Eliminating Duplicates Under Interval and Fuzzy Uncertainty: An Asymptotically Optimal Algorithm and Its Geospatial Applications

Roberto Torres[1,2], G. Randy Keller[2],
Vladik Kreinovich[1,2], and Luc Longpré[1,2]

[1]Department of Computer Science and
[2]Pan-American Center for Earth and
Environmental Studies (PACES)
University of Texas at El Paso
El Paso, TX 79968, USA
contact email vladik@cs.utep.edu

Abstract

Geospatial databases generally consist of measurements related to points (or pixels in the case of raster data), lines, and polygons. In recent years, the size and complexity of these databases have increased significantly and they often contain duplicate records, i.e., two or more close records representing the same measurement result. In this paper, we address the problem of detecting duplicates in a database consisting of point measurements. As a test case, we use a database of measurements of anomalies in the Earth's gravity field that we have compiled. In this paper, we show that a natural duplicate deletion algorithm requires (in the worst case) quadratic time, and we propose a new asymptotically optimal $O(n \cdot \log(n))$ algorithm. These algorithms have been successfully applied to gravity databases. We believe that they will prove to be useful when dealing with many other types of point data.

1 Case Study: Geoinformatics Motivation for the Problem

Geospatial databases: general description. In many application areas, researchers and practitioners have collected a large amount of geospatial data.

For example, geophysicists measure values d of the gravity and magnetic fields, elevation, and reflectivity of electromagnetic energy for a broad range of wavelengths (visible, infrared, and radar) at different geographical points (x, y); see, e.g., [19]. Each type of data is usually stored in a large geospatial database that contains corresponding records (x_i, y_i, d_i). Based on these measurements, geophysicists generate maps and images and derive geophysical models that fit these measurements.

Gravity measurements: case study. In particular, measuring gravity is one of the most important sources of geophysical and geological information. There are two reasons for this importance. First, in contrast to more widely used geophysical data like remote sensing images, that mainly reflect the conditions of the Earth's surface, gravitation comes from the whole Earth (e.g., [9, 10]). Thus gravity data contain valuable information about much deeper geophysical structures. Second, in contrast to many types of geophysical data, which usually cover a reasonably local area, gravity measurements cover broad areas and thus provide important regional information.

The accumulated gravity measurement data are stored at several research centers around the world. One of these data storage centers is located at the University of Texas at El Paso (UTEP). This center contains gravity measurements collected throughout the United States and Mexico and parts of Africa.

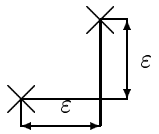
The geophysical use of gravity database compiled at UTEP is illustrated for a variety of scales in [1, 3, 6, 8, 12, 17, 20, 21].

Duplicates: where they come from. One of the main problems with the existing geospatial databases is that they are known to contain many duplicate points (e.g., [7, 14, 18]). The main reason why geospatial databases contain duplicates is that the databases are rarely formed completely "from scratch", by simply placing together the measurement results. These databases usually combine measurement results with the data from the existing databases. Some measurement results are represented in several of combined databases, so we get duplicate records.

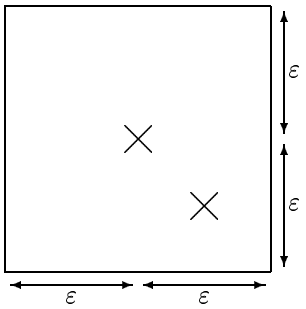
Why duplicates are a problem. Duplicate values can corrupt the results of statistical data processing and analysis. For example, when instead of a single (actual) measurement result, we see several measurement results confirming each other, and we may get an erroneous impression that this measurement result is more reliable that it actually is. Detecting and eliminating duplicates is therefore an important part of assuring and improving the quality of geospatial data, as recommended by the US Federal Standard [5].

Duplicates correspond to interval uncertainty. In the ideal case, when measurement results are simply stored in their original form, duplicates are identical records, so they are easy to detect and to delete. In reality, however, different databases may use different formats and units to store the same data: e.g., the latitude can be stored in degrees (as 32.1345) or in degrees, minutes,

and seconds. As a result, when a record (x_i, y_i, d_i) is placed in a database, it is transformed into this database's format. When we combine databases, we may need to transform these records into a new format – the format of the resulting database. Each transformation is approximate, so the records representing the same measurement in different formats get transformed into values which correspond to close but not identical points $(x_i, y_i) \neq (x_j, y_j)$. Usually, geophysicists can produce a threshold $\varepsilon > 0$ such that if the points (x_i, y_i) and (x_j, y_j) are ε-close – i.e., if $|x_i - x_j| \leq \varepsilon$ and $|y_i - y_j| \leq \varepsilon$ – then these two points are duplicates.



In other words, if a new point (x_j, y_j) is within a 2D *interval* $[x_i - \varepsilon, x_i + \varepsilon] \times [y_i - \varepsilon, y_i + \varepsilon]$ centered at one of the existing points (x_i, y_i), then this new point is a duplicate:



If the two points are duplicates, we should delete one of these two points from the database. Since the difference between the two points is small, it does not matter much which of the two points we delete. In other words, we want to continue deleting duplicates until we arrive at a "duplicate-free" database. There may be several such duplicate-free databases, all we need is one of them.

Duplicates are not easy to detect and delete. At present, the detection and deletion of duplicates is done mainly "by hand", by a professional geophysicist looking at the raw measurement results (and at the preliminary results of processing these raw data). This manual cleaning is very *time-consuming*. It is therefore necessary to design *automated* methods for detecting duplicates.

If the database was small, we could simply compare every record with every other record. This comparison would require $n(n-1)/2 \sim n^2/2$ steps. Alas,

real-life geospatial databases are often large, they may contain up to 10^6 or more records; for such databases, $n^2/2$ steps is too long. We need faster methods for deleting duplicates.

From interval to fuzzy uncertainty. Sometimes, instead of a single threshold value ε, geophysicists provide us with several possible threshold values $\varepsilon_1 < \varepsilon_2 < \ldots < \varepsilon_m$ that correspond to decreasing levels of their certainty:

- if two measurements are within ε_1 from each other, then we are 100% certain that they are duplicates;
- if two measurements are within ε_2 from each other, then with some degree of certainty, we can claim them to be duplicates,
- if two measurements are within ε_2 from each other, then with an even smaller degree of certainty, we can claim them to be duplicates,
- etc.

In this case, we must eliminate *certain* duplicates, and mark *possible* duplicates (about which are not 100% certain) with the corresponding degree of certainty.

In this case, for each of the coordinates x and y, instead of a single interval $[x_i - \varepsilon, x_i + \varepsilon]$, we have a nested family of intervals $[x_i - \varepsilon_j, x_i + \varepsilon_j]$ corresponding to different degrees of certainty. Such a nested family of intervals is also called a *fuzzy set*, because it turns out to be equivalent to a more traditional definition of fuzzy set [2, 13, 15, 16] (if a traditional fuzzy set is given, then different intervals from the nested family can be viewed as α-cuts corresponding to different levels of uncertainty α).

In these terms, in addition to detecting and deleting duplicates under interval uncertainty, we must also detect and delete them under fuzzy uncertainty.

What we are planning to do. In this paper, we propose methods for detecting and deleting duplicates under interval and fuzzy uncertainty, and test these methods on our database of measurements of the Earth's gravity field.

2 Geospatial Databases: Brief Introduction

Geospatial databases: formal description. In accordance with our description, a *geospatial database* can be described as a finite set of *records* $r_1, \ldots, r_n$, each of which is a triple $r_i = (x_i, y_i, d_i)$ consisting of two rational numbers x_i and y_i that describe coordinates and some additional data d_i.

The need for sorting. One of the main objectives of a geospatial database is to make it easy to find the information corresponding to a given geographical area. In other words, we must be able, given one or two coordinates (x and/or y) of a geographical point (center of the area of interest), to easily find the data corresponding to this point and its vicinity.

It is well known that if the records in a database are not sorted by a parameter a, then in order to find a record with a given value of a, there is no faster way than linear (exhaustive) search, in which we check the records one by one until we find the desired one. In the worst case, linear search requires searching over all n records; on average, we need to search through $n/2$ records. For a large database with thousands and millions of record, this takes too much times.

To speed up search, it is therefore desirable to *sort* the records by the values of a, i.e., to reorder the records in such a way that the corresponding values of a are increasing: $a_1 \le a_2 \le \ldots \le a_n$.

Once the records are sorted, instead of the time-consuming linear search, we can use a much faster *binary search* (also known as *bisection*). At each step of the binary search, we have an interval $a_l \le a \le a_u$. We start with $l = 1$ and $u = n$. On each step, we take a midpoint $m = \lfloor (l + u)/2 \rfloor$ and check whether $a < a_m$. If $a < a_m$, then we have a new half-size interval $[a_l, a_{m-1}]$; otherwise, we have a half-size interval $[a_m, a_u]$ containing a. In $\log_2(n)$ steps, we can thus locates the record corresponding to the desired value of a.

How to Sort: Mergesort Algorithm. Sorting can be done, e.g., by *mergesort* – an asymptotically optimal sorting algorithm that sorts in $O(n \cdot \log(n))$ computational steps (see, e.g., [4]).

Since the algorithms that we use for deleting duplicates are similar to mergesort, let us briefly describe how mergesort works. This algorithm is *recursive* in the sense that, as part of applying this algorithm to the databases, we apply this same algorithm to its sub-databases. According to this algorithm, in order to sort a list consisting of n records $r_1, \ldots, r_n$, we do the following:

- first, we apply the same mergesort algorithm to sort the first half of the list, i.e., the records $\langle r_1, \ldots, r_{\lfloor n/2 \rfloor} \rangle$ (if we only have one record in this half-list, then this record is already sorted);
- second, we apply the same mergesort algorithm to sort the remaining half of the list, i.e., the records $\langle r_{\lfloor n/2 \rfloor + 1}, \ldots, r_n \rangle$ (if we only have one record in this half-list, then this record is already sorted);
- finally, we merge the two sorted half-lists into a single sorted list; we start with an empty sorted list; then, at each step, we compare the smallest two elements of the remaining half-lists, and move the smaller of them to the next position on the merged list.

For example, if we start with sorted half-lists $\langle 10, 30 \rangle$ and $\langle 20, 40 \rangle$, then we do the following:

- First, we compare 10 and 20, and place the smaller element 10 as the first element of the originally empty sorted list.

- Then, we compare the first elements 30 and 20 of the remaining half-lists $\langle 30\rangle$ and $\langle 20, 40\rangle$ and place 20 as the second element into the sorted list – so that the sorted list now becomes $\langle 10, 20\rangle$.
- Third, we compare the first elements 30 and 40 of the remaining half-lists $\langle 30\rangle$ and $\langle 40\rangle$, and place 30 as the next element into the sorted list – which is now $\langle 10, 20, 30\rangle$.
- After that, we have only one remaining element, so we place it at the end of the sorted list – making it the desired $\langle 10, 20, 30, 40\rangle$.

How many computational steps does this algorithm take? Let us start counting with the merge stage. In the merge stage, we need (at most) one comparison to get each element of the resulting sorted list. So, to get a sorted list of n elements, we need $\le n$ steps. If by $t(n)$, we denote the number of steps that mergesort requires on lists of size n, then, from the structure of the algorithm, we can conclude that $t(n) \le 2 \cdot t(n/2) + n$. If $n/2 > 1$, we can similarly conclude that $t(n/2) \le 2 \cdot t(n/4) + n/2$ and therefore, that

$$t(n) \le 2 \cdot t(n/2) + n \le 2 \cdot (2 \cdot t(n/2) + n/2) + n \le$$

$$4 \cdot t(n/4) + 2 \cdot (n/2) + n = 4 \cdot t(n/4) + 2n.$$

Similarly, for every k, we can conclude that $t(n) \le 2^k \cdot t(n/2^k) + k \cdot n$. In particular, when $n = 2^k$, then we can choose $k = \log_2(n)$ and get $t(n) \le n \cdot y(1) + k \cdot n$. A list consisting of a single element is already sorted, so $t(1) = 0$ hence $t(n) \le k \cdot n$, i.e., $t(n) \le n \cdot \log_2(n)$.

Specifics of geospatial databases. In a geospatial database, we have two coordinates by which we may want to search: x and y. If we sort the records by x, then search by x becomes fast, but search by y may still require a linear search – and may thus take a lot of computation time.

To speed up search by y, a natural idea is to sort the record by y as well – with the only difference that we do not physically reorder the records, we just memorize where each record should be when sorted by y. In other words, to speed up search by x and y, we do the following:

- First, we sort the records by x, so that $x_1 \le x_2 \le \ldots \le x_n$.
- Then, we sort these same records by y, i.e., produce n different values $i_1, \ldots, i_n$ such that $y_{i_1} \le y_{i_2} \le \ldots \le y_{i_n}$ (and n values $j(1), \ldots, j(n)$ such that $j(i_k) = k$).

For example, if we start with the records corresponding to the points $(20, 10)$, $(10, 40)$, and $(30, 30)$, then we:

- first, sort them by x, ending in $(x_1, y_1) = (10, 40)$, $(x_2, y_2) = (20, 10)$, and $(x_2, y_2) = (30, 30)$;

- then, sort the values of y; we end up with $i_1 = 2$, $i_2 = 3$ and $i_3 = 1$ (and, correspondingly, $j(1) = 3$, $j(2) = 1$, and $j(3) = 2$), so that

$$y_{i_1} = y_2 = 10 \le y_{i_2} = y_3 = 30 \le y_{i_3} = y_1 = 40.$$

The resulting "double-sorted" database enables us to search fast both by x and by y.

3 The Problem of Deleting Duplicates: Ideal Case of No Uncertainty

To come up with a good algorithm for detecting and eliminating duplicates in case of interval uncertainty, let us first consider an ideal case when there is no uncertainty, i.e., when duplicate records $r_i = (x_i, y_i, d_i)$ and $r_j = (x_j, y_j, d_j)$ mean that the corresponding coordinates are equal: $x_i = x_j$ and $y_i = y_j$.

In this case, to eliminate duplicates, we can do the following. We first sort the records in lexicographic order, so that r_i goes before r_j if either $x_i < x_j$, or $x_i = x_j$ and $y_i \le y_j$. In this order, duplicates are next to each other.

So, we first compare r_1 with r_2. If coordinates in r_2 are identical to coordinates in r_1, we eliminate r_2 as a duplicate, and compare r_1 with r_3, etc. After the next element is no longer a duplicate, we take the next record after r_1 and do the same for it, etc.

After each comparison, we either eliminate a record as a duplicate, or move to a next record. Since we only have n records in the original database, we can move only n steps to the right, and we can eliminate no more than n records. Thus, totally, we need no more than $2n$ comparison steps to complete our procedure.

Since $2n$ is asymptotically smaller than the time $n \cdot \log(n)$ needed to sort the record, the total time for sorting and deleting duplicates is $n \cdot \log(n) + 2n \sim n \cdot \log(n)$. Since we want a sorted database as a result, and sorting requires at least $n \cdot \log(n)$ steps, this algorithm is asymptotically optimal.

It is important to mention that this process does not have to be sequential: if we have several processors, then we can eliminate records in parallel, we just need to make sure that if two record are duplicates, e.g., $r_1 = r_2$, then when one processor eliminates r_1 the other one does not eliminate r_2.

Formally, we say that a subset of the database is obtained by a *cleaning step* if:

- it is obtained from the original database by selecting one or several different pairs of duplicates and deleting one duplicate from each pair, and
- from each *duplicate chain* $r_i = r_j = \ldots = r_k$, at least record remains in the database after deletion.

A sequence of cleaning steps after which the resulting subset is duplicate-free (i.e., does not contain any duplicates) is called *deleting duplicates*.

The goal is to produce a (duplicate-free) subset of the original database obtained by deleting duplicates – and to produce it sorted by x_i.

4 Interval Modification of the Above Algorithm: Description, Practicality, Worst-Case Complexity

In the previous section, we described how to eliminate duplicates in the ideal case when there is no uncertainty.

In real life, as we have mentioned, there is an interval uncertainty. A natural idea is therefore to modify the above algorithm so that it detects not only exact duplicate records but also records that are within ε of each other.

In precise terms, we have a geospatial database $\langle r_1, \ldots, r_n \rangle$, where $r_i = (x_i, y_i, d_i)$, and we are also given a positive rational number ε. We say that records $r_i = (x_i, y_i, d_i)$ and $r_j = (x_j, y_j, d_j)$ are *duplicates* (and denote it by $r_i \sim r_j$) if $|x_i - x_j| \le \varepsilon$ and $|y_i - y_j| \le \varepsilon$.

We say that a subset of the database is obtained by a *cleaning step* if:

- it is obtained from the original database by selecting one or several different pairs of duplicates and deleting one duplicate from each pair, and
- from each *duplicate chain* $r_i \sim r_j \sim \ldots \sim r_k$, at least record remains in the database after deletion.

A sequence of cleaning steps after which the resulting subset is duplicate-free (i.e., does not contain any duplicates) is called *deleting duplicates*.

The goal is to produce a (duplicate-free) subset of the original database obtained by deleting duplicates – and to produce it sorted by x_i (and double-sorted by y).

Similarly to the ideal case of no uncertainty, to avoid comparing all pairs (r_i, r_j) – and since we need to sort by x_i anyway – we first sort the records by x, so that $x_1 \le x_2 \le \ldots \le x_n$. Then, first we detect and delete all duplicates of r_1, then we detect and delete all duplicates of r_2 (r_1 is no longer considered since its duplicates have already been deleted), then duplicates of r_3 (r_1 and r_2 are no longer considered), etc.

For each i, to detect all duplicates of r_i, we check r_j for the values $j = i+1, i+2, \ldots$ while $x_j \le x_i + \varepsilon$. Once we have reached the value j for which $x_j > x_i + \varepsilon$, then we can be sure (since the sequence x_i is sorted by x) that $x_k > x_i + \varepsilon$ for all $k \ge j$ and hence, none of the corresponding records r_k can be duplicates of r_i.

While $x_j \le x_i + \varepsilon$, we have $x_i \le x_j \le x_i + \varepsilon$ hence $|x_i - x_j| \le \varepsilon$. So, for these j, to check whether r_i and r_j are duplicates, it is sufficient to check whether $|y_i - y_j| \le \varepsilon$.

Thus, the following algorithm solves the problem of deleting duplicates:

Algorithm 1.

1. Sort the records by x_i, so that $x_1 \le x_2 \le \ldots \le x_n$.
2. For i from 1 to $n-1$, do the following:

   for $j = i+1, i+2, \ldots$, while $x_j \le x_i + \varepsilon$
   if $|y_j - y_i| \le \varepsilon$, delete r_j.

For the gravity database, this algorithm works reasonably well, but we cannot be sure that it will always work well, because its worst-case complexity is still $n(n-1)/2$. Indeed, if all n records have the same value of x_i, and all the values y_i are drastically different: e.g., $y_i = y_1 + 2 \cdot (i-1) \cdot \varepsilon$ – then the database is duplicate-free, but the above algorithm requires that we compare all the pairs.

For gravity measurements, this is, alas, a very realistic situation, because measurements are sometimes made when a researcher travels along a road and makes measurements along the way – and if the road happens to be vertical ($x \approx$ const), we end up with a lot of measurements corresponding to very close values of x.

We therefore need a faster algorithm for deleting duplicates.

5 New Algorithm: Motivations, Description, Complexity

How can we speed up the above algorithm? The above example of when the above algorithm does not work well shows that it is not enough to sort by x – we also need to sort by y. In other words, it makes sense to have an algorithm with the following structure:

Algorithm 2.

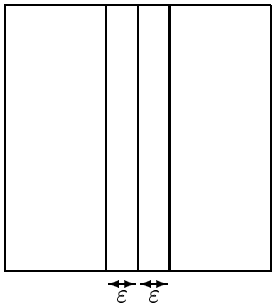
1. Sort the records by x, so that $x_1 \le x_2 \le \ldots \le x_n$.
2. Sort these same records by y, i.e., produce n different values $i_1, \ldots, i_n$ such that $y_{i_1} \le y_{i_2} \le \ldots \le y_{i_n}$ (and n values $j(1), \ldots, j(n)$ such that $j(i_k) = k$).
3. Delete duplicates from the resulting "double-sorted" database.

To describe the main part – Part 3 – of this algorithm, we will use the same recursion that underlies mergesort:

Part 3 of Algorithm 2. To delete duplicates from the double-sorted database $\langle r_1, \ldots, r_n \rangle$, we do the following:

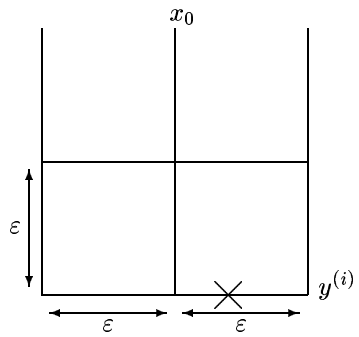
1. We apply the same Part 3 of Algorithm 2 to delete duplicates from the left half $\langle r_1, \ldots, r_{\lfloor n/2 \rfloor} \rangle$ of the database (if we only have one record in this half-list, then this half-list is free of duplicates, so we do not need to delete anything);
2. We apply the same Part 3 of Algorithm 2 to delete duplicates from the right half $\langle r_{\lfloor n/2 \rfloor + 1}, \ldots, r_n \rangle$ of the database (similarly, if we only have one record in this half-list, then this half-list is free of duplicates, so we do not need to delete anything);
3. We merge and clean the resulting duplicate-free subsets by using an appropriate merge-and-clean algorithm.

How can we merge and clean? Since both merged databases are duplicate-free, the only possible duplicates in their union is when r_i is from the first half-database, and r_j is from the second half-database. Since the records are sorted by x, for the first database, $x_i \le x_0 \stackrel{\text{def}}{=} x_{\lfloor n/2 \rfloor}$, and for the second database, $x_0 \le x_j$, so $x_i \le x_0 \le x_j$. If r_i and r_j are duplicates, then the distance $|x_i - x_j|$ between x_i and x_j does not exceed ε, hence the distance between each of these values x_i, x_j and the intermediate point x_0 also cannot exceed ε. Thus, to detect duplicates, it is sufficient to consider records for which $x_i, x_j \in [x_0 - \varepsilon, x_0 + \varepsilon]$ – i.e., for which x_i belongs to the narrow interval centered in x_0.



It turns out that for these points, the above Algorithm 1 (but based on sorting by y) runs fast. Indeed, since we have already sorted the values y_i, we can sort all k records for which x is within the above narrow interval by y, into a sequence $r_{(1)}, \ldots, r_{(k)}$ for which $y_{(1)} \le y_{(2)} \le \ldots \le y_{(k)}$. Then, according to Algorithm 1, we should take each record $r_{(i)}$, $i = 1, 2, \ldots$, and check whether any of the following records $r_{(i+1)}, r_{(i+2)}, \ldots$ is a duplicate of the record $r_{(i)}$.

For each record $r_{(i)} = (x_{(i)}, y_{(i)}, d_{(i)})$, desired duplicates (x, y, d) must satisfy the condition $y_{(i)} \le y \le y_{(i)} + \varepsilon$; the corresponding value x is, as we have mentioned, between $x_0 - \varepsilon$ and $x_0 + \varepsilon$; thus, for duplicates, the coordinates (x, y) must come either from the square $[x_0 - \varepsilon, x_0] \times [y_{(i)}, y_{(i)} + \varepsilon]$ (corresponding to the first half-database) or from the square $[x_0, x_0 + \varepsilon] \times [y_{(i)}, y_{(i)} + \varepsilon]$ (corresponding to the second half-database).



Each of these two squares is of size $\varepsilon \times \varepsilon$, therefore, within each square, every two points are duplicates. Since we have already deleted duplicates within each of the two half-databases, this means that within each square, there is no more than one record. The original record $r_{(i)}$ is within one of these squares, so this square cannot have any more records $r_{(j)}$; thus, only the other square can have another record $x_{(j)}$ inside. Since the records are sorted by y, and $r_{(j)}$ is the only possible record with $y_{(i)}) \le y_{(j)} \le y_{(i)} + \varepsilon$, this possible duplicate record (if it exists) is the next one to $r_{(i)}$, i.e., it is $r_{(i+1)}$. Therefore, to check whether there is a duplicate to $r_{(i)}$ among the records records $r_{(i+1)}, r_{(i+2)}, \ldots$, it is sufficient to check whether the record $r_{(i+1)}$ is a duplicate for $r_{(i)}$. As a result, we arrive at the following "merge and clean" algorithm:

Merge and Clean Algorithm.

1. Select all the records r_i from both merged half-databases for which $x_i \in [x_0 - \varepsilon, x_0 + \varepsilon]$, where $x_0 \stackrel{\text{def}}{=} x_{\lfloor n/2 \rfloor}$.
2. Since we have already sorted the values y_i, we can sort the all the selected records by y into a sequence $r_{(1)}, \ldots, r_{(k)}$ for which $y_{(1)} \le y_{(2)} \le \ldots \le y_{(k)}$.
3. For i from 1 to $k-1$, if $|y_{(i+1)} - y_{(i)}| \le \varepsilon$ and $|x_{(i+1)} - x_{(i)}| \le \varepsilon$, delete $r_{(i+1)}$.

This completes the description of Algorithm 2. In the process of designing this algorithm, we have already proven that this algorithm always returns the solution to a problem of deleting duplicates. The following result show that this algorithm is indeed asymptotically optimal:

Proposition 1. *Algorithm* 2 *requires* $O(n \cdot \log(n))$ *steps in the worst case, and no algorithm with asymptotically smaller worst-case complexity is possible.*

Proof. Algorithm 2 consists of two sortings – each of which requires $O(n \cdot \log(n))$ steps – and the main part. Application of the main part to n records consist of two applications of the main part to $n/2$ records plus merge. Merging, as we have seen, requires no more than n steps; therefore, the worst-case complexity of applying the main part to a list of n elements can be bounded by $2t(n/2) + n$:

$t(n) \le 2t(n/2) + n$. From the functional inequality, we can conclude (see, e.g., [4]) that the main part requires $t(n) = O(n \cdot \log(n))$ steps. Thus, the total time of Algorithm 2 is also $\le O(n \cdot \log(n))$.

On the other hand, since our problem requires sorting, we cannot solve it faster than in $O(n \cdot \log(n))$ steps that are needed for sorting [4]. Proposition is proven.

6 Deleting Duplicates Under Fuzzy Uncertainty

As we have mentioned, in some real-life situations, in addition to the threshold ε that guarantees that vε-close data are duplicates, the experts also provide us with additional threshold values $\varepsilon_i > \varepsilon$ for which ε_i-closeness of two data points means that we can only conclude with a certain degree of certainty that one of these data points is a duplicate. The corresponding degree of certainty decreases as the value ε_i increases.

In this case, in addition to deleting records that are absolutely certainly duplicates, it is desirable to mark possible duplicates – so that a professional geophysicist can make the final decision on whether these records are indeed duplicates.

A natural way to do this is as follows:

- First, we use the above algorithm to delete all the certain duplicates (corresponding to ε).
- Then, we use the same algorithm to the remaining records and mark (but not actually delete) all the duplicates corresponding to the next value ε_2. The resulting marked records are duplicates with the degree of confidence corrsponding to ε_2.
- After that, we apply the same algorithm with the value ε_3 to all unmarked records, and mark those which the algorithm detects as duplicates with the degree of certainty corresponding to ε_3,
- etc.

In other words, to solve a fuzzy problem, we solve several interval problems corresponding to different levels of uncertainty. It is worth mentioning that this "interval" approach to solving a fuzzy problem is in line with many other algorithms for processing fuzzy data; see, e.g., [2, 13, 15, 16].

7 Possibility of Parallelization

If we have several processors that can work in parallel, we can speed up computations:

Proposition 2. *If we have an unlimited number of processors (at least $n^2/2$), then we can delete duplicates in $O(n \cdot \log(n))$ steps.*

Proof. It is known that we can sort a list in parallel in $O(\log(n))$ steps; see, e.g., [11]. Let us show that after sorting by x, deleting duplicates can actually be done in a single additional step. For n records, we have $n \cdot (n-1)/2$ pairs to compare. We can let each of $\geq n^2/2$ processors handle a different pair, and, if elements of the pair turn out to be duplicates, delete one of them. Thus, we indeed delete all duplicates in a single step. The proposition is proven.

In the proof of Proposition 2, we have actually proven the following result:

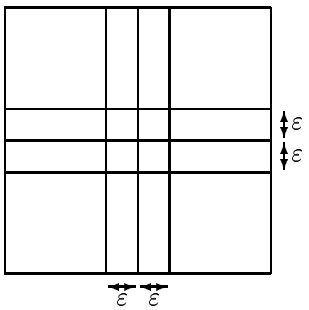
Proposition 3. *If we have an unlimited number of processors (at least $n^2/2$), and the records are already sorted by x, then we can delete duplicates in a single step.*

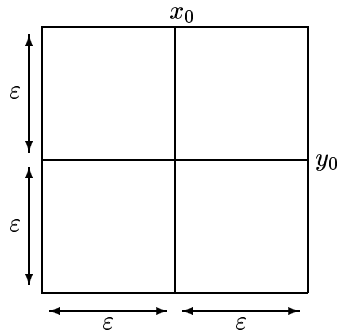
If we have fewer than $n^2/2$ processors, we also get a speed up:

Proposition 4. *If we have at least n processors, then we can delete duplicates in $O(\log^2(n))$ time.*

Proof. Let us show how Algorithm 2 can be implemented in parallel. As we have mentioned in the proof of Proposition 2, sorting can be done in $O(\log(n))$ steps.

In the main step, we can clean both half-databases in parallel, thus, the total time $t(n)$ for cleaning the database with n records is equal to the time $t(n/2)$ of cleaning a database with $n/2$ records plus the time $t_m(n)$ for merging (and cleaning) the cleaned half-databases. This merging and cleaning can also be done in parallel: one group of processors merges and cleans records $r_{(1)}, \ldots, r_{(\lfloor k/2 \rfloor)}$, while the other group of processors merges and cleans records $r_{(\lfloor k/2 \rfloor+1)}, \ldots, r_{(k)}$. After that, the only remaining duplicates can be in the $2\varepsilon \times 2\varepsilon$ square $S \stackrel{\text{def}}{=} [x_0 - \varepsilon, x_0 + \varepsilon] \times [y_0 - \varepsilon, y_0 + \varepsilon]$, where $y_0 \stackrel{\text{def}}{=} y_{(\lfloor k/2 \rfloor)}$:





Similarly to the proof of Proposition 2, it is easy to show that each of the four $\varepsilon \times \varepsilon$ squares that form S can contain at most one record, so the number of steps required to detect and delete the corresponding duplicates is bounded by a constant C. Thus, $t_m(n) \le t_m(n/2) + C$, hence $t_m(n) \le C \cdot \log(n)$. Due to $t(n) \le t(n/2) + t_m(n) \le t(n/2) + C \cdot \log(n)$, we can now conclude that $t(n) = O(\log^2(n))$. The proposition is proven.

If we have a smaller number $p < n$ processors, then parallelization of Algorithm 2 leads to the following result:

Proposition 5. *If we have $p < n$ processors, then we can delete duplicates in $O\left(\frac{n}{p} \cdot \log(n) + \log(n) + \log^2(p)\right)$ time.*

Proof. It is known that sorting can be done in time $O\left(\frac{n}{p} \cdot \log(n) + \log(n)\right)$ [11].

In the main step – actually deleting duplicates – we divide n records, in the order of x_i, into p segments of size n/p each, so the first segment contains records $x_1, x_2, \ldots, x_{\lfloor n/p \rfloor}$, etc. Then, each of p processors uses Algorithm 2 to delete duplicates from the corresponding segment; according to Proposition 1, this deletion requires time $O\left(\frac{n}{p} \cdot \log\left(\frac{n}{p}\right)\right)$.

After that, we do the merging as follows:

- We divide p processors into $p/2$ pairs: #1 with # 2, # 3 with # 4, etc., and use each pair to merge the corresponding segments with each other. To clean the result of merging the two segments, we divide (as in the proof of Proposition 4) all $\le 2n/p$ records in the narrow border zone into two halves, clean both halves in parallel – which takes $\le n/p$ time, and then clean the $2\varepsilon \times 2\varepsilon$ square – which requires a constant number of steps C. As a result of this merging, we get $p/2$ cleaned segments of size $2n/p$.

- Now, we divide processors into $p/4$ groups of 4: #1 through # 4, # 5 through # 8, etc., and use each group to merge and clean the two neighboring segments of size $2n/p$. In cleaning the narrow zone, we divide the records within this zone (there are $\le 4n/p$ of them) into 4 subgroups of size $\le n/p$, and merge then in parallel; this takes $\le n/p$ time. Then,

we must merge these subgroups. First, we merge, in parallel, the 1st and the 2nd subgroup, and the 3rd and the 4th subgroup; after that, we merge them together. Each merging requires a constant time C, so we need a total time of $2C$. As a result of this merging, we get $p/4$ cleaned segments of size $4n/p$.

- In general, after k merger steps, we get $p/2^k$ cleaned segments of size $2^k \cdot n/p$. We divide processors into $p/2^{k+1}$ groups of 2^{k+1}, and and use each group to merge and clean two neighboring segments of size $2^k \cdot n/p$. In cleaning the narrow zone, we divide the records within this zone (there are $\leq 2^{k+1} \cdot n/p$ of them) into 2^{k+1} subgroups of size $\leq n/p$, and merge then in parallel; this takes $\leq n/p$ time. Then, we must merge these subgroups. First, we merge, in parallel, the 1st and the 2nd subgroup, and the 3rd and the 4th subgroup; after that, we merge these subgroups into groups of twice larger size, etc. There are $\leq 2^{k+1} \cdot n/p$ records in each narrow zone, so we need at most $k+1$ sequential mergers. Each merging requires C steps, so we need a total time of $(k+1) \cdot C$ to merge.

Overall, we continue until we get a single cleaned database, i.e., until $2^k \cdot n/p = n$. In other words, we need $k = \log(p)$ iterations. During each iteration, we need time $\leq n/p + k \cdot C$ to merge, so the total time is $\leq (n/p) \cdot \log(p) + C \cdot \log^2(p)$. Since $p \leq n$, this time is $\leq (n/p) \cdot \log(n) + C \cdot \log^2(p)$.

Adding up the times necessary for sorting, for cleaning p segments, and for merging them into a single cleaned database, we get the desired estimate. The proposition is proven.

8 The Problem of Deleting Duplicates: Multi-D Version

At present, the most important case of duplicate detection is a 2-D case, when record are 2-dimensional, i.e., of the type $r = (x, y, d)$. What is we have multi-D records of the type $r = (x_1, \ldots, x_m, d)$, and we define $r = (x_1, \ldots, x_m, d)$ and $r' = (x'_1, \ldots, x'_m, d')$ to be duplicates if $|x_i - x'_i| \leq \varepsilon$ for all i? For example, we may have measurements of geospatial data not only at different locations (x_1, x_2), but also at different depths x_3 within each location.

In this case, we can still use an $O(n^2)$ Algorithm 1, but we can also use a faster algorithm modeled along the lines of Algorithm 2:

Proposition 6. *For every $m \geq 2$, there exists an algorithm A_m that solves the duplicate deletion problem in time $O(n \cdot \log^{m-1}(n))$.*

Proof. The main idea of these algorithms is similar to the main idea behind Algorithm 2: we divide the database into two halves, delete duplicates from each half, and then merge and clean the resulting half-databases. When both

halves are cleaned, we only need to clean the values for which one of m variables – e.g., x_1 – lies within a narrow interval of width 2ε. For that, we can use the algorithm A_{m-1} for cleaning $(m-1)$-dimensional databases.

Let us prove, by induction over m, that the worst-case complexity $t_m(n)$ of algorithm A_m is $O(n \cdot \log^{m-1}(n))$.

- *Base.* We have already proven this for $m = 2$.
- *Induction step.* In general, $t_m(n) \le 2t_m(n/2)+t_{m-1}(n)$, so if we know that $t_{m-1}(n) = O(n \cdot \log^{m-2}(n))$, we can conclude – using known techniques (see, e.g., [4]) – that $t_m(n) = O(n \cdot \log^{m-1}(n))$.

The proposition is proven.

For $m = 2$, the above algorithm is – as we have shown – asymptotically optimal. Whether it is asymptotically optimal for $m > 2$ is an open question.

Acknowledgments

This work was supported in part by NASA under cooperative agreement NCC5-209 and grant NCC2-1232, by Future Aerospace Science and Technology Program (FAST) Center for Structural Integrity of Aerospace Systems, effort sponsored by the Air Force Office of Scientific Research, Air Force Materiel Command, USAF, under grant F49620-00-1-0365, and by NSF grants CDA-9522207, EAR-0112968, EAR-0225670, and 9710940 Mexico/Conacyt.

This research was partly done when V.K. was a Visiting Faculty Member at the Fields Institute for Research in Mathematical Sciences (Toronto, Canada)

References

[1] Adams, D.C., Keller, G.R., 1996. Precambrian basement geology of the Permian Basin region of West Texas and eastern New Mexico: A geophysical perspective, American Association of Petroleum Geologists Bulletin 80, 410–431.

[2] Bojadziev, G., and Bojadziev, M., *Fuzzy sets, fuzzy logic, applications*, World Scientific, Singapore, 1995.

[3] Cordell, L., Keller, G.R., 1982. Bouguer Gravity Map of the Rio Grande Rift, Colorado, New Mexico, and Texas Geophysical investigations series, U.S. Geological Survey.

[4] Cormen, T.H., Leiserson, C.E., Rivest, R.L., and Stein, C., 2001. Introduction to Algorithms, MIT Press, Cambridge, MA, and Mc-Graw Hill Co., N.Y.

[5] FGDC Federal Geographic Data Committee, 1998. FGDC-STD-001-1998. Content standard for digital geospatial metadata (revised June 1998), Federal Geographic Data Committee, Washington, D.C., http://www.fgdc.gov/metadata/contstan.html

[6] Fliedner, M.M., Ruppert, S.D., Malin, P.E., Park, S.K, Keller, G.R., Miller, K.C., 1996. Three-dimensional crustal structure of the southern Sierra Nevada from seismic fan profiles and gravity modeling, Geology 24, 367–370.

[7] Goodchild, M., Gopal, S. (Eds.), 1989. Accuracy of Spatial Databases, Taylor & Francis, London.

[8] Grauch, V.J.S., Gillespie, C.L., Keller, G.R., 1999. Discussion of new gravity maps of the Albuquerque basin, New Mexico Geol. Soc. Guidebook 50, 119–124.

[9] Heiskanen, W.A., Meinesz, F.A., 1958. The Earth and its gravity field, McGraw-Hill, New York.

[10] Heiskanen, W.A., Moritz, H., 1967. Physical Geodesy, W.H. Freeman and Company, San Francisco, California.

[11] Jájá, J., 1992. An Introduction to Parallel Algorithms, Addison-Wesley, Reading, MA.

[12] Keller, G.R., 2001. Gravitational Imaging, In: The Encyclopedia of Imaging Science and Technology, John Wiley, New York.

[13] Klir, G., and Yuan, B., *Fuzzy Sets and Fuzzy Logic: Theory and Applications*, Prentice Hall, Upper Saddle River, NJ, 1995.

[14] McCain, M., William C., 1998. Integrating Quality Assurance into the GIS Project Life Cycle, Proceedings of the 1998 ESRI Users Conference. http://www.dogcreek.com/html/documents.html

[15] Nguyen, H. T., and Kreinovich, V., "Nested Intervals and Sets: Concepts, Relations to Fuzzy Sets, and Applications", In: R. B. Kearfott et al (eds.), *Applications of Interval Computations*, Kluwer, Dordrecht, 1996, 245–290.

[16] Nguyen, H. T., and Walker, E. A., *First Course in Fuzzy Logic*, CRC Press, Boca Raton, FL, 1999.

[17] Rodriguez-Pineda, J.A., Pingitore, N.E., Keller, G.R., Perez, A., 1999. An integrated gravity and remote sensing assessment of basin structure and hydrologic resources in the Chihuahua City region, Mexico, Engineering and Environ. Geoscience 5, 73–85.

[18] Scott, L., 1994. Identification of GIS Attribute Error Using Exploratory Data Analysis, Professional Geographer 46(3), 378–386.

[19] Sharma, P., 1997. Environmental and Engineering Geophysics, Cambridge University Press, Cambridge, U.K.

[20] Simiyu, S.M. Keller, G.R., 1997. An integrated analysis of lithospheric structure across the East African Plateau based on gravity anomalies and recent seismic studies, Tectonophysics 278, 291–313.

[21] Tesha, A.L., Nyblade, A.A., Keller, G.R., Doser, D.I., 1997. Rift localization in suture-thickened crust: Evidence from Bouguer gravity anomalies in northeastern Tanzania, East Africa, Tectonophysics, 278, 315–328.