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], Luc Longpré[1,2], and Scott A. Starks[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., [35]. 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, gravity measurements are one of the most important sources of information about subsurface structure and physical conditions. 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., [19, 20]). 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.

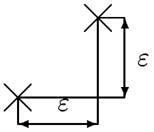
The accumulated gravity measurement data are stored at several research centers around the world. One of these data storage centers is located at the University of Texas at El Paso (UTEP). This center contains gravity measurements collected throughout the United States and Mexico and parts of Africa.

The geophysical use of gravity database compiled at UTEP is illustrated for a variety of scales in [1, 6, 13, 18, 22, 32, 36, 38].

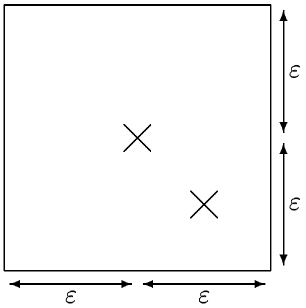
Duplicates: where they come from. One of the main problems with the existing geospatial databases is that they are known to contain many duplicate points (e.g., [16, 28, 34]). The main reason why geospatial databases contain duplicates is that the databases are rarely formed completely "from scratch", and instead are built by combining measurements from numerous sources. Since some measurements are represented in the data from several of the sources, 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 than 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 [12].

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 $|x_i - x_j| \le \varepsilon$ and $|y_i - y_j| \le \varepsilon$ – 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:



From the practical viewpoint, it usually does not matter which of the duplicate points we delete. 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.

To be more precise, we say that a subset of the original 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 one 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.

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 [3, 23, 29, 30] (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.

Comment. In our specific problem of detecting and deleting duplicates in geospatial databases, the only fuzziness that is important to us is the simple fuzziness of the threshold, when the threshold is a fuzzy number – or, equivalently, when we have several different threshold values corresponding to different levels of certainty.

It is worth mentioning that in other important geospatial applications, other – more sophisticated – fuzzy models and algorithms turned out to be very useful. There are numerous papers on this topic, let us just give a few relevant examples:

- fuzzy numbers can be used to describe the uncertainty of measurement results, e.g., the results of measuring elevation; in this case, we face an interesting (and technically difficult) interpolation problem of reconstructing the (fuzzy) surface from individual fuzzy measurement results; see, e.g., [27, 33];
- fuzzy sets are much more adequate than crisp sets in describing geographic entities such as biological species habitats, forest regions, etc.; see, e.g., [14, 15];
- fuzzy sets are also useful in describing to what extent the results of data processing are sensitive to the uncertainties in raw data; see, e.g., [25, 26].

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.

Some of our results have been announced in [4].

2 Relation to Computational Geometry

Intersection of rectangles. The problem of deleting duplicates is closely related to several problems that are solved in Computational Geometry.

One of such problems if the problem of intersection of rectangles. Namely, if around each point (x_i, y_i), we build a rectangle

$$R_i = \left[x_i - \frac{\varepsilon}{2}, x_i + \frac{\varepsilon}{2}\right] \times \left[y_i - \frac{\varepsilon}{2}, y_i + \frac{\varepsilon}{2}\right],$$

then, as one can easily see, the points (x_i, y_i) and (x_j, y_j) are duplicates if and only if the corresponding rectangles R_i and R_j intersect.

Problems related to intersection of rectangles are well known – and well understood – in Computational Geometry; see, e.g., [2, 8, 17, 24, 31]. Among these problems, the closest to deleting duplicates is the *reporting problem:* given n rectangles, list all intersecting pairs. Once the reporting problem is solved, i.e., once we have a list of intersecting pairs, then we can easily delete all the duplicates: e.g., we can simply delete, from each intersecting pair $R_i \cap R_j \neq \emptyset$, a record with the larger index.

There exist algorithms that solve the reporting problem in time

$$O(n \cdot \log(n) + k),$$

where k is the total number of intersecting pairs; see, e.g., [31]. Readers of *Reliable Computing* may be interested to know that some of these algorithms use a special data structure – *interval tree* introduced first in [9]; a detailed description of how interval trees can be used to solve the reporting problem is given in [31].

It is easy to conclude that since we actually need to list all k pairs, we cannot solve this problem in time smaller than $O(k)$. In the cases when there are few intersecting pairs – i.e., when k is small – we thus have a fast algorithm for deleting duplicates.

However, in principle, the number of intersecting pairs can be large. For example, if all the records are duplicates of the same record, then all the pairs intersect, so we have $k = n \cdot (n-1)/2 \sim n^2$ intersecting pairs. This is an extreme case, but we can have large numbers of intersecting pairs in more realistic situations as well. So, in the worst case, duplicate detection methods based on the solution to the reporting problem still require $n^2/2$ steps – which is too long.

Another Computational Geometry problem related to intersection of rectangles is the *counting problem:* given n rectangles, count the total number of intersecting pairs. It is known that this problem can be solved in time $O(n \cdot \log(n))$, i.e., much faster than n^2. In other words, we can compute the total number of intersecting pairs reasonably fast. However, since we do not know which pairs intersect, it is not clear how to detect and delete duplicates if all we know is the total number of intersecting pairs.

In other words, to detect and delete duplicates in time $\ll n^2$, we cannot simply use known rectangle intersection algorithms from Computational Geometry, we must first modify these algorithms. This is what we will, in effect, do in this paper.

Before we explain how this is done, let us describe other possible connections to Computational Geometry problems and explain why neither of these connections immediately leads to a desired fast algorithm.

Range searching. Another related Computational Geometry problem is a *range searching* problem: given a rectangular range, find all the points within this range. The relation between this problem and the duplicate elimination is straightforward: a point (x_j, y_j) is a duplicate of another point (x_i, y_i) if the first point (x_j, y_j) belongs to the rectangular range $\mathcal{R}_i \stackrel{\text{def}}{=} [x_i - \varepsilon, x_i + \varepsilon] \times [y_i - \varepsilon, y_i + \varepsilon]$. In other words, duplicates of a point (x_i, y_i) are exactly the points that belong to the corresponding rectangular range $\mathcal{R}_i$. Thus, to find all the duplicates, it is sufficient to list all the points in all such ranges.

It is known (see, e.g., [8]) that, based on n data points, we can construct a special data structure called *layered range tree* in time $O(n \cdot \log(n))$, after which, for each rectangular range $\mathcal{R}_i$, we can list all the points from this range in time $O(\log(n) + k_i)$, where k_i is the total number of points in the range $\mathcal{R}_i$. If we repeat this procedure for all n points, then we get the list of all duplicates in time $O(n \cdot \log(n) + k)$, where $k = \sum k_i$ is the total number of pairs that are duplicates to each other. In other words, we get the same asymptotic time as for the rectangle intersection algorithm.

This coincidence is not accidental: it is known that one of the ways to get a list of intersecting pairs fast is by using range searching; see, e.g., [17].

Voronoi diagrams and nearest points. If each record could have only one duplicate, then we could find all the duplicates by finding, for each point, the nearest one, and checking how close they are. The fastest algorithms for finding the nearest point are based on first building a Voronoi diagram which takes time $O(n \cdot \log(n))$; after this, for each of n points, it takes time $O(\log(n))$ to find its nearest neighbor; see, e.g., [8, 17, 31]. Overall, it therefore takes time $O(n \cdot \log(n))$.

Alas, in reality, a point may have several duplicates, so after eliminating nearest duplicates, we will have to run this algorithm again and again until we eliminate them all. In the worst case, it takes at least n^2 steps.

Summary. To detect and delete duplicates in time $\ll n^2$, we cannot simply use known algorithms from Computational Geometry, we must first modify these algorithms. This is what we will do.

3 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 records, this takes too much time.

To speed up the 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: $a_1 \leq a_2 \leq \ldots \leq a_n$.

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 $a_l \leq a \leq a_u$. 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., [7]).

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 takes 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.

4 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 $y_i \leq y_j)$. 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.

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

Definitions: reminder. 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 $|x_i - x_j| \leq \varepsilon$ and $|y_i - y_j| \leq \varepsilon$.

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 one 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).

Motivations and description of the resulting algorithm. 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 $x_1 \leq x_2 \leq \ldots \leq x_n$. 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 $x_j \leq x_i + \varepsilon$. 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 $k \geq j$ and hence, none of the corresponding records r_k can be duplicates of r_i.

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

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

Algorithm 1.

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

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

How practical is this algorithm. For the gravity database, this algorithm works reasonably well. We have implemented it in Java, as part of our gravity data processing system, and it deleted duplicates from thousands of records in a few second, and from a few millions of records in a few minutes.

Limitations of the algorithm. Although this algorithm works well in most practical cases, 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.

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

To describe the main part – Part 2 – of this algorithm, we will use the same recursion that underlies mergesort:

Part 2 of Algorithm 2. To sort the (x-sorted) database $\langle r_1, \ldots, r_n\rangle$ by y and delete duplicates from the resulting double-sorted database, we do the following:

2.1. We apply the same Part 2 of Algorithm 2 to sort by y and 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 already sorted by y and free of duplicates, so we do not need to sort or delete anything);

2.2. We apply the same Part 2 of Algorithm 2 to sort by y and 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 already sorted by y and free of duplicates, so we do not need to sort or delete anything);

2.3. We merge the y-orders of the resulting duplicate-free subsets so that the merged database becomes sorted by y.

2.4. Finally, we clean the merged database to eliminate all possible duplicates introduced by merging.

Let us describe Step 2.3 (merging and sorting by y) and Step 2.4 (cleaning) in detail. Let us start with Step 2.3. We have two half-databases $\langle r_1, \ldots, r_{\lfloor n/2\rfloor}\rangle$ and $\langle r_{\lfloor n/2\rfloor+1}, \ldots, r_n\rangle$, each of which is already sorted by y. In other words:

- we have $\lfloor n/2\rfloor$ values $i'_1, \ldots, i'_{\lfloor n/2\rfloor} \in \{1, 2, \ldots, \lfloor n/2\rfloor\}$ such that $y_{i'_1} \le y_{i'_2} \le \ldots$, and
- we have $n - \lfloor n/2\rfloor$ values $i''_1, \ldots, i''_{n-\lfloor n/2\rfloor} \in \{\lfloor n/2\rfloor + 1, \ldots, n\}$ such that $y_{i''_1} \le y_{i''_2} \le \ldots$

The result $\langle r_1, \ldots, r_n\rangle$ of merging these two databases is already sorted by x; so to complete the merger, we must sort it by y as well, i.e., we must find the values $i_1, \ldots, i_n \in \{1, \ldots, n\}$ for which $y_{i_1} \le y_{i_2} \le \ldots \le y_{i_n}$.

In other words, we want to merge the y-orders of half-databases into a single y-order. This can be done similarly to the way mergesort merges the two orders of half-lists into a single oder, the only difference is that in mergesort, we actually move the elements around when we sort them, while here, we only move indices i_j but not the original records.

Specifically, we start with the two arrays $i'_1, i'_2, \ldots$ and $i''_1, i''_2, \ldots$ Based on these two arrays, we want to fill a new array $i_1, \ldots, i_n$. For each of these three index arrays, we set up a pointer. A pointer p' will be an index of an element of the array $i'_1, i'_2, \ldots$: if $p' = 1$, this means that we are currently considering the element i'_1 of this array; if $p' + 2$, this means that we are currently considering the element i'_2, etc. Once we have processed the last element $i'_{\lfloor n/2\rfloor}$ of this array, we move the pointer one more step to the right, and set p' to $i'_{\lfloor n/2\rfloor} + 1$. Similarly, a pointer p'' will be pointing to an element of the array $i''_1, i''_2, \ldots$, and a pointer p will be pointing to an element of the array $i_1, i_2, \ldots$

In the beginning, we have $p' = p'' = p = 1$. At each step, we do the following:

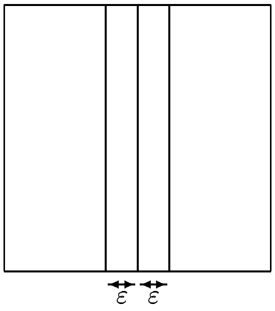
- If neither of the two arrays i' and i'' are exhausted, i.e., if both p' and p'' point to actual elements of these arrays, then we compare the corresponding y-values $y_{i'_{p'}}$ and $y_{i''_{p''}}$.
  - If $y_{i'_{p'}} \le y_{i''_{p''}}$, this means that the element $i'_{p'}$ is first in y-order, so we set $i_p := i'_{p'}$ and – since we have already placed the element $i'_{p'}$ – we move the pointer p' to its next position $p' := p' + 1$.
  - If $y_{i'_{p'}} > y_{i''_{p''}}$, this means that the element $i''_{p''}$ is first in y-order, so we set $i_p := i''_{p''}$ and – since we have already placed the element $i''_{p''}$ – we move the pointer p'' to its next position $p'' := p'' + 1$.

  In both cases, since we have filled the value i_p, we move the pointer p to the next position $p := p + 1$.
- If one of two arrays i' and i'' – e.g., i'' – is already exhausted, we simply copy the remaining elements of the non-exhausted array into the array i that we are filling; specifically, we take $i_p := i'_{p'}$, $i_{p+1} := i'_{p'+1}$, $\ldots$, until we exhaust both arrays.
- If both arrays i' and i'' are exhausted, we stop.

Let us now describe the cleaning Step 2.4. (This step is similar to divide-and-conquer algorithm for finding the closest pairs of points; see, e.g., [24].) How can we 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,

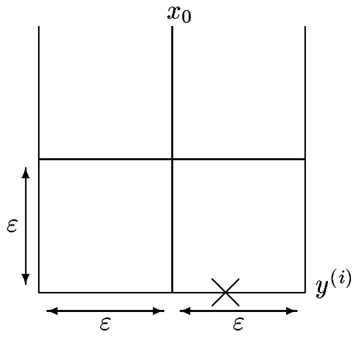
$$x_i \le x_0 \stackrel{\text{def}}{=} \frac{x_{\lfloor n/2 \rfloor} + x_{\lfloor n/2 \rfloor + 1}}{2},$$

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 $r_{(i+1)}, r_{(i+2)}, \ldots$, 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 "cleaning" algorithm:

Part 2.4 of Algorithm 2.

1. Select all the records r_i from both merged half-databases for which $x_i \in [x_0 - \varepsilon, x_0 + \varepsilon]$, where
$$x_i \le x_0 \stackrel{\rm def}{=} \frac{x_{\lfloor n/2 \rfloor} + x_{\lfloor n/2 \rfloor + 1}}{2}.$$
2. Since we have already sorted the values y_i, we can sort 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 performs in $O(n \cdot \log(n))$ steps in the worst case, and no algorithm with asymptotically smaller worst-case complexity is possible.*

Proof. Algorithm 2 consists of a sorting – that takes $O(n \cdot \log(n))$ 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, takes 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., [7]) that the main part takes $t(n) = O(n \cdot \log(n))$ 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 [7]. Proposition is proven.

Comment: how practical is this algorithm. It is well known that the fact that an algorithm is asymptotically optimal does not necessarily mean that it is good for reasonable values of n. To see how good our algorithm is, we implemented it in C, and tested it both on real data, with n in thousands, and on the artificial worst-case data when all the x-values are almost the same. In both cases, this algorithm performed well – ran a few seconds on a PC, and for the artificial worst case, it ran much faster than Algorithm 1.

Comment: an alternative)$(n \cdot \log(n))$ *algorithm.* As we have mentioned, the above Algorithm 2 is, in effect, a modification (and simplification) of the known algorithms from Computational Geometry.

It is worth mentioning that the above algorithm is not the only such modification: other $O(n \cdot \log(n))$ modifications are possible. For example, it is possible to use a range searching algorithm and still keep the computation time within $O(n \cdot \log(n))$.

To explain how we can do it let us recall that (see, e.g., [8]), based on n data points, we can arrange them into a layered range tree in time $O(n \cdot \log(n))$, after which, for each rectangular range $\mathcal{R}_i = [x_i - \varepsilon, x_i + \varepsilon] \times [y_i - \varepsilon, y_i + \varepsilon]$, we can list all the points from this range in time $O(\log(n) + k_i)$, where k_i is the total number of points in the range $\mathcal{R}_i$.

We have already mentioned that if we simply repeat this procedure for all n points, then, in the worst case, we will need $\sim n^2$ computational steps. To decrease the number of computational steps, we can do the following:

- we start with the record # $i = 1$, and use the range searching algorithm to find (and delete) all its duplicates;

- at each step, we take the first un-visited un-deleted record, and use the range searching algorithms find (and delete) all its duplicates;

- we stop when all the un-deleted records have already been visited.

How much time does this algorithm take?

- The original arrangement into a tree takes $O(n \cdot \log(n))$ steps.

- Each step of the iterative part takes $O(\log(n)) + k_i)$ steps. The overall sum of n (or fewer) $O(\log(n))$ parts is $O(n \cdot \log(n))$. As for $\sum k_i$, once a point is in the range, it is deleted as a duplicate; thus, the overall number $\sum k_i$ of such points cannot exceed the total number n of original points. Hence, the iterative part takes $O(n \cdot \log(n)) + O(n) = O(n \cdot \log(n))$ steps.

Thus, overall, this algorithm takes $O(n \cdot \log(n)) + O(n \cdot \log(n)) = O(n \cdot \log(n))$ steps – asymptotically the same as Algorithm 2.

This new algorithm takes fewer lines to explain, so why did not we use it? Well, it takes only a few lines to explain only because we relied on the range searching algorithm, and that algorithm actually requires quite a few pages to explain (see [8]). If we had to explain it from scratch (and program from scratch), it would take much longer than the simple algorithm described above – and our preliminary experiments showed that our Algorithm 2, while having the same asymptotics, is indeed much faster.

7 Deleting Duplicates Under Fuzzy Uncertainty

As we have mentioned, in some real-life situations, in addition to the threshold ε that guarantees that ε-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 corresponding 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., [3, 23, 29, 30].

8 The Problem of Deleting Duplicates: Multi-Dimensional Case

Formulation of the problem. 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 if we have multi-D records of the type $r = (x, \ldots, y, d)$, and we define $r_i = (x_i, \ldots, y_i, d_i)$ and $r_j = (x_j, \ldots, y_j, d_j)$ to be duplicates if $|x_i - x_j| \le \varepsilon$, $\ldots$, and $|y_i - y_j| \le \varepsilon$? For example, we may have measurements of geospatial data not only at different locations (x, y), but also at different depths z within each location.

Related problems of Computational Geometry: intersection of hyper-rectangles. Similar to the 2-D case, in m-dimensional case ($m > 2$), the problem of deleting duplicates is closely related to the problem of intersection of hyper-rectangles. Namely, if around each point $r_i = (x_i, \ldots, y_i, d_i)$, we build a hyper-rectangle

$$R_i = \left[x_i - \frac{\varepsilon}{2}, x_i + \frac{\varepsilon}{2}\right] \times \ldots \times \left[y_i - \frac{\varepsilon}{2}, y_i + \frac{\varepsilon}{2}\right],$$

then, as one can easily see, the points r_i and r_j are duplicates if and only if the corresponding hyper-rectangles R_i and R_j intersect.

In Computational Geometry, it is known that we can list all the intersecting pairs in time $O(n \cdot \log^{m-2}(n) + k)$ [5, 10, 11, 31]. It is also know how to solve the corresponding counting problem in time $O(n \cdot \log^{m-1}(n))$ [37, 31].

Related problems of Computational Geometry: range searching. Another related Computational Geometry problem is the range searching problem: given a hyper-rectangular range, find all the points within this range. The relation between this problem and the duplicate elimination is straightforward: a record $r_i = (x_i, \ldots, y_i, d_i)$ is a duplicate of another record $r_j = (x_j, \ldots, y_j, d_j)$ if the point point $(x_j, \ldots, y_j)$ belongs to the hyper-rectangular range

$$\mathcal{R}_i \stackrel{\text{def}}{=} [x_i - \varepsilon, x_i + \varepsilon] \times \ldots \times [y_i - \varepsilon, y_i + \varepsilon].$$

In other words, duplicates of a record r_i are exactly the points that belong to this hyper-rectangular range $\mathcal{R}_i$. Thus, to find all the duplicates, it is sufficient to list all the points in all such ranges.

It is known (see, e.g., [8]) that, based on n data points in m-dimensional space, we can construct a layered range tree in time $O(n \cdot \log^{m-1}(n))$; after this, for each hyper-rectangular range $\mathcal{R}_i$, we can list all the points from this range in time $O(\log^{m-1}(n) + k_i)$, where k_i is the total number of points in the range $\mathcal{R}_i$.

If we repeat this procedure for all n points, then we get the list of all duplicates in time $O(n \cdot \log^{m-1}(n) + k)$, where $k = \sum k_i$ is the total number of pairs that are duplicates to each other. In other words, we get an even worse asymptotic time than for the hyper-rectangle intersection algorithm.

If we use a speed-up trick that we explained in 2-dimensional case, then we can delete all the duplicates in time $O(n \cdot \log^{m-1}(n))$.

Related problems of Computational Geometry: Voronoi diagrams and nearest points. Even when each record has only one duplicate, and we can thus find them all by looking for the nearest neighbors of each point, we still need time $O(n^{\lceil m/2 \rceil})$ to build a Voronoi diagram [8, 17, 31]. Thus, even in this ideal case, the Voronoi diagram techniques would require much more time than search ranging.

What we will do. In this paper, we show that for all possible dimensions m, the duplicate elimination problem can be solved in the same time $O(n \cdot \log(n))$ as in the 2-dimensional case – much faster than for all known Computational Geometry algorithms.

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

Proof. This new algorithm starts with a database of records $r_i = (x_i, \ldots, y_i, d_i)$ and a number $\varepsilon > 0$.

Algorithm 3.

1. For each record, compute the indices $p_i = \lfloor x_i/\varepsilon \rfloor, \ldots, q_i = \lfloor y_i/\varepsilon \rfloor$.
2. Sort the records in lexicographic order $\leq$ by their index vector $\vec{p}_i = (p_i, \ldots, q_i)$. If several records have the same index vector, keep only one of these records and delete others as duplicates. As a result, we get an index-lexicographically ordered list of records: $r_{(1)} \leq \ldots \leq r_{(n_0)}$, where $n_0 \leq n$.
3. For i from 1 to n, we compare the record $r_{(i)}$ with its immediate neighbors; if one of the immediate neighbors is a duplicate to $r_{(i)}$, then we delete this neighbor.

Let us describe Part 3 in more detail. By an *immediate neighbor* to a record r_i with an index vector $(p_i, \ldots, q_i)$, we mean a record r_j for which the index vector $\vec{p}_j \neq \vec{p}_i$ has the following two properties:

- $\vec{p}_i \leq \vec{p}_j$, and
- for each index, $p_j \in \{p_i - 1, p_i, p_i + 1\}$, $\ldots$, and $q_j \in \{q_i - 1, q_i, q_i + 1\}$.

It is easy to check that if two records are duplicates, then indeed their indices can differ by no more than 1, i.e., the differences $\Delta p \overset{\text{def}}{=} p_j - p_i, \ldots, \Delta q \overset{\text{def}}{=} q_j - q_i$ between the indices can only take values -1, 0, and 1. To guarantee that $\vec{p}_j \geq \vec{p}_i$ in lexicographic order, we must make sure that the first non-zero term of the sequence $(\Delta p, \ldots, \Delta q)$ is 1.

Overall, there are 3^m sequences of -1, 0, and 1, where m denotes the dimension of the vector $(x, \ldots, y)$. Out of these vectors, one is $(0, \ldots, 0)$, and half of the rest – to be more precise, $N_m \overset{\text{def}}{=} (3^m - 1)/2$ of them – correspond to immediate neighbors.

To describe all immediate neighbors, during Step 3, for each i and for each of N_m difference vectors $\vec{d} = (\Delta p, \ldots, \Delta q)$, we keep the index $j(\vec{d}, i)$ of the first record $r_{(j)}$ for which $\vec{p}_{(j)} \geq \vec{p}_{(i)} + \vec{d}$ (here, $\geq$ means lexicographic order). Then:

- If $\vec{p}_{(j)} = \vec{p}_{(i)} + \vec{d}$, then the corresponding record $r_{(j)}$ is indeed an immediate neighbor of $r_{(i)}$, so must check whether it is a duplicate.

- If $\vec{p}_{(j)} > \vec{p}_{(i)} + \vec{d}$, then the corresponding record $r_{(j)}$ is not an immediate neighbor of $r_{(i)}$, so no duplicate check is needed.

We start with $j(\vec{d}, 0) = 1$ corresponding to $i = 0$. When we move from i-th iteration to the next $(i+1)$-th iteration, then, since the records $r_{(k)}$ are lexicographically ordered, for each of N_m vectors $\vec{d}$, we have $j(\vec{d}, i+1) \geq j(\vec{d}, i)$. Therefore, to find $j(\vec{d}, i+1)$, it is sufficient to start with $j(\vec{d}, i)$ and add 1 until we get the first record $r_{(j)}$ for which $\vec{p}_{(j)} \geq \vec{p}_{(i+1)} + \vec{d}$.

To complete the proof, we need to show that Algorithm 3 produces the results in time $O(n \cdot \log(n))$. Indeed, Algorithm 3 consists of a sorting – which takes $O(n \cdot \log(n))$ steps – and the main Part 3. During the Part 3, for each of N_m vectors $\vec{d}$, we move the corresponding index j one by one from 1 to $n_0 \leq n$; for each value of the index, we make one or two comparisons. Thus, for each vector $\vec{d}$, we need $O(n)$ comparisons.

For a given dimension m, there is a fixed number N_m of vectors $\vec{d}$, so we need the total of $N_m \cdot O(n) = O(n)$ computational steps. Thus, the total running time of Algorithm 3 is $O(n) + O(n \cdot \log(n)) = O(n \cdot \log(n))$. The proposition is proven.

Comment. Since our problem requires sorting, we cannot solve it faster than in $O(n \cdot \log(n))$ steps that are needed for sorting [7]. Thus, Algorithm 3 is asymptotically optimal.

9 Possibility of Parallelization

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

Proposition 3. *If we have at least $n^2/2$ processors, then, if we simply want to delete duplicates (and we do not want sorting), we can delete duplicates in a single step.*

Proof. 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 (r_i, r_j) $(i < j)$ turn out to be duplicates, delete one of them – the one with the largest number (i.e., r_j). Thus, we indeed delete all duplicates in a single step. The proposition is proven.

Comments.

- If we also want sorting, then we need to also spend time $O(\log(n))$ on sorting [21].
- 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 $O(\log(n))$ time.*

Proof. Let us show how Algorithm 3 can be implemented in parallel. Its first stage is sorting, and we have already mentioned that we can sort a list in parallel in time $O(\log(n))$.

Then, we assign a processor to each of n points. For each point, we find each of $N_m = (3^m - 1)/2$ indices by binary search (it takes $\log(n)$ time), and check whether the corresponding record is a duplicate.

As a result, with n processors, we get duplicate elimination in time $O(\log(n))$. The proposition is proven.

Proposition 5. *If we have $p < n$ processors, then we can delete duplicates in $O((n/p) \cdot \log(n) + \log(n))$ time.*

Proof. It is known that we can sort a list in parallel in time

$$O((n/p) \cdot \log(n) + \log(n));$$

see, e.g., [21].

Then, we divide n points between p processors, i.e., we assign, to each of p processors, n/p points. For each of these points, we check whether each of its N_m immediate neighbors is a duplicate – which takes $O(\log(n))$ time for each of these points. Thus, overall, checking for duplicates is done in time $(n/p) \cdot \log(n))$.

Hence, the overall time for this algorithm is

$$O((n/p) \cdot \log(n) + \log(n)) + O((n/p) \cdot \log(n)) = O((n/p) \cdot \log(n) + \log(n))$$

– the same as for sorting. The proposition is proven.

Comment: relation to Computational Geometry. Similarly to the sequential multi-dimensional case, we can solve the duplicate deletion problem much faster than a similar problem of listing all duplicate pairs (i.e., equivalently, all pairs of intersecting hyper-rectangles R_i). Indeed, according to [2, 17], even on the plane, such listing requires time $O(\log^2(n) + k)$.

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, by NSF grants CDA-9522207, EAR-0112968, EAR-0225670, and 9710940 Mexico/Conacyt, by Army Research Laboratories grant DATM-05-02-C-0046, by IEEE/ACM SC2002 Minority Serving Institutions Participation Grant, and by Hewlett-Packard equipment grants 89955.1 and 89955.2.

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

The authors are thankful to the anonymous referees for their help, and to Weldon A. Lodwick for his advise and encouragement.

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] Akl, S.G., and Lyons, K.A., 1993. Parallel Computational Geometry, Prentice Hall, Englewood Cliffs, New Jersey.

[3] Bojadziev, G., Bojadziev, M., 1995. Fuzzy Sets, Fuzzy Logic, Applications, World Scientific, Singapore.

[4] Campos, C., Keller, G.R., Kreinovich, V., Longpré, L., Modave, M., Starks, S.A., and Torres, R., 2003. The Use of Fuzzy Measures as a Data Fusion Tool in Geographic Information Systems: Case Study, Proceedings of the 22nd International Conference of the North American Fuzzy Information Processing Society NAFIPS'2003, Chicago, Illinois, July 24–26, 2003 (to appear).

[5] Chazelle, B.M., and Incerpi, J., 1983. Triangulating a polygon by divide-and-conquer, Proc. 21st Allerton Conference on Communications, Control, and Computation, 447–456.

[6] 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.

[7] 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.

[8] de Berg, M., van Kreveld, M., Overmars, M., and Schwarzkopf, O., 1997. Computational Geometry: Algorithms and Applications, Springer-Verlag, Berlin-Heidelberg.

[9] Edelsbrunner, E., 1980. Dynamic Data Structures for Orthogonal Intersection Queries, Report F59, Institute für Informationsverarbeitung, Technical University of Graz.

[10] Edelsbrunner, E., 1983. A new approach to rectangle intersections, Part II, Int'l Journal of Computer Mathematics 13, 221–229.

[11] Edelsbrunner, E., and Overmars, M.H., 1985. Batched dynamic solutions to decomposable searching problems, Journal of Algorithms 6, 515–542.

[12] 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

[13] 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.

[14] Fonte, C.C., and Lodwick, W.A., 2001. Modeling and Processing the Positional Uncertainty of Geographical Entities with Fuzzy Sets, Technical Report 176, Center for Computational Mathematics Reports, University of Colorado at Denver, August 2001.

[15] Fonte, C.C., and Lodwick, W.A., 2003. Areas of Fuzzy Geographical Entities, Technical Report 196, Center for Computational Mathematics Reports, University of Colorado at Denver, March 2003.

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

[17] Goodman, J.E., and O'Rourke, J., 1997. Handbook of Discrete and Computational Geometry, CRC Press, Boca Raton, Florida.

[18] 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.

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

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

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

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

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

[24] Laszlo, M.J., 1996. Computational Geometry and Computer Graphics in C++, Prentice Hall, Upper Saddle River, New Jersey.

[25] Lodwick, W.A., 1989. Developing Confidence Limits on Errors of Suitability Analyses in Geographic Information Systems. In: Goodchild, M., and Suchi, G. (Eds.), Accuracy of Spatial Databases, Taylor and Francis, London, 69–78.

[26] Lodwick, W.A., Munson, W., and Svoboda, L., 1990. Attribute Error and Sensitivity Analysis of Map Operations in Geographic Information Systems: Suitability Analysis, The International Journal of Geographic Information Systems 4(4), 413–428.

[27] Lodwick, W.A., Santos, J., 2003. Constructing consistent fuzzy surfaces from fuzzy data, Fuzzy Sets and Systems 135, 259–277.

[28] 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

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

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

[31] Preparata, F.P., and Shamos, M.I., 1989. Computational Geometry: An Introduction, Springer-Verlag, New York.

[32] 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.

[33] Santos, J., Lodwick, W.A., and Neumaier, A., 2002, A New Approach to Incorporate Uncertainty in Terrain Modeling. In: Egenhofer, M., and Mark, D. (eds.), GIScience 2002, Springer-Verlag Lecture Notes in Computer Science 2478, 291–299.

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

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

[36] 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.

[37] Six, H.W., and Wood, D, 1982. Counting and reporting intersections of *d*-ranges, IEEE Transactions on Computers C-31, 181–187.

[38] 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.