on the 1-depth contour, they are removed from S and added to T; also added to S are $nextCH(A)$, $nextCH(B)$ and $nextCH(C)$. However, since point B is in the initial S, $nextCH(B)$ has, in fact, already been added to S. As point A is in $ST_{1,1}$, $nextCH(A)$, which is $nextCH_{1,1}$, has now to be added to S; similarly, point C is in $ST_{2,1}$, and $nextCH(C)$, which is $nextCH_{2,1}$, has also to be added to S. S now has 30 data points. Figure 5-13 shows all the data points newly added to S due to the removal of A, B and C. In general, the data points common to two different next convex hulls are added to S only once.

To compute the 2-depth contour, we use the updated T and the correspondingly updated CH. Figure 5-14 shows all the data points added at different stages and the depth contours computed by FDC-M3. On comparing these depth contours with those computed by FDC-M1 and FDC-M2, we are satisfied that all three space-constrained variants of FDC-basic produce matching results.

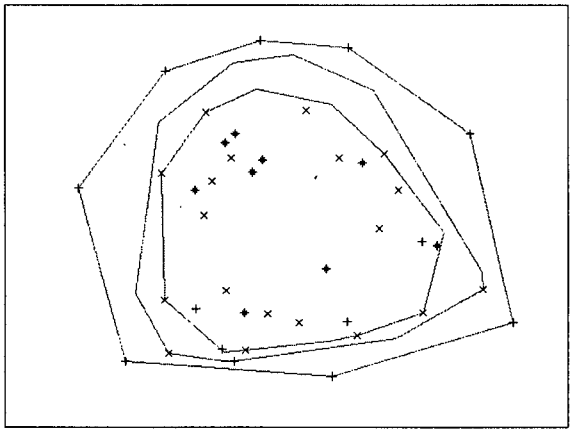


Figure 5-14 0-Depth to 2-Depth Contours of *S*

These variants of our basic algorithm are designed for use in situations with buffer-space constraints; and as space-constrained variants, all of them have an additional data-pruning process to remove the "useless" points (those with depth > k) so that S will fit the buffer limit. With FDC-M1 and FDC-M2, pruning takes place long before the final computation. In the case of FDC-M3, pruning is taken care of by Step 5.1.3. In the above example, S does not exceed the buffer limit, and no pruning is done. In fact, with FDC-M3, S does not normally require pruning; and if it does, pruning takes place only half way through the final computation, for example, after the computation of the first i^{th} depth contours. In that case, we reset S to the data points of depths from i to k by calling FDC-basic to compute the j+1st outer point set of S, where $j = k - i + 1$. We need not work up to the k+1st outer point set because S must consist of data points of depth $\geq i$ and only those data points of depth $\leq k$ are useful data points. These useful data points are located in the first j^{th} depth contours of S; therefore, the j+1st outer point set of S becomes the new S.

Chapter Six: 3-DIMENSIONAL FDC

Although FDC has been designed with a 2-dimensional scenario in mind, we have successfully generalized our study to cover 3-dimensional situations. In this chapter, we are going to show the reasoning, the working, and the performance of our 3-dimensional version of Algorithm FDC, FDC-3D.

We shall begin by first reviewing some of the terms we have defined for use in a 2-dimensional space. A data point is a 2-tuple (p_x, p_y) where p_x is the x-coordinate and p_y is the y-coordinate. $<p, q>$ is an e-divider of an n-point dataset D if at most e points are to the outside of the directed line passing through points p and q, where the outside of $<p, q>$ is its right side and the inside is its left side. The inside region of $<p, q>$ is the intersection of the convex hull of D (the smallest convex polygon that contains all data points in D) and the half-plane to the inside of $<p, q>$. The (expanded) e-intersected inside region is the convex polygon resulted from the intersection of all the (expanded) inside regions of a set of e-dividers.

3-Dimensional Depth Contours

In a 3-dimensional context, however, a data point becomes a triplet (p_x, p_y, p_z) where p_x is the x-coordinate, p_y is the y-coordinate and p_z is the z-coordinate. $[p, q, r]$ is an e-divider of an n-point dataset D if at most e data points are to the outside of the *plane* passing through points p, q and r. The outside of $[p, q, r]$ is the side out of which the normal vector of $[p, q, r]$ points; the inside is the opposite side. The inside region of $[p, q, r]$ is

the intersection of the convex hull of D (the smallest convex *polyhedron* that contains all data points in D) and the *half-space* to the inside of $[p, q, r]$. The (expanded) e-intersected inside region is the convex polyhedron resulted from the intersection of all the (expanded) inside regions of a set of the e-dividers.

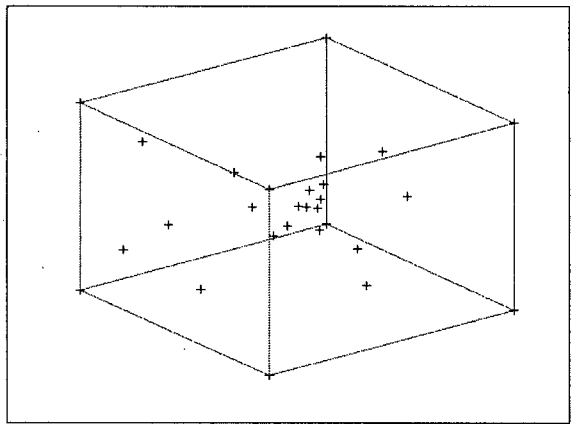


Figure 6-1 0-Depth Contour

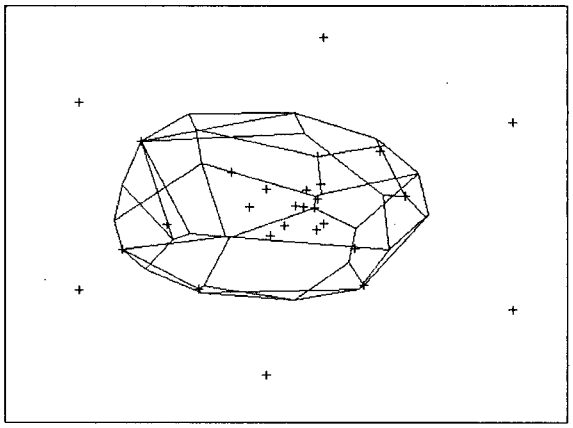


Figure 6-2 1-Depth Contour

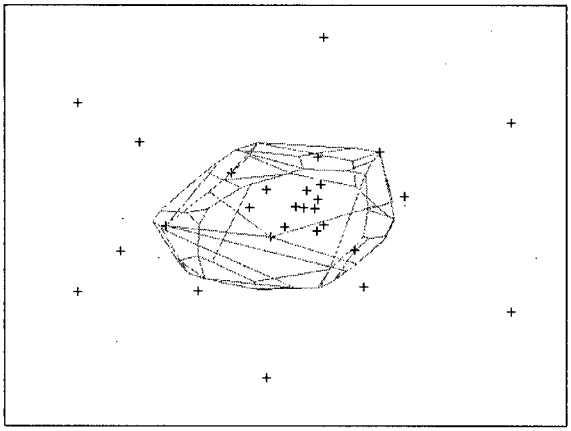


Figure 6-3 2-Depth Contour

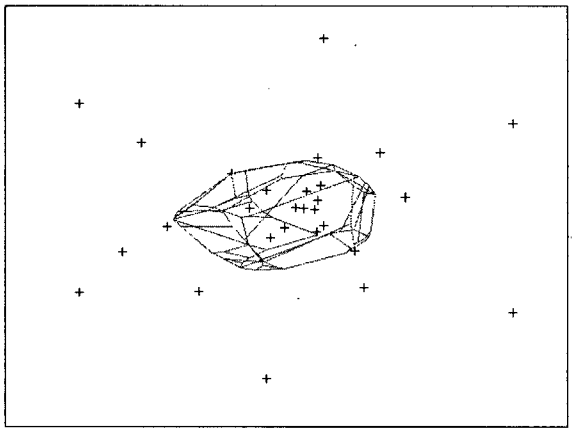


Figure 6-4 3-Depth Contour

3-dimensional depth contours are like their 2-dimensional counterparts in that they are also convex and nested. The above four figures (Figure 6-1 to Figure 6-4) show respectively the 0-depth contour, 1-depth contour, 2-depth contour, and 3-depth contour of a new sample dataset. This same sample dataset set is used throughout Chapter 6 to demonstrate the various phases of our 3-dimensional algorithm.

We have designed our 3-dimensional FDC algorithm – FDC-3D – following closely the logic of our 2-dimensional one. However, new steps are introduced when we expand the e-inside regions and intersect the half-spaces. Of course, the implementation of the two algorithms has to be different even if it is only because there are three attributes in a 3-dimensional dataset instead of two as in a 2-dimensional scenario.

Algorithm FDC(D, k)

Input: D = the dataset and k = an integer.

Output: Contours of depth from 0 to k.

1 Compute the 0-depth contour as in Steps 1 to 4 of FDC-basic;

2 For ($d = 1$; $d \leq k$;) { /* new peel */

2.1 Update D, T, and CH as in Steps 5.1 to 5.3 of FDC-basic;

2.2 If $|CH| == 4$, continue; /* degenerate case of having a tetrahedron as the convex hull */

2.3 For ($e = d$; $e < |CH|/2$;) { /* Otherwise: the general case */

2.3.1 For all pairs of points $p, q \in T$ ($p \neq q$) {

2.3.1.1 Find all points r in T (if any) so that $[p, q, r]$ is an e-divider of the points in T;

2.3.1.2 If every point in CH is contained in $IR([p, q, r])$,

$IR[p][q] = IR([p, q, r])$; $CT[p][q] = \varnothing$;

2.3.1.3 Else call Procedure Expand-3D(p, q, r, CH, T, e) to compute $IR[p][q]$ and $CT[p][q]$;

} /*end for */

2.3.2 Output the boundary of the region ($\cap_{p,q \in T} IR[p][q]$) as the d-depth contour;

2.3.3 If $d == k$, break; /*Done and Stop */

2.3.4 /* Otherwise: */ $d = d + 1$; $e = e + 1$;

2.3.5 If $\cup_{p,q \in T} CT[p][q] \neq \varnothing$ {

2.3.5.1 $T = T \cup (\cup_{p,q \in T} CT[p][q])$; $D = D - (\cup_{p,q \in T} CT[p][q])$;

2.3.5.2 CH = vertices of the convex hull of (the update) D;

} /* end if */

} /* end for */

} /* end for */

Expanding *e*-Inside Region

In our 2-dimensional algorithm, we expand the inside region of an *e*-divider $<p, q>$ with the points in the inner convex hull *CH* that are outside $IR(<p, q>)$ as follows. We sort the points outside $IR(<p, q>)$ in a counter-clockwise direction as $\{r_1, \ldots, r_n\}$. Then we find r_i that maximizes the angle between the segments $<r_i, p>$ and $<p, q>$ and r_j that maximizes the angle between the segments $<r_j, q>$ and $<q, p>$. The expanded series $<p, r_i, \ldots, r_j, q>$ is the list of line segments $\{<p, r_i>, \ldots, <r_j, q>\}$. The expanded inside region is the intersection of the inside regions of the line segments in the expanded series that are *e*-dividers. However, this particular Procedure Expand is applicable only to the 2-dimensional algorithm.

In a 3-dimensional algorithm, we cannot use this relatively straightforward procedure. First of all, we have no easy way to enumerate the vertices of the inner convex hull: we can no longer follow only one direction in the enumeration because a point now has both a longitude and two co-latitude co-ordinates. Figure 6-5a shows how θ alone can define a sequence in *CH* in a 2-dimensional case, while Figure 6-5b demonstrates that both θ and ϕ are needed in a 3-dimensional situation.

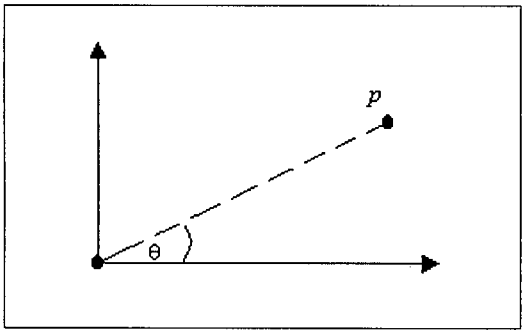


Figure 6-5a Point *p* in the 2-Dimensioanl Space

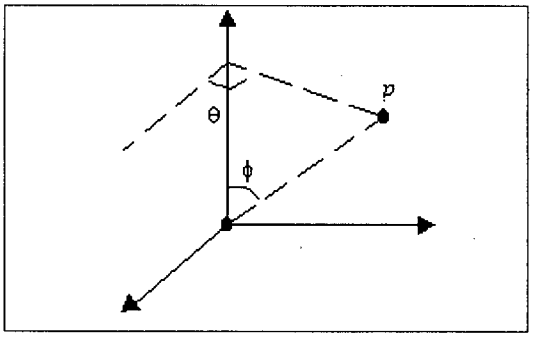


Figure 6-5b Point *p* in the 3-Dimensional Space

Then, as the points are not sorted in one direction in the 3-dimensional algorithm, we cannot obtain the sorted list $\{r_1, \ldots, r_n\}$ simply by comparing the angles of the points in *CH*. In the 2-dimensional algorithm, once we find r_i and r_j, we can easily locate all the points to be included in the expanded series, knowing that the points between r_i and r_j are also relevant. If we apply the same concept to a 3-dimensional space, we can name the data points in *CH* that are outside the *e*-divider $[p, q, r]$ $\{s_1, \ldots, s_n\}$. We can even find s_i that maximize the angle with the edge $<p, q>$, s_j that maximizes the angle with the edge $<q, r>$, and s_k that maximizes the angle with the edge $<r, p>$. However, as $\{ s_1, \ldots, s_n\}$ cannot be sorted in any single direction, we will still be unable to tell exactly where all relevant data points are "among" s_i, s_j and s_k. In other words, if we use the same concept as we do in our 2-dimensional algorithm, we will not be able to know what other data points and in what order these data points should be included in the expanded series.

Finally, if we follow the same concept, we cannot obtain a list of planes to expand the original *e*-divider. On summary, we are stuck with a set of unordered data points which does not enable us to derive the necessary list of planes for expanding the inside region.

Therefore, in the 3-dimensional algorithm, we conceive a new procedure to work out the correct planes. We institute Procedure Expand-3D by working on the convex hull of the data points $\{p, q, r, s_1, \ldots, s_n\}$. The expanded series $\{[p, q, s_i], [q, r, s_j], [r, p, s_k], \ldots\}$ is the list of facets of the convex hull excluding the plane $[p, q, r]$. The expanded inside region is the intersection of the inside regions of the planes that are *e*-dividers.

Procedure Expand-3D(*p, q, r, CH, T, e*)

Input: $[p, q, r]$ = an e-divider of points in T, and CH contains point(s) outside $IR([p, q, r])$.

Output: $IR[p][q]$ and $CT[p][q]$.

1 Find all the points $\{s_1, \ldots, s_n\}$ in CH that are outside $[p, q, r]$;

2 Compute the convex hull of $\{p, q, r, s_1, \ldots, s_n\}$;

3 Initialize $IR[p][q]$ = convex hull of T and $CT[p][q] = \varnothing$; /* convex hull of $T = DC_0$ */

4 For all facets F of the convex hull {

4.1 If F is not an e-divider of the points in $CH \cup \{s_1, \ldots, s_n\}$,

continue;

4.2 Else { /* expand the inside region */

4.2.1 $IR[p][q] = IR[p][q] \cap IR(F)$;

4.2.2 Add the vertices of F to $CT[p][q]$;

} /* end else */

} /* end for */

5 Return $IR[p][q]$ and $CT[p][q]$;

Let us now consider the examples below.

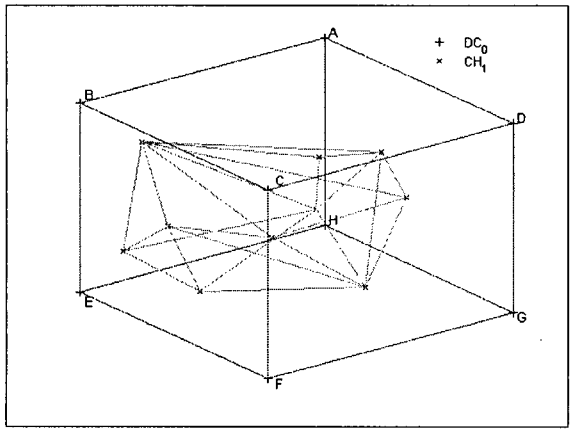


Figure 6-6 0-Depth Contour and 1st Inner Convex Hull

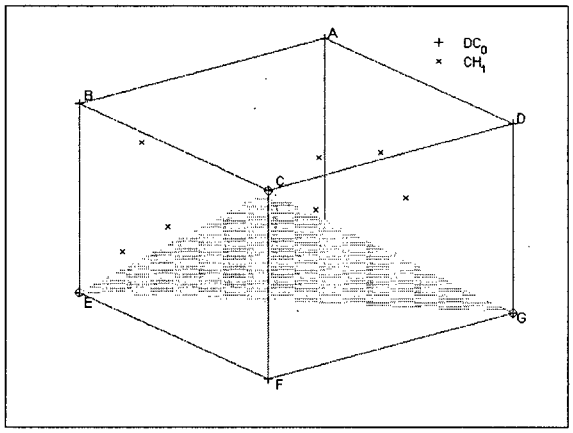


Figure 6-7 No points in CH_1 is to the outside of 1-Divider [C, E, G]

Figure 6-6 shows the 0-depth contour (DC_0) and the first inner convex hull (CH_1) of the sample dateset. To simplify the drawing, we will display only the vertices of the inner convex hull in the subsequent figures.

We want to find the 1-dividers of points $\{A, \ldots, H\}$. Figure 6-7 shows the 1-divider $[C, E, G]$. Since no points in CH_1 are outside $[C, E, G]$, no expansion is necessary.

In Figure 6-8, point I in CH_1 is outside the 1-divider $[B, D, F]$. The convex hull of $\{B, D, F, I\}$ gives the expanded series $\{[B, D, I], [D, F, I], [F, B, I]\}$. Because all of these three planes $[B, D, I]$, $[D, F, I]$ and $[F, B, I]$ are 1-dividers, the expanded inside region is the intersection of the inside regions $IR([B, D, I])$, $IR([D, F, I])$ and $IR([F, B, I])$.

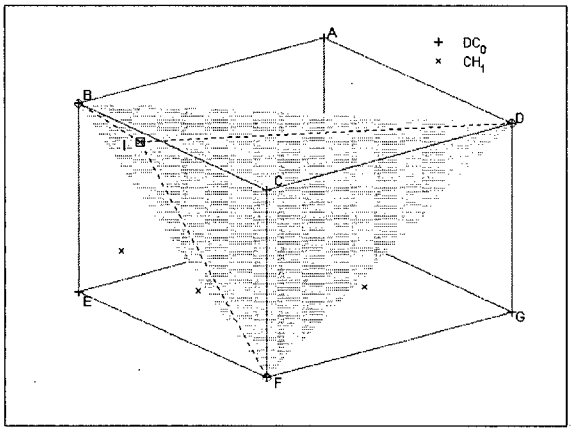


Figure 6-8 Expand $IR([B, D, F])$ with I

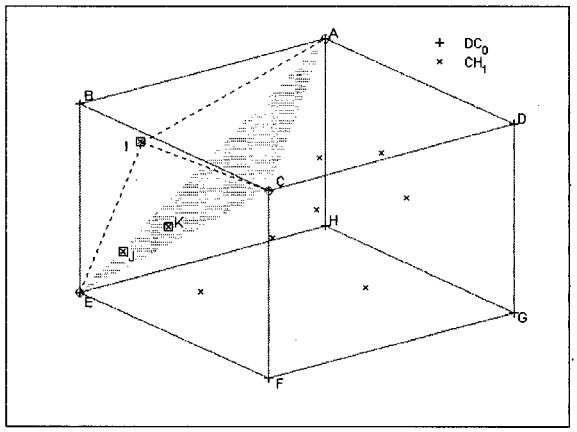


Figure 6-9 Expand $IR([A, C, E])$ with I, J, and K

In Figure 6-9, points I, J and K are outside the 1-divider $[A, C, E]$. As J and K are not contained in the convex hull of $\{A, C, E, I, J, K\}$, the expanded series has only planes $[A, C, I]$, $[C, E, I]$ and $[E, A, I]$, all of which are 1-divider. The expanded inside region is the intersection of their inside regions.

Intersecting Half-Spaces

Another major difference between the implementation of the 3-dimensional algorithm and that of the 2-dimensional algorithm lies in the intersection of half-spaces (which are half-planes in 2-dimension). We need the intersection of half-spaces when we expand an inside region and when we compute the intersected inside regions. In either case, we intersect some half-spaces with the convex hull of the dataset.

In 2-dimensional cases, we construct the intersection of n half-planes using a straightforward $O(n^2)$ computational geometric approach. (Preparata and Shamos has suggested a divide-and-conquer approach to compute the intersection of N half-planes in $O(N \log N)$ time. But, due to the unavailability of the source code, we use a simpler method in our implementation. We will improve on this part of the implementation in future.) Both the convex hull of the dataset and the half-planes are convex regions. Intersecting the convex hull with the half-plane to the inside of an e-divider simply involves slicing the convex hull with the e-divider and retaining the piece to the left of the e-divider, which can be done in linear time. The intersection of two convex regions is always convex. If we continue intersecting the intersection with the next half-plane, the final intersection can be found in quadratic time.

The corresponding operation in a 3-dimensional situation is more complicated. There is currently no computational geometry code available that can handle the intersection of 3-dimensional half-spaces in a straightforward and effective manner. As a result, we compute the intersection of half-spaces using both linear programming and duality transformation.

Duality transformation is a one-to-one mapping of a point to a plane, and vice versa. It maps objects from the *primal* space to the *dual* space. The image of an object under a duality transformation is called the dual of the object. Duality transformation is incidence preserving. A point p lies on a plane P if and only if the point P^* (the dual of P) lies on the plane p^* (the dual of p). If points p_1, p_2, p_3 and p_4 lie on plane P, then planes p_1^*, p_2^*, p_3^* and p_4^* intersects at point P^*. Several different transformations that preserve incidence are possible. A simple one is the following. The dual of point $p = (p_x, p_y, p_z)$ is the plane $p^* = (p_x x + p_y y + p_z - 1 = 0)$; the dual of plane $P = (Ax + By + Cz + D = 0)$ is the point $P^* = ({}^{-A}/_{D}, {}^{-B}/_{D}, {}^{-C}/_{D})$.

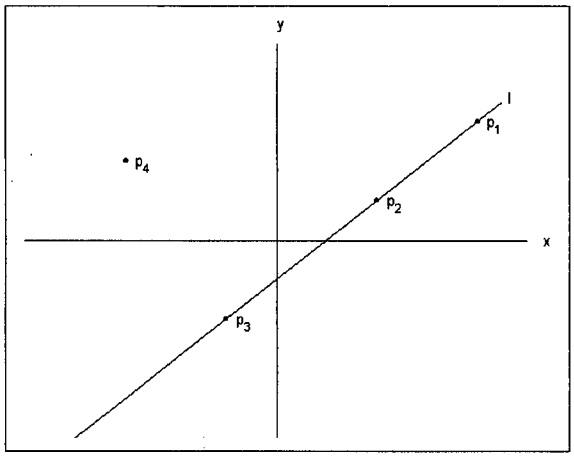


Figure 6-10a Primal Plane

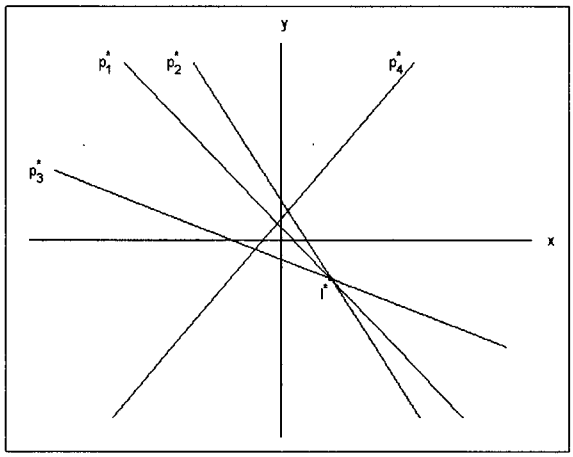


Figure 6-10b Dual Plane

Figure 6-10a and Figure 6-10b are examples of a 2-dimensional duality transformation. Point $p = (p_x, p_y)$ in the primal plane becomes line $p^* = (p_x x + p_y y - 1 = 0)$ in the dual plane; line $l = (Ax + By + C = 0)$ becomes point $l^* = ({}^{-A}/_{C}, {}^{-B}/_{C})$. The three points p_1, p_2 and p_3 lie on the line l in the primal plane; the lines p_1^*, p_2^* and p_3^* go through the point l^* in the dual plane.

The intersection of N half-spaces $H = \{h_1, \ldots, h_N\}$ can be computed via dualization if all N half-spaces contain the origin. We will not get into the details of the geometric theory behind the computation. In brief, the intersection of H is the dual of the convex hull of the point set $H^* = \{h_1^*, \ldots, h_N^*\}$, where h_i^* is the dual point of the half-space h_i.

To explain the dualization approach for the intersection of half-spaces while keeping the drawing simple, we illustrate the intersection of half-planes instead of half-spaces in Figure 6-11a. We want to find the intersection of the 12 half-planes in the primal plane. Each half-plane in the primal plane is dualized to a point in the dual plane (Figure 6-11b). The convex hull of the dual points is the dual of the intersection. Each line supporting the convex hull dualizes to a vertex of the intersection. For example, the line supporting edge 1 of the convex hull dualizes to vertex 1 of the intersection.

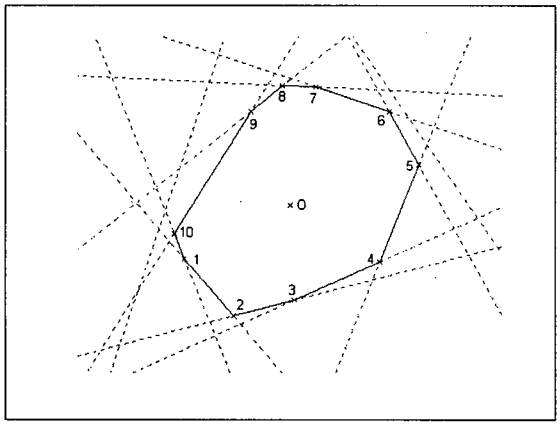


Figure 6-11a Primal Plane: Intersection of 12 half-planes (in dotted line).

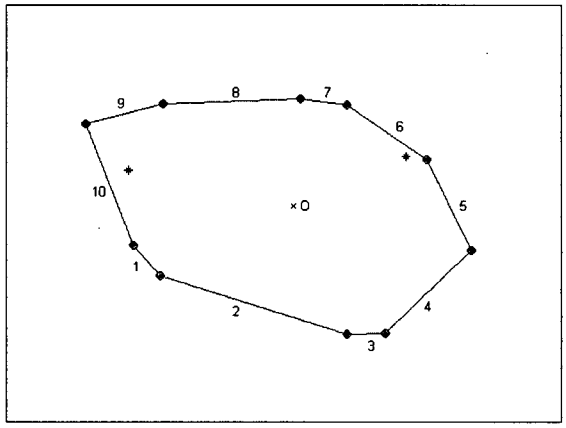


Figure 6-11b Dual Plane: Convex hull of the duals of the half-spaces.

While the dualization method solves our library software problem, its use hinges on two pre-requisites: (1) all the half-spaces must have a common intersection; and (2) the origin must be contained in the intersection. For the origin to be contained in the

intersection, we must first ascertain if the origin is contained in all the half-spaces. If that is not the case, we must find a common interior point for all the half-spaces before we can intersect the half-spaces. In our implementation, we use linear programming to find the common interior point. Each half-space is defined by an inequality $Ax + By + Cz + D \geq 0$. The objective of the linear program is to maximize w subject to the constraints $A_ix + B_iy + C_iz + D_iu_i - w_i \geq 0$ for $1 \leq i \leq N$. The optimal solution (x, y, z, u, w) with $u, w > 0$ gives an interior point $(x/w, y/w, z/w)$ of H.

Once we know an interior point, we can translate the half-spaces so that the origin coincides with the interior point. Then we can obtain the intersection using the dualization method. Finally, we translate the origin back to its original position. Of course, it is possible that there is no solution for the linear program. In that case, no points are inside the intersection, that is, the intersection is empty.

Complexity Analysis

In Chapter 3, we have broken down FDC-basic into four parts to analyze its complexity. For easy reference, we summarize below the complexities of the four parts of FDC-basic:

(n = the size of the dataset; k = the largest depth of contours desired; t = the size of the final outer point set T; c = the maximum cardinality of the first k CH)

1. Computing and maintaining convex hulls: $O(n \log n + t \log^2 n)$
2. Finding initial e-dividers: $O(kt^3)$
3. Expanding e-inside regions: $O(ckt^2)$
4. Intersecting half-planes: $O(kt \log t)$

We have analyzed the complexity of FDC-3D in the same way. Due to the existence of an additional dimension in FDC-3D, the complexity of each part is expected to be different when compared with the complexity of its 2-dimensioanl counterpart. However, we find that with standard routines, parts 1 and 4 maintain the same complexities in both dimensions. We will present the complexity analyses of parts 2 and 3 of our 3-dimensional algorithm as follows:

In Part 2, the complexity of finding the e-dividers for one point becomes $O(t^3)$ because for each point p, there are t^2 dividers $[p, q, r]$. We need to find the e-dividers for t points in each of the k iterations. Thus the total complexity of finding the initial e-dividers is $O(kt^4)$.

In Part 3, we need to compute the convex hull of the e-divider and the points in CH to get the expanded series. This step takes $O(c \log c)$ time. With at most $O(kt)$ expansions, the total complexity for expanding the e-dividers is $O(ckt \log c + ckt^2)$.

Thus, the total complexity of our 3-dimensioanl FDC is $O(n \log n + t \log^2 n + kt \log t + ckt \log c + ckt^2 + kt^4)$. Once again, Part 2 dominates the algorithm. We claims that it is possible to optimize Part 2 using the sorted-list technique as in the enhanced version of the 2-dimensional FDC.

Chapter Seven: PERFORMANCE AND EXPERIMENTAL RESULTS

We have described the algorithms for the different versions of FDC – the basic version (FDC-basic), the enhanced version (FDC-enhanced), the limited-buffer-space versions (FDC-M1, FDC-M2 and FDC-M3), and the 3-dimensional version (FDC-3D). We shall now review and compare their performances. We are going to compare the performance of FDC-basic with that of ISODEPTH; the performance of FDC-enhanced with that of FDC-basic; and the performance of FDC-M1, FDC-M2 and FDC-M3 among themselves.

All the above versions are implemented in C++; and because computational geometry plays an important role in the computation of depth contours, we have used the *LEDA* library in most of our computational geometry operations. These operations include the manipulation of geometric objects, the computation of convex hulls, and the intersection of half-planes. In the implementation of our 3-dimensional algorithm, we also use in addition to the LEDA library, the *LP_SOLVE* library when looking for interior points of the intersections of half-spaces, and the *QHULL* library when computing the intersections of half-spaces.

For the purpose of our study, we have translated Rousseeuw and Ruts' ISODEPTH program, which was written in Fortran, into C. We understand that ISODEPTH does not work with datasets that include duplicated or collinear points, and therefore we have been careful to remove such data points from our sample datasets when carrying out our comparative studies involving ISODEPTH. All our sample datasets are generated by using the *randn*(n, 2) command in MATLAB to generate n normally

distributed random number pairs; and all our smaller datasets are contained in the bigger ones.

Performance: FDC-basic vs. ISODEPTH

As discussed in Chapters 2 and 3, FDC-basic achieves a complexity of $O(n \log n + t \log^2 n + kt \log t + ckt^2 + kt^3)$ in computing the first k+1 depth contours (where n is the size of the dataset, t is the size of the outer point set, and c is the size of the inner convex hull), which is an improvement on ISODEPTH's complexity, $O(n^2 \log n + kn^2 + kn^{1.5} \log n)$. In support of this claim, we now stipulate the results of our various experiments on the computation time[4] and the relative performance of ISODEPTH and FDC-basic. First, in computing the first twenty-one depth contours of different datasets, we discovered that FDC-basic significantly outperforms ISODEPTH in computation time as n increases. Table 7-1 shows the actual time taken by the two algorithms in computing the first twenty-one depth contours of 100 to 5,000 data points. It also shows the relative performance of FDC-basic and ISODEPTH. When n is small, the two algorithms are competitive. For example, when there are fewer than 500 points to compute, FDC-basic merely needs one-third of the time taken by ISODEPTH. As n increases to 1,000, it runs 7 times faster than ISODEPTH; and when n reaches 3000, the disparity in favour of FDC-basic becomes apparent. Then, when n reaches 5000 with FDC-basic running more than 115 times faster than ISODEPTH, the improvement becomes indeed very significant. The change in gradient in the graph (Figure 7-1) illustrates the improvement

[4] The computation time (in seconds) is the average time for three runs performed to produce the first k+1 depth contours after the data points are loaded.

in computation time achieved by FDC-basic with respect to the increase in dataset size. Our comparison stops at n = 6500 because the version of ISODEPTH we use returns a segmentation fault when the dataset has more than 6500 data points.

n	ISODEPTH (sec)	FDC-basic (sec)	ISODEPTH / FDC - basic
100	1.212	1.208	1.003
250	3.413	1.956	1.745
500	9.599	2.731	3.515
750	17.997	3.245	5.546
1000	29.474	3.880	7.596
3000	214.859	4.547	47.253
5000	571.405	4.964	115.110
6500	954.588	6.180	154.464

Table 7-1 FDC-basic vs. ISODEPTH in computing the first 21 depth contours of different datasets

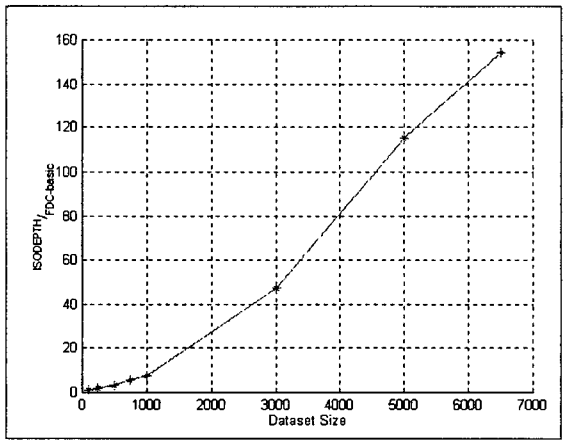


Figure 7-1 Compare FDC-basic and ISODEPTH wrt dataset size

Our studies reveal that the comparative advantage of FDC-basic over ISODEPTH in computation time and performance with respect to the number of depth contours to be computed is the greatest at the outset of the computation. This comparative advantage diminishes as the number of depth contours increases. Table 7-2 shows the computation time and relative performance of FDC-basic and ISODEPTH for the computation of 0- to k-depth contours of a 6500-point dataset, where $5 \leq k \leq 30$.

As k increases, FDC-basic continues to enjoy a substantial advantage over ISODEPTH, but the margin becomes closer. For example, in computing the first five depth contours, FDC-basic is 750 times faster than ISODEPTH. However, in computing thirty depth contours, FDC-basic is merely 79 times faster. This latter improvement in computation time achieved by FDC-basic is still considered significant. The decrease in

comparative advantage is predicted in our analysis, for we know that when k gets larger, the difference between n and t gets smaller. For example, in a 6500-point dataset, when k = 6, t = 61 (that is 0.94% of n); but, when k = 30, t = 228 (that is 3.51% of n). This observation matches our analysis.

k	ISODEPTH (sec)	FDC-basic (sec)	ISODEPTH / FDC - basic
5	359.096	0.477	753.559
10	550.069	1.730	317.959
15	746.278	3.678	202.903
20	954.588	6.180	154.464
25	1175.748	10.769	109.179
30	1400.608	17.645	79.377

Table 7-2 FDC-basic vs. ISODEPTH in computing different numbers of depth contours of 6500 data points

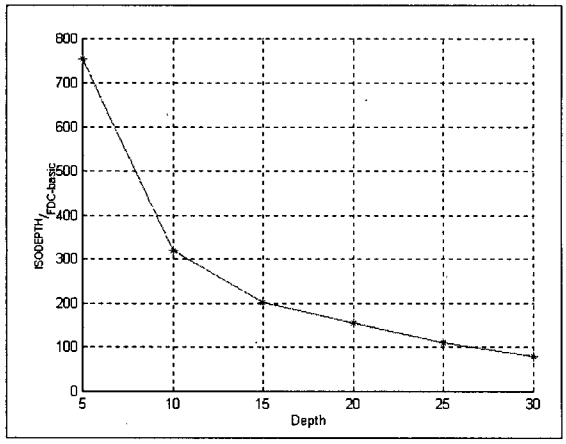


Figure 7-2 Compare FDC-basic and ISODEPTH wrt the depth

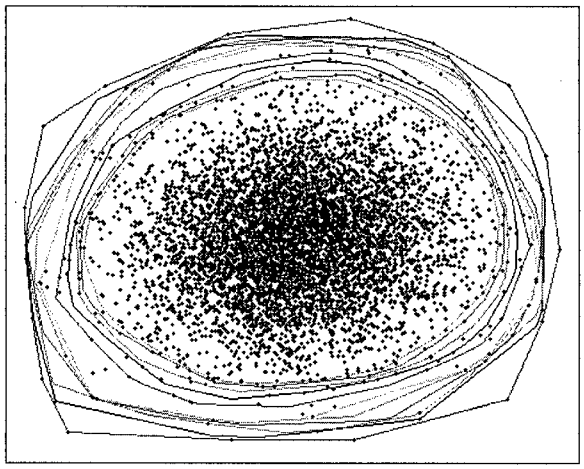


Figure 7-3 0- to 5-, 10-, 15-, 20-, 25-, and 30-Depth Contours of a 6500-point dataset

However, as mentioned earlier, the purpose of depth contour computation is to find the outliers of a large dataset, and these outliers are often embedded in the shallower depth contours, which are always computed first. A comparative advantage in computing

the shallow depth contours at the outset is, therefore, a significant improvement that contributes toward the overall performance of the algorithm.

Performance: FDC-enhanced vs. FDC-basic

In Chapter 4, we show that FDC-enhanced excels FDC-basic when k is large enough so that the reduction of the time from $O(kt^3)$ to $O(kt)$ by using sorted lists exceeds the extra time $O(t^3 + kt^2)$ required for their maintenance. Table 7-3 shows the computation time and the relative performance of FDC-enhanced and FDC-basic in computing the first k+1 depth contours of a 100,000-point dataset. When k = 25, FDC-enhanced takes slightly more time than FDC-basic; but as k approaches to 50, FDC-enhanced becomes 1.26 times faster than FDC-basic. When k = 100, FDC-enhanced is almost 3 times faster. The graph in Figure 7-4 shows the progressive improvement of FDC-enhanced over FDC-basic as the computation of depth contours moves progressively inwards.

k	FDC-basic (sec)	FDC-enhanced (sec)	$\frac{\text{FDC - basic}}{\text{FDC - enhanced}}$
25	35.119	42.001	0.84
50	184.715	146.423	1.26
75	677.518	347.367	1.95
100	1894.533	720.222	2.63
125	4320.423	1241.300	3.48
150	8592.299	1987.325	4.32

Table 7-3 FDC-enhanced vs. FDC-basic in computing different numbers of depth contours of 100,000 data points

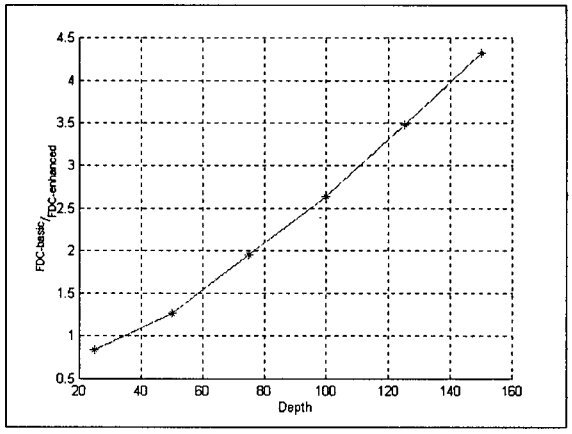


Figure 7-4 Compare FDC-enhanced and FDC-basic wrt the depth

The above comparative study focuses on the value of k while keeping the size of the dataset constant at 100,000 data points. To complete our study, we now consider the computation time and performance of FDC-enhanced and FDC-basic with respect to datasets of various sizes. We have conducted two computation runs – the first run covering the first twenty-one depth contours, and the second run covering the first one hundred and fifty-one depth contours. Both computation runs are based on the same eight datasets shown in Table 7-4 below. The datasets differ in size starting from a dataset containing 250 data points and culminating in a dataset containing 100,000 data points. (The first dataset of 250 data point has only one hundred and eighteen depth contours; therefore, as the only exception in this part of our study, the second computation run for this particular dataset stops at the 117-depth contour instead of the 150-depth contour.) Our findings are reflected in Table 7-4. Column 4 of Table 7-4 records the performance of FDC-basic relative to that of FDC-enhanced for computing a small number of depth contours; Column 7 of the same table records the relative performance of the two algorithms in computing a larger number of depth contours.

n	0- to 20-Depth Contours			0- to 150-Depth Contours		
	FDC-basic (sec)	FDC-enhanced (sec)	FDC - basic / FDC - enhanced	FDC-basic (sec)	FDC-enhanced (sec)	FDC - basic / FDC - enhanced
250	2.674	2.497	1.07	198.013	44.414	4.46
500	3.149	3.034	1.04	983.820	157.831	6.23
750	3.712	3.617	1.03	1380.745	242.220	5.70
1000	3.931	3.780	1.04	1644.037	302.097	5.44
3000	5.385	6.072	0.89	2887.457	579.034	4.99
5000	5.513	6.117	0.90	3611.877	766.316	4.71
10000	7.423	9.456	0.79	4732.456	1017.312	4.65
100000	23.434	28.872	0.81	8592.299	1987.325	4.32

Table 7-4 FDC-basic vs. FDC-enhanced in computing the first 21 and the first 151 depth contours of different datasets

Reading across the table, we find that all the results in Column 7 compare favourably with those in Column 4. This confirms that, irrespective of the data size, FDC-enhanced is the more powerful algorithm when k is large as Table 7-3 already demonstrates. However, when k is small, the relative advantage of FDC-enhanced over FDC-basic is less obvious, and even turns negative when dataset gets bigger as in the last four cases Column 4 of Table 7-4.

Performance: FDC-M1, FDC-M2 and FDC-M3

FDC-M1, FDC-M2 and FDC-M3 are designed for situations in which the buffer space is insufficient to accommodate the computation of the entire dataset. FDC-M1 repeats a simple divide-and-conquer technique to reduce the original dataset to a final dataset sufficiently small for the buffer to perform the final computation. FDC-M2 adopts also a periodically adjusted controlling convex hull to exclude more unqualified data points from the final dataset. FDC-M3 starts the final computation with a starter dataset that contains only points of 0-depth, and expands the dataset by relatively small increments before the next computation.

Our comparative studies show the total computation time[5] and the relative performance of FDC-M1, FDC-M2 and FDC-M3 in computing the first thirty-one depth contours of six datasets of increasing sizes, using two buffer-space constraints, namely, 250 data points (Table 7-5) and 500 data points (Table 7-6). The second column (t) in

[5] Total computation time, in seconds, includes the time to load the data points, to divide the data points into smaller subsets, to compute the outer point sets and related intermediate steps, and to compute the final depth contours.

each table records the number of data points outside the 30-depth contour of each dataset in the first column (n), and constitutes the minimum buffer space requirement for all the three methods at any particular level. It is important to note that the t values for the bigger datasets are fairly close to the 250 buffer limit.

n	t	FDC-M1 (sec)	FDC-M2 (sec)	FDC-M3 (sec)	FDC - M1 / FDC - M2	FDC - M1 / FDC - M3	FDC - M2 / FDC - M3
500	157	21.743	18.981	16.489	1.15	1.32	1.15
750	174	36.927	25.229	22.007	1.46	1.68	1.15
1000	184	52.547	38.918	27.495	1.35	1.91	1.42
3000	214	251.433	91.760	53.976	2.74	4.66	1.70
5000	231	557.971	499.862	85.411	1.12	6.53	5.85
10000	241	2346.565	5045.281	197.922	0.47	11.86	25.49

Table 7-5 Total computation time and relative performance of FDC-M1, FDC-M2 and FDC-M3 in computing the first 31 depth contours of different datasets with buffer size = 250

n	t	FDC-M1 (sec)	FDC-M2 (sec)	FDC-M3 (sec)	FDC - M1 / FDC - M2	FDC - M1 / FDC - M3	FDC - M2 / FDC - M3
750	174	19.889	17.434	20.204	1.14	0.98	0.86
1000	184	23.817	20.925	24.436	1.14	0.97	0.86
3000	214	61.947	56.531	49.266	1.10	1.26	1.15
5000	231	104.125	87.218	77.346	1.19	1.35	1.13
10000	241	207.852	166.320	153.465	1.25	1.35	1.08

Table 7-6 Total computation time and relative performance of FDC-M1, FDC-M2 and FDC-M3 in computing the first 31 depth contours of different datasets with buffer size = 500

The results of our experiments confirm three distinctive characteristics of our solutions to the buffer-space problem. First, in general, both FDC-M2 and FDC-M3 outperform FDC-M1, with FDC-M3 taking the lead. Second, FDC-M2 is the least effective when the buffer size is close to t and very small when compared with n. Third, FDC-M3 is decidedly the best solution for larger datasets under buffer-space constraints.

The better performance of FDC-M2 and FDC-M3 is predictable because we know that FDC-M2 and FDC-M3 are more effective in reducing the number of data points than FDC-M1. Fewer data points for processing means less buffer-space requirement and shorter processing time.

When the dataset is large and the buffer space is severely constrained, FDC-M1 and FDC-M2 crunch more data points before the final computation because of the way they go about screening the data points. The last two experiments recorded in Table 7-5 show that both FDC-M1 and FDC-M2 drop in performance when the buffer size is close to t and very small when compared with n. We have looked into these two experiments and observed that FDC-M1 and FDC-M2 repeat their respective intermediate steps a great number of times. For example, in the case when n = 10,000 and t = 241, FDC-M1 carries out 53 division-and-elimination runs while FDC-M2 conducts a total of 334 runs of batch merging. Repeating these intermediate steps to reduce the number of data points has caused the two algorithms to slow down before the final computation begins.

Then, we compared the time taken by the division-and-elimination process and the time taken to complete a single run of batch merging. We found that while a single run of batch merging takes a shorter time, the total time required for the completion of 334 runs of batch merging has actually dragged down the performance of FDC-M2. We further noticed that when S has accumulated 200 data points after the first four runs of batch merging, FDC-M2 goes into an "equilibrium" stage. Thereafter, when S grows to 230 data points, FDC-M2 completes its eighth run of batch merging; when S grows to 240 data points, FDC-M2 completes the seventeenth run. In other words, FDC-M2 still has to do more than 300 runs of batch merging, each updating fewer than 10 data points.

This is the reason why FDC-M2 proves to be the least efficient of the three algorithms when the buffer size is close to t and very small when compared with n.

By contrast, the performance of FDC-M3 is the most stable among the three algorithms. It is the one least affected by an increase in the size of the dataset. Indeed, FDC-M3 performs best among the three when the dataset is very large – a scenario we are most likely to encounter in outlier detection projects. In Table 7-5, the last two records show that FDC-M3 is 11.86 times faster than FDC-M1, and 25.49 times faster than FDC-M2 when n = 10000, t = 241 and the buffer limit is 250. In the corresponding records in Table 7-6, when n = 10000, t = 241 and the buffer limit is 500, FDC-M3 excels FDC-M1 by 1.35 times; and FDC-M2 by 1.08 times.

When the buffer space is increased from 250 to 500, all three algorithms work faster. However, while the increase in efficiency experienced by FDC-M1 and FDC-M2 ranges from over 30% (when the dataset is small) to over 90% (when the dataset is large), FDC-M3 maintains a steady improvement of under 25%. Indeed, FDC-M3 needs more time than FDC-M1 and FDC-M2 when the dataset is small and the buffer size is relaxed as in the first two experiments in Table 7-6, in which the buffer can accommodate 500 data points, and there are only 750 points and 1,000 points respectively in the given datasets. The relaxation in buffer space enables FDC-M1 and FDC-M2 to cut down on the repetition of their intermediate steps drastically, but FDC-M3 must still work through its time-consuming intermediate step. This insistence writes off some of the gains in efficiency by FDC-M3 in its final computation. However, as these three variants are designed to work with large datasets under buffer-space constraints, FDC-M3 is still the best solution.

Performance: FDC-3D

We admit that our 3-dimensional algorithm is still at a rudimentary stage of its development. With the addition of one more dimension to the algorithm, the complexity of FDC-3D becomes $O(n \log n + t \log^2 n + kt \log t + ckt \log c + ckt^2 + kt^4)$. Geometrical computations are more costly in 3-dimensional situations. Thus, we expect a drop in the performance of FDC-3D as the data set gets larger. Moreover, our FDC-3D is a generalization of only the basic version of our Algorithm FDC. There is still room for improvement and further studies. In this study, we just want to show that FDC can be generalized to cover 3-dimensional situations.

Table 7-7 shows the computation time of FDC-3D in computing the first 21 and the first 31 depth contours of eight datasets of various sizes ranging from 100 data points to 10,000 data points each.

n	$k = 20$	(Time in Sec.)	$k = 30$	(Time in Sec.)
100		64.425		162.442
250		293.530		970.763
500		713.617		2648.016
750		855.178		3506.481
1000		1283.118		5335.907
3000		3461.361		15534.112
5000		4379.839		19546.748
10000		6352.490		29352.494

Table 7-7 Computation time of FDC-3D in computing the first 21 and 31 depth contours of different datasets.

Chapter Eight: CONCLUSION

In data mining, there are now three general methods for outlier detection: a statistical approach, a distance-based approach, and a depth-based approach. Unfortunately, the statistical approach works only with single-attribute databases, and requires *apriori* knowledge of the distribution of the data; and the distance-based approach requires the existence of a well-defined metric distance function. A depth-based approach is not limited by either of these restrictions. By looking at the depths of data objects within a dataset and constructing a depth contour to surround all the data objects of the same depth, a depth-contour algorithm can indicate precisely where the outliers are within a dataset. Different depth contours reveal data objects lying within different layers of the dataset, each layer closer to the centre of the dataset being more robust with respect to the outliers. Rousseeuw and Ruts' depth contour algorithm, ISODEPTH, is an important pioneer in furthering this approach.

However, we find that ISODEPTH leaves something to be desired both in its speed and its universal application. We have, therefore, designed a Fast Depth Contour Algorithm – FDC. Aimed at efficiency and effectiveness, Algorithm FDC has two distinguishing features: it need not process all n data points to compute the required depth contours, and it relaxes all data positioning criteria. We have invented a basic version of FDC (FDC-basic), and created an enhanced version (FDC-enhanced) as well as a 3-dimensional version (FDC-3D). Moreover, we developed three other variants of our Algorithm FDC-basic (FDC-M1, FDC-M2, and FDC-M3) to cover the eventuality of

insufficient computer buffer space. All three versions and all three variants of our Algorithm FDC are presented in this thesis.

Algorithm FDC-basic is our original design to solve the problem posted by ISODEPTH. We recognize that the outliers of a dataset comprise only a small percentage of the dataset in the shallow depth contours, and that only a few depth contours are needed to identify the outliers. The bulk of the dataset not contributing to the computation of the required depth contours, FDC-basic selects only the boundary points and the exterior points for computing the k-depth contour. By definition, the k-depth contour separates the boundary points (data points of depth = k) and the exterior points (data points of depth < k) from the interior points (data points of depth > k).

For the computation of each depth contour, we make FDC-basic divide the dataset into two disjoint subsets: the outer point set T, and the inner point set D. T consists of the data points outside the depth contour, and D contains the data points inside the depth contour. We further make FDC-basic use only the data points in T and the data points in the convex hull of D, which we call the inner convex hull CH, when computing the current depth contour.

As the convex hull of the dataset is the 0-depth contour, the vertices of the convex hull become the first T for computing the 1-depth contour. The inner convex hull CH is the convex hull of the remaining data points. To compute the 1-depth contour, FDC-basic finds all the 1-dividers of T. For every 1-divider found, FDC-basic checks if the inside region of the 1-divider contains all the points in CH. If so, the 1-divider is not only

a 1-divider of T, but also a 1-divider of the dataset. If some points in CH are outside the inside region, the inside region is expanded.

To expand the inside region of an e-divider $<p, q>$, FDC-basic finds all the data points $\{r_1, \ldots, r_n\}$ in CH that are outside the insider region. Then, among those data points, it finds a point r_i that maximizes the angle $\angle r_i pq$ and a point r_j that maximizes the angle $\angle r_j qp$ (where r_i and r_j can be the same). The expanded series is a chain of line segments, where the first point of the first segment is p, the last point of the last segment is q, and the segments in between are the edges of CH starting from point r_i to point r_j. The expanded inside region is the intersection of the inside regions of the segments that are e-dividers.

After finding the (expanded) inside regions of all the 1-divider, FDC-basic computes the intersection of these (expanded) inside regions and returns the boundary of the intersection as the 1-depth contour.

If none of the inside regions are expanded, FDC-basic proceeds to compute the 2-depth contour with the existing T and CH. But, if some inside regions are expanded, FDC-basic updates T and CH. All the points in CH that are outside some inside regions are removed from the inner point set D and added to T. Whenever D is changed, CH is updated as well. After updating CH, FDC-basic proceeds to compute the next depth contour.

Thus, we have an algorithm that computes the depth contours of a dataset effectively, using only a subset of the dataset. The complexity of our Algorithm FDC-basic is $O(n \log n + t \log^2 n + kt^3 + ckt^2 + kt \log t)$, where n is the size of the dataset, t is

the maximum number of points in T, c is the maximum cardinality of CH, and k is the number of desired depth contours. When n is large, FDC-basic significantly outperforms ISODEPTH. When computing the first 21 depth contours of a 6500-point dataset, FDC-basic is 150 times faster than ISODEPTH.

Despite this breakthrough by FDC-basic, we believe that our Algorithm FDC can be developed further. We note that the $O(kt^3)$ complexity of FDC-basic relates to the finding of the initial e-dividers. To find an e-divider for point p, FDC-basic goes through each point q in T and checks if $<p, q>$ is an e-divider. Then, when FDC-basic needs to find the e+1-divider, it goes through all the points in T again just because some points have been freshly added, and repeats all the steps from scratch.

To avoid this repetition, we have created FDC-enhanced. We made FDC-enhanced remember the e-values of all the dividers, and put them into sorted lists. If no new points are added to T, we can now read the set of e+1-dividers from the sorted lists directly. Even if some points have been added, FDC-enhanced simply updates the sorted lists by providing new e-values to the affected dividers and associates each data point in T with a sorted list. The sorted list of a point p has at most k+1 elements, where the i^{th} element of the sorted list of p consists of all the points q in T, where $<p, q>$ is an i-divider. The initial e-dividers can now be read from the sorted lists in $O(kt)$ time.

However, FDC-enhanced also introduces two new procedures: the maintenance and the creation of the sorted lists. When new data points are added to the outer point set, FDC-enhanced has to maintain the existing sorted lists and create new sorted lists for the new data points. The maintenance procedure involves the insertion of the new points into

the existing sorted lists and the re-arrangement of the points in the sorted lists so as to maintain the correct position of all the elements. As the position of a data point p in the sorted list for a new data point q is exactly the opposite of the position of q in the sorted list for p, the creation of the new sorted lists is a by-product. The maintenance procedure and the creation procedure together require $O(t^3 + kt^2)$ time.

Since FDC-basic requires $O(kt^3)$ time to find the initial e-dividers and FDC-enhanced requires a total of $O(t^3 + kt^2 + kt)$ time to find the initial e-dividers and to maintain the sorted lists, the comparison of the two algorithms relies on how many depth contours are requested. When only a few depth contours are requested, that is when k is small, FDC-enhanced may not excel FDC-basic. However, when more depth contours are requested, FDC-enhanced becomes more efficient than FDC-basic. For a 100,000-point sample dataset, FDC-enhance takes slightly more time than FDC-basic to compute the first 21 depth contours; but it takes only a quarter of the time required by FDC-basic to compute the first 151 depth contours.

As both FDC-basic and FDC-enhanced are main-memory algorithms, we have been wary of insufficient computer buffer space. What should we do if we want to compute the first k+1 depth contours of a dataset of size n when the buffer space can accommodate only $m < n$ data points? To provide for such a case, we have come up with three variants of FDC-basic.

FDC-M1 is the basic method that uses a simple divider-and-conquer technique. It first divides the dataset into smaller subsets, then computes the k+1st outer point sets of all the subsets, and finally computes the depth contours of the union of all the outer point

sets. After computing the outer point set, it discards those data points of depth > k in each subset because a point of depth > k in a subset must have depth > k in the original dataset; and, therefore, is not needed in the computation of the first k+1 depth contours.

FDC-M2 uses the batch-merging method. It is similar to FDC-M1, except that it also organizes the data points in the outer point sets by their depth and performs an extra filtering procedure in the batch-merging phase to reduce the number of data points to be considered in the final computation. It picks a convenient subset and uses its k+1st outer point set as the initial S (the dataset for the final computation) and its k+1st inner convex hull as *CCH* (the controlling convex hull). The data points in the outer point sets of the other subsets are categorized by their respective depths within the subsets, and added to S only if they are outside the controlling convex hull. We have developed this procedure because we know that if a data point is inside the controlling convex hull, it will have depth > k and will be too deep for use in finding the first k+1 depth contour. If S already reaches the buffer space limit before taking up all the qualified data points, FDC-M2 will re-initiate S to the k+1st outer point set of S and *CCH* to the k+1st inner convex hull of S. Once S contains all the qualified data points, FDC-M2 will proceed to compute the depth contours of S.

Our FDC-M3 proceeds in a manner different from that of FDC-M1 or FDC-M2. When it starts computing the outermost depth control, it uses but a minimal S derived from a full merging of the 0-depth points in all the subsets. Then, as the computation of depth contours proceeds inward, contour by contour, it reinitiates S by merging all the next convex hulls of points freshly moved to the outer point set T and adding them to the points that have not been touched. In this manner, FDC-M3 saves much buffer space by

minimizing the number of data points to be considered in each round of the final computation. All versions of FDC share the standard procedure of computing the 0-depth contour as the convex hull of S. However, before they begin the computation, FDC-M1 and FDC-M2 fill S with all the necessary data points. Then, they simply move the points on the 0-depth contour from S to the outer point set T, and set the inner convex hull to the convex hull of the remaining points in S. Not having filled S in a like manner when computing the 0-depth contour, FDC-M3 must add a further minimum number of necessary data points to S before computing its inner convex hull and returning to the other regular FDC-basic procedures. For the addition of further points, FDC-M3 makes use of the fact that when a point is moved from S to T, its next convex hull is the set of data points immediately behind it. Indeed, FDC-M3 has already associated a next convex hull with each data point in the outer point set of each subset into which FDC-M3 divides up the original dataset. Some of these next convex hulls are now referenced to one of the points moved to T, and become eligible for addition to S. For each data point moved to T, FDC-M3 adds its next convex hull to S if that next convex hull has not already been included once. Avoidance of duplication is necessary because data points of the same depth have the same next convex hull.

All three methods have achieved some savings of buffer space by reducing the size of the original dataset before commencing the final computation. FDC-M1 is the naïve and slowest method among the three. FDC-M3 is slightly better than FDC-M2 in general, and greatly outperforms FDC-M2 when the buffer size is severely constrained and the dataset is large. Although FDC-M2 does a better job when the dataset is small and the buffer space is less constrained because fewer repetitions of the intermediate step

are then necessary, as an algorithm to overcome buffer-space constraints, FDC-M3 is the more versatile.

Finally, our product, FDC-3D, is a generalization of Algorithm FDC-basic to accommodate 3-dimensional situations. For the present study, we have stopped the generalization at the 3-dimensional level as computational geometry in even higher dimensions is quite costly and the resultant depth contours may appear hard to visualize. We have also noted that even at the 3-dimensional level, the complexity is already $O(n \log n + t \log^2 n + kt \log t + ckt \log c + ckt^2 + kt^4)$. As a preliminary study, this complexity may still be acceptable; but we intend to improve the performance of the FDC-3D algorithm in the future by using sorted lists as we have done in FDC-enhanced.

All in all, our FDC algorithms have pointed out new directions for working at depth-contour computation; and we wish to devote our future study to improving further their efficiency and effectiveness. Therefore, besides the proposed enhancement in FDC-3D, we have planned to add the following new features to our 2-dimentional FDC algorithms.

At present, users of our FDC algorithms must supply the maximum desired depth k in order to computes the first k+1 depth contours. In other words, for the purpose of outlier detection, the users must first make a legitimate guess of the appropriate k. This being a less familiar concept than the universally understood notation of percentages, most users will find it easier to work with FDC if they need simply to specify a desired percentage of data points to initiate the algorithm. Our next enhancement, therefore, is

to enable FDC to work on the generation of depth contours until the specified percentage of data points are included in the depth contours.

Moreover, in the current FDC, the depth contours are computed one by one consecutively. As the set of data points outside the i-depth contour does not usually differ a great deal from that outside the i+1-depth contour, a strict adherence to one-by-one consecutive computation may not always be necessary. We will work on an enhancement aimed at allowing the users to specify the incremental value they consider appropriate in the circumstances so that the depth contours will then *jump* instead of stepping up.

Finally, in our study of FDC-M1, FDC-M2 and FDC-M3, we have adopted just the simplest method in dividing up the original data set into subsets. However, other division methods are also available. We believe that some of these other methods are worth testing. Our future study in respect of FDC-M1, FDC-M2 and FDC-M3 will focus on the search for the most efficient and effective means of dividing up the data points so as to facilitate the intermediate steps and enhance the overall efficiency of the algorithms.

Our studies up to this point have produced six new algorithms for computing depth contours. All six algorithms – FDC-basic, FDC-enhanced, FDC-M1, FDC-M2, FDC-M3 and FDC-3D – have been implemented with sample data that we believe to be unbiased. The results of our lab-tests are reported and commented on in this thesis. In general, they have demonstrated new methods of improving the way in which we tackle the problem of finding extraordinary information in large databases. As our information age progresses into the twenty-first century, people will see more and more uses of

databases in all aspects of their activities. Although data management techniques have improved in leaps and bounds following the rapid development of hardware and software, databases are increasing and expanding at such alarming rates at all levels that data-mining techniques can hardly catch up. Such phenomena point directly to a corresponding increase in the importance of studies concerning outliers and their discovery. It is our fervent hope that the modest improvement achieved by our Algorithm FDC in the speed of computing depth contours will help propagate more uses of databases in the next millennium, and kindle a greater interest in academic studies in this specialized area.

BIBLIOGRAPHY

1. J. Han, Y. Cai, and N. Cercone. Knowledge Discovery in Databases: An Attribute-Oriented Approach. In *Proc. of the 18th VLDB Conference*, pages 547-559, Vancouver, British Columbia, Canada, 1992.

2. E. M. Knorr and R. T. Ng. Find Aggregate Proximity Relationships and Commonalties in Spatial Data Mining. *IEEE Transactions on Knowledge and Data Engineering*, 8(6): 884-897, 1996.

3. E. M. Knorr and R. T. Ng. Extraction of Spatial Proximity Patterns by Concept Generalization. In *Proc. of the 2nd KDD Conference (KDD-96)*, pages 347-350, Portland, Oregon, USA, 1996.

4. R. Agrawal and A. Swami. Fast Algorithms for Mining Association Rules. In *Proc. of the 20th VLDB Conference*, pages 487-499, Santiago, Chile, 1994.

5. R. Agrawal and R. Srikant. Mining Generalized Association Rules. In *Proc. of the 21st VLDB Conference*, pages 407-419, Zurich, Switzerland, 1995.

6. J. Han and Y. Fu. Discovery of Multiple-Level Association Rules from Large Databases. In *Proc. of the 21st VLDB Conference*, pages 420-431, Zurich, Switzerland, 1995.

7. R. Agrawal and R. Srikant. Mining Quantitative Association Rules in Large Relational Tables. In *Proc. of the 1996 ACM SIGMOD International Conference on Management of Data*, pages 1-12, Montreal, Quebec, Canada, 1996.

8. R. Agrawal and R. Srikant. Mining Sequential Patterns. In *Proc. of the 11th International Conference on Data Engineering*, pages 3-14, Taipei, Taiwan, 1995.

9. H. Mannila, H. Toivonen, and A. I. Verkamo. Discovering Frequent Episodes in Sequences. In *Proc. of the 1st KDD Conference (KDD-95)*, pages 210-215, Montreal, Quebec, Canada, 1995.

10. R. Agrawal, S. P. Ghosh, T. Imielinski, B. R. Iyer, and A. N. Swami. An Interval Classifier for Database Mining Applications. In *Proc. of the 18th VLDB Conference*, pages 560-573, Vancouver, British Columbia, Canada, 1992.

11. R. T. Ng and J. Han. Efficient and Effective Clustering Methods for Spatial Data Mining. In *Proc. of the 20th VLDB Conference*, pages 144-155, Santiago, Chile, 1994.

12. M. Ester, H. -P. Kriegel, J. Sander, and X. Xu. A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise. In *Proc. of the 2nd KDD Conference (KDD-96)*, pages 226-231, Portland, Oregon, USA, 1996.

13. R. Agrawal, K.-I. Lin, H. S. Sawhney, and K. Shim. Fast Similarity Search in the Presence of Noise, Scaling, and Translation in Time-Series Databases. In *Proc. of the 21st VLDB Conference*, pages 490-501, Zurich, Switzerland, 1995.

14. D. Angluin and P. Laird. Learning From Noisy Examples. *Machine Learning*, 2(4): 373-370, 1988.

15. E. M. Knorr. On Digital Money and Card Technologies. Technical Report 97-02, University of British Columbia, 1997.

16. D. Hawkins. *Identification of Outliers*. Chapman and Hall, London, 1980.

17. V. Barnett and T. Lewis. *Outliers in Statistical Data*. John Wiley & Sons, 1994.

18. D. C. Hoaglin, F. Mosteller, J. W. Tukey. *Understanding Robust and Exploratory Data Analysis*. Wiley, New York, 1983.

19. Arning, R. Agrawal, P. Raghavan. A Linear Method for Deviation Detection in Large Databases. In *Proc. of the 2nd KDD Conference (KDD-96)*, pages 164-169, Portland, Oregon, USA, 1996.

20. E. M. Knorr and R. T. Ng. Algorithms for Mining Distance-Based Outliers in Large Datasets. In *Proc. of the 24th VLDB Conference*, pages 392-403, New York City, New York, USA, 1998.

21. F. P. Preparata and M. I. Shamos. *Computation Geometry: an Introduction*. Springer-Verlag, New York, 1985.

22. J. W. Tukey. Mathematics and the Picture of Data. In *Proc. International Congress on Mathematics*, pages 523-531, 1975.

23. J. W. Tukey. *Exploratory Data Analysis*. Addison-Wesley, Don Mills, Ontario, 1977.

24. P. J. Rousseeuw and I. Ruts. Computing Depth Contours of Bivariate Point Clouds. *Computational Statistics & Data Analysis*, 23: 153-168, 1996.

25. T. Johnson, I. Kwok, and R. T. Ng. Fast Computation of 2-Dimensional Depth Contours. In *Proc. of the 4th KDD Conference (KDD-98)*, pages 224-228, New York City, New York, USA, 1998