Watch Your Data Structures!

Dongliang Peng[1,2], Alexander Wolff[1]

[1]Institute of Computer Science, University of Würzburg,
Am Hubland, D-97074 Würzburg, Germany
Telephone: +49 (0) 931-31-85055
Fax: +49 (0) 931-31-84600
alexander.wolff@uni-wuerzburg.de
http://www1.informatik.uni-wuerzburg.de/en/staff

[2]Department of Geo-Informatics, Central South University, China

Abstract: When we plan to implement a program, there are always many data structures we can use to achieve a certain goal. If the data structures are not used carefully, however, inefficient programs may result. As an example, we consider the problem of searching pairs of close points from a dataset. We consider two points to be close if they lie within a square of pre-specified side length ε. We compare three obvious algorithms to solve the problem: a sweep-line (SL) algorithm, an algorithm based on the Delaunay Triangulation (DT) of the input points, and a hashing-like algorithm which overlays the input points with a rectangular grid. We implemented the algorithms in C# and tested them on randomly generated and real-world data. We used the DT available in ArcGIS Engine for the DT-based algorithm. We used three different *balanced binary search* tree data structures to implement the SL algorithm. However, the simple grid-based algorithm turned out to run faster than any of the other algorithms by a factor of at least 2.

Keywords: data structure, algorithms, sweep-line, Delaunay triangulation, grid, balanced binary search tree

1. Introduction

In map generalization, data integration, or data conflation, one often needs to detect pairs of points that are meant to be the same point but have slightly different coordinates due to rounding coordinates or data imprecision. We model this problem as follows. We want to find, in a given set of points, all pairs of *close* points. We consider two points to be close if they lie within a square of pre-specified side length ε. In other words, p and q are close if $L_\infty(p,q) = \max(\Delta x, \Delta y) \leq \varepsilon$, where $\Delta x = |p_x - q_x|$ and $\Delta y = |p_y - q_y|$.

When implementing and testing some ad-hoc solutions for this problem, we made a number of observations that we found worth sharing with the GIS community. In other words, the ultimate goal of this paper is not to identify the algorithm that performs best for the problem at hand. Instead, we want to address the issues we had during implementation and testing, and discuss the lessons we learned.

A brute-force approach for finding all pairs of close points requires $\Theta(n^2)$ time, where n is the number of points. This is worst-case optimal since the size of the output can be $\Theta(n^2)$ if ε is sufficiently large. Typically, however, the size of the output is small. Hence, it is desirable to use algorithms whose running time does not only depend on the size of the input (n, the

number of points), but also on the size of the output, that is, the number of close pairs, which we denote by k. Such an algorithm is called *output-sensitive*.

We consider three obvious output-sensitive algorithms: a sweep-line (SL) algorithm, an algorithm based on the Delaunay Triangulation (DT), and a hashing-like approach that uses a grid. The SL-algorithm runs in $O(k + n \log n)$ worst-case time. The same holds for the DT-based algorithm under the assumption that the input points are randomly and independently drawn from the unit square. Under the same assumption regarding the distribution of the input, the grid-based algorithm runs in $O(k + n)$ time. We sketch the three algorithms in Section 2. We have implemented them, and we have compared their performance on random data and real-world data; see Section 3. We conclude the paper in Section 4.

We remark that we focus on methods that can be implemented easily. For this reason we have not included a method that uses two-dimensional range trees, a two-level data structure based on BBST. The method works as follows. We insert all n input points into the range tree and then query the tree, for each point p, with a range of size $2\varepsilon \times 2\varepsilon$, centered at p. The running time of this method is $O(k + n \log^2 n)$, the memory consumption $O(n \log n)$ (De Berg, Cheong, Van Kreveld, & Overmars, 2008). The running time can be improved to $O(k + n \log n)$ by the use of fractional cascading (De Berg, Cheong, Van Kreveld, & Overmars, 2008), but this would mean additional implementational effort.

2. Algorithms

In the following, we sketch the three algorithms. We denote the set of input points by $P = \{p_1, p_2, \dots, p_n\}$; we denote the coordinates of point p_i by (x_i, y_i). While the algorithms work for any input, our running-time analyses will assume that the input points are uniformly and independently distributed (u.i.d.) in the unit square $[0,1] \times [0,1]$. We do not record the pairs of close points but just count them, thus we basically do not need any extra memory for the output.

2.1 The SL Algorithm

The SL paradigm is a common tool in computational geometry (De Berg, Cheong, Van Kreveld, & Overmars, 2008). Intuitively, a line sweeps the plane, stops at certain events and changes its internal status. One usually employs two data structures; the *event queue* and the *status*. For our problem, the search for close points, we sweep a horizontal line from top to bottom. Our sweep line $\ell: y = y_\ell$ stops at the y-coordinates $\{y_i, y_i - \varepsilon \mid i = 1, 2, \dots, n\}$, which are stored (together with references to the corresponding points) in the event queue in decreasing order. The status contains all points in a horizontal strip of height ε bounded by ℓ and $\ell_\varepsilon: y = y_\ell + \varepsilon$. We stores the points according to their x-coordinate using a balanced binary search tree (BBST). To implement the event queue, it suffices to store the $2n$ coordinates in an array and sort it.

We have only two types of events: *enter* and *leave*. A point p_i enters the status when the sweep line hits it, that is, when $y_\ell = y_i$. At the same time, we report each pair (p_i, p_j), where p_j is a point in the status with $x_i - \varepsilon \le x_j \le x_i + \varepsilon$. Such a *points-in-interval* query is supported by BBSTs; it takes $O(k_i + \log n)$ time, where k_i is the number of points that are reported for p_i. Then we add p_i to the status. The point p_i leaves the status when the sweep line reaches the y-coordinate $y_i - \varepsilon$. Summing up yields a total running time of $O(k +$

$n \log n)$, where k is the size of the output, that is, the number of close point pairs. The memory consumption of the SL algorithm is $O(n)$.

Unfortunately, the C# implementation of BBST based SortedDictionary does not offer a specific points-in-interval query. Instead, the interface offers the method *where*, which takes an arbitrary predicate as argument and returns all currently stored objects that fulfil the predicate – but this method takes linear time. This is not a problem as long as ε is so small that the height-ε strip above the sweep line never contains many points. In the worst case, however, the running time becomes quadratic even if there are no close pairs at all. Luckily, the BBST implementations SortedSet in .NET Framework 4.0 and TreeSet in the open-source C# data-structure library C5 (Kokholm & Sestoft, 2007) both support the points-in-interval query that we need (and so does, e.g., TreeSet in Java).

2.2 The DT-Based Algorithm

The DT is a useful tool to partition the plane such that spatially close points are connected. For example, it is well-known that the DT always connects a closest point pair. Given the DT, we go through the points and start a modified *breadth-first search* (BFS) from each of them. BFS is a well-known graph traversal algorithm (Cormen, Leiserson, Rivest, & Stein, 2009). For an input point p, our BFS considers every point $q \neq p$ with $L_\infty(p,q) \leq (1+\sqrt{2})\varepsilon/2$. We say that $r_1 = (1+\sqrt{2})\varepsilon/2 \approx 1.4\varepsilon$ is the *radius* of our BFS. The reason for using r_1 is simply that a radius of ε is not sufficient; we may, in rare cases, oversee some close pairs. In Figure 1, for an instance, if we set $\varepsilon = 0.003$, then p and q are a pair of close points. But we cannot find this pair if we only check points within a distance of ε, because there is at least one point lying outside the square in every path from p to q or from q to p. It is not hard to see that r_1 is necessary. Unfortunately, we could not prove that r_1 is sufficient. We conjecture that r_1 is indeed sufficient. This conjecture is supported by our experiments where we found all pairs of close points by using radius r_1.

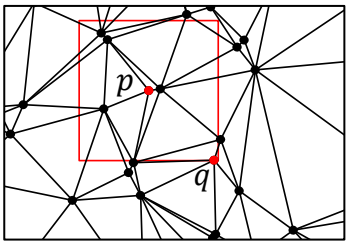


Figure 1. An instance of the DT. The line segments between the points are the edges of the DT. The side length of the square is $2\varepsilon = 0.006$; $x_p = 0.169134$, $y_p = 0.264491$, $x_q = 0.171957$, and $y_q = 0.261496$, thus $|\Delta x_{pq}| = 0.002823$ and $|\Delta y_{pq}| = 0.002995$.

Here, we show that a radius $r_2 = (1+\sqrt{7})\varepsilon/2 \approx 1.8\varepsilon$ is sufficient. According to Xia (2013), the DT contains, for any two input points p and q, a path of length less than $2|pq|$ connecting p and q. Observe that such a path is contained in the ellipse E_{pq} with foci p and q and major axis $2|pq|$. We are interested in the maximum x- or y-coordinate of E_{pq} for a fixed point p (say $p = (0,0)$) and any point q of L_∞-distance at most ε. One can show that the maximum x-coordinate of E_{pq} is maximized if $q = (\varepsilon, \varepsilon)$; see Figure 2 (a). In this case, the maximum x-coordinate of E_{pq} is $(1+\sqrt{7})\varepsilon/2$, which is the value we used for r_2. This can be seen by some elementary geometry; see Figure 2 (b). We move p to $(-1/2,0)$ and q to $(1/2,0)$.

Then E_{pq} is described by the equation $3x^2 + 4y^2 = 3$, and the right tangent of t with slope 1 has the equation $y = x - \sqrt{7}/2$. In the original coordinate system (Figure 2 (a)), this tangent corresponds to the vertical line $x = (1 + \sqrt{7})\varepsilon/2$.

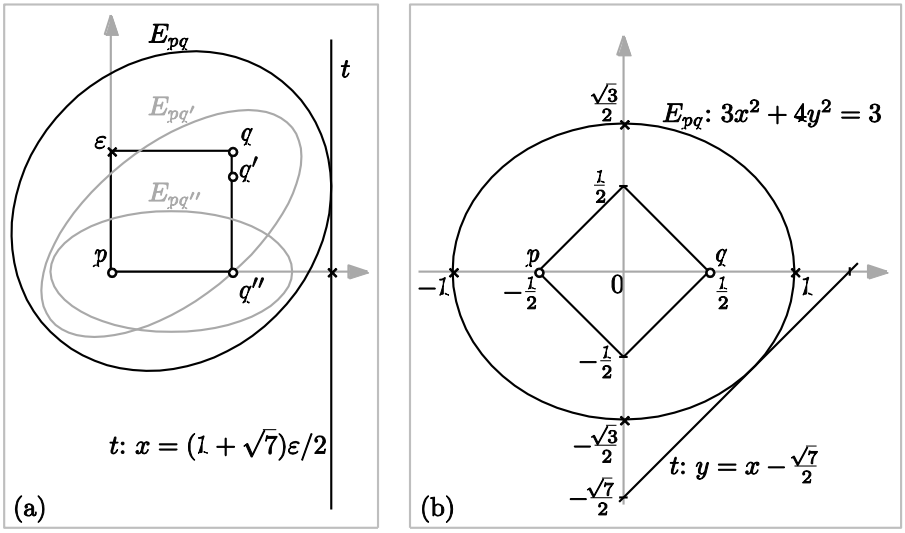


Figure 2. Among all points of L_∞-distance at most ε from p, the point $q = (\varepsilon, \varepsilon)$ gives rise to an ellipse E_{pq} whose right vertical tangent t has maximum x-coordinate (a). For computing the equation of t more easily, we transform p, q, E_{pq}, and t into the coordinate system (b).

Constructing the DT takes $O(n \log n)$ time and $O(n)$ memory (De Berg, Cheong, Van Kreveld, & Overmars, 2008). (Actually, under our assumption concerning the input distribution, the DT can be constructed in linear time (Buchin, 2009), but we will not exploit this here.) Assuming that the points are u.i.d. in the unit square, the running time of the DT-based algorithm is $O(k + n \log n)$. The memory consumption of the DT-based algorithm is $O(n)$.

2.3 The Grid-Based Algorithm

The third algorithm that we consider overlays the input points with a regular rectangular grid. It makes sense to set the side length σ of the grid cells to at least ε. Then, for each input point p, it suffices to compute the cell that contains p and to check the points in that cell and in the at most eight neighboring cells. To represent the grid, we use a two-dimensional array of size $(maxY - minY/\sigma) \times (maxX - minX/\sigma)$, where $maxX$ denotes the maximum x-coordinate among all the points; $minX$, $maxY$, and $minY$ are defined analogously to $maxX$. Each entry of the grid has a reference to a list (LinkedList in C#) that stores the points that lie in the corresponding cell. In order to ensure a memory consumption of $O(n)$, we set σ to $\max(\varepsilon, \sqrt{c/n})$, where c is a constant.

After we have computed the size of the grid cells, we go through the input points once to compute the two indices of the cell that contains each input point p. This corresponds to (i) dividing the coordinates of p by σ and (ii) applying the floor function. If we assume that the input is u.i.d. in the unit square, the expected number of point pairs we check in total is $O(\varepsilon^2 n^2) = O(k)$, and our algorithm runs in $O(k + n)$ time.

3. Case Study

We implemented the three algorithms in C# (using the .NET Framework 4.0). We ran our experiments under Windows 7 on a 3.3 GHz quad core CPU with 8 GB RAM. We measured time and memory consumption by using the built-in C# methods System.Environment.TickCount and GC.GetTotalMemory(true), respectively. For the DT, we took advantage of an implementation available in ArcGIS Engine 10.1. As we did not find a way to measure the memory consumption of the DT directly, we saved the files for the DT, i.e., files in .adf format from ArcGIS Engine 10.1 (an instance of the DT consists of 10 files), to the hard disk and measured the sum of the sizes of the 10 files. We show the results obtained by the DT-based algorithm with both radii r_1 and r_2; we use *DT total r1* and *DT total r2* to denote the respective total running times. We use *DT total* to denote the memory consumption of the DT-based algorithm and *DT constr.* to denote the time or memory consumption of the DT construction; these values are independent of the radius.

We tested the three algorithms on both random data and real-world data. There were ten sets of points for each type of data. We used N to denote the number of points in the set that had most points among the ten sets. We considered two different ways to set ε. One way was that we set ε to a certain value, say ε_0, independently of the instance size. This means that the size of the output, $k = \Theta(\varepsilon^2 n^2)$, grows quadratically. The other way was that we set $\varepsilon = \varepsilon_0\sqrt{N/n}$, which means that ε decreases from $\varepsilon_0\sqrt{N/n}$ to ε_0 and k grows linearly.

3.1 Case Study on Random Data

We randomly generated ten point sets u.i.d. in the unit square. The sizes of the point sets range from $20{,}000$ to $200{,}000$ with steps of size $20{,}000$. We set $\varepsilon_0 = 0.003$, and according to our description $N = 200{,}000$. We set the side length σ to $\varepsilon_0\sqrt{N/n}$ for the grid-based algorithm.

3.1.1 Time Consumption

In the experiment with $\varepsilon = \varepsilon_0$ (see Figure 3), the quadratic size of the output dominates the actual time consumption of the DT-based algorithm. The same holds for the C# SortedDictionary implementation of the SL algorithm. The corresponding C# SortedSet and C5 TreeSet implementations perform linearithmically, and the grid-based algorithm performs linearly: in these cases, the actual time consumption is dominated by the term that depends on the size n of the input.

In the experiment with $\varepsilon = \varepsilon_0\sqrt{N/n}$ (see Figure 4), the DT-based algorithm, however, now shows a (near-) linear time consumption. Still, it is much slower than the other four implementations. Interestingly, the C# SortedDictionary implementation still shows a quadratic behavior. This is due to the fact that the height-ε strip above the sweep line contains an expected linear number of points ($n\varepsilon$), which are traversed by the *where* method of the SortedDictionary data structure. The C# SortedSet, C5 TreeSet, and the grid-based implementations perform similarly as in the experiment with $\varepsilon = \varepsilon_0$. In both experiments, the simple grid-based algorithm is by far (by a factor of roughly 7) the fastest.

We also observe that in both experiments the time it took to only compute the DT was about the same as the running times of the two implementations of the SL algorithms. In addition, The DT-based algorithm with radius r_1 is faster than that with radius r_2 by a factor of 2.

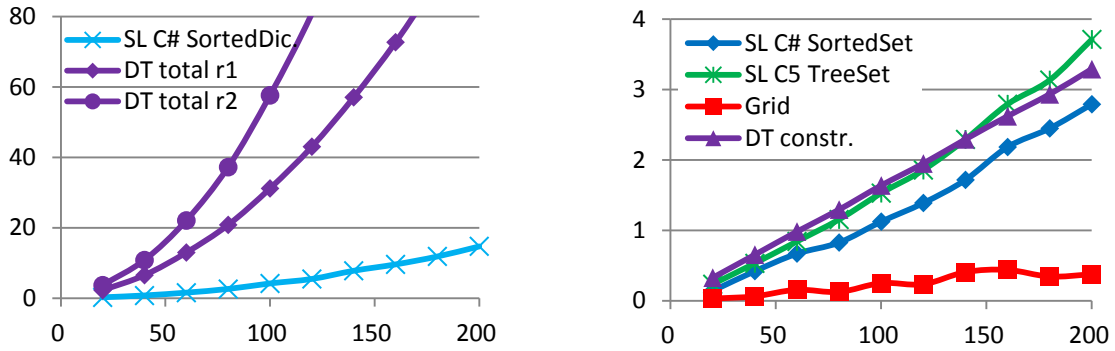


Figure 3. Time consumption of the algorithms for $\varepsilon = \varepsilon_0$. The DT-based algorithm took $109\ s$ with radius r_1 ("DT total r1") and $217\ s$ with radius r_2 ("DT total r2") for $n = 200,000$. In both graphs, the x-axis displays the size of the input ($n/1,000$) and the y-axis displays the time consumption in seconds.

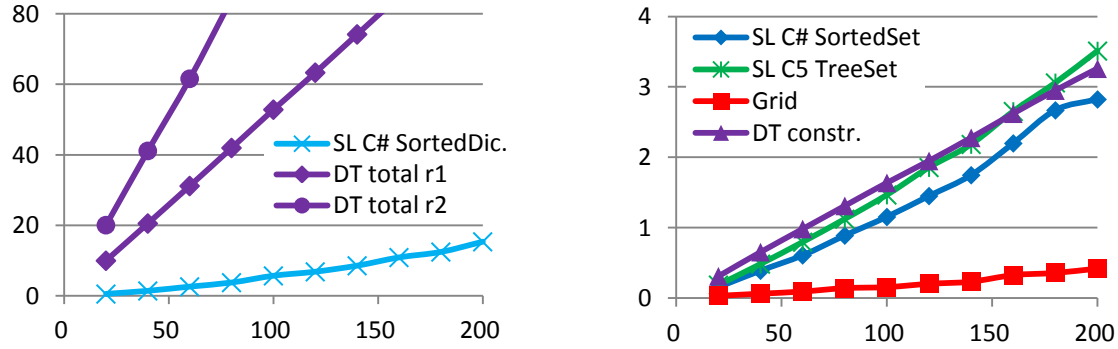
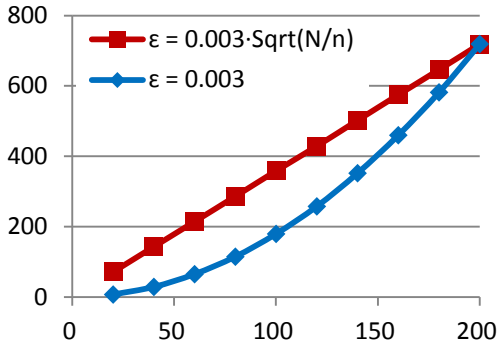


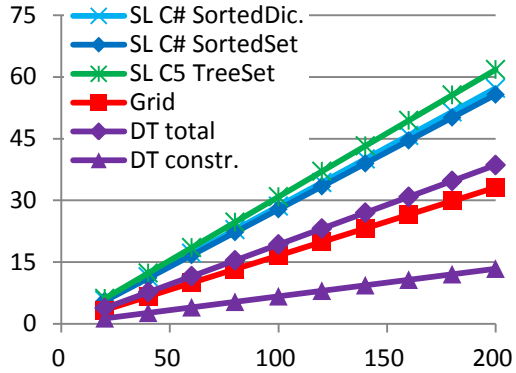
Figure 4. Time consumption of the algorithms for $\varepsilon = \varepsilon_0\sqrt{N/n}$. The DT-based algorithm took $106\ s$ with radius r_1 ("DT total r1") and $216\ s$ with radius r_2 ("DT total r2") for $n = 200,000$. The axes and the notations are as in Figure 3.

3.1.2 Output Size and Memory Consumption

The curves of the output size perform as expected. The output size grows quadratically when we set $\varepsilon = \varepsilon_0$, and grows linearly when we set $\varepsilon = \varepsilon_0\sqrt{N/n}$; see Figure 5(a). As said before, we did not record the pairs of close points but just counted the number of pairs, we basically did not need any extra memory for the output. Therefore, the two experiments with different values of ε need the same amount of memory. Figure 5(b) shows that the memory consumption of all our methods grows linearly. Among the five implementations, the grid-based algorithm uses the least amount of memory, which is less than the DT-based algorithm by a factor of 1.2. We can also see that the C# SortedSet BBST needs the least memory to implement the SL algorithm; about 10% less than the C5 TreeSet implementation.



(a) Output size: the y-axis displays $m/1{,}000$, where m is the number of pairs of close points.



(b) Memory consumption: the y-axis displays the memory consumption in MB.

Figure 5. Output size and memory consumption of the algorithms. The x-axis displays the size of the input ($n/1{,}000$). Sqrt means square root.

3.2 Case Study on Real-World Data

We use a set P of 277,034 points from OpenStreetMap that represent bus stops, milestones, hotels, post boxes etc. in the state of Bavaria, Germany; see Figure 6. After deleting duplicates, we had $n = 276{,}992$ points left. We computed the *average distance* as

$$AvgDis = \sqrt{(maxX - minX) \cdot (maxY - minY)/n}$$

where $maxX$, $minX$, $maxY$, and $minY$ are defined as in Section 2.3.

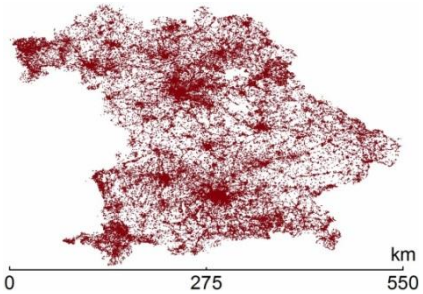


Figure 6. The point data of Bavaria.

We perturbed the points according to a Gaussian distribution. For each point, we generated a pair of normally distributed numbers X and Y by the Box-Muller method (Box & Muller, 1958). Then we set the new coordinates as

$$x' = x + \delta \cdot X$$
$$y' = y + \delta \cdot Y$$

where x and y are the original coordinates, and we use the standard deviation $\delta = AvgDis/6$. After perturbing, we had two points sets, i.e., the original set P and a perturbed set P'. This models the problem that we have two point sets from different sources, and we try to find the corresponding points. In order to extract from P ten data sets $P_1, \ldots, P_{10}$ of different sizes, we selected for P_i each point in P with probability $i/10$. Hence, $|P_i| \approx |P| \cdot i/10$ and $|P_{10}| = |P|$.

For $i = 1, \ldots, 10$, let $\bar{P}_i = P_i \cup P_i'$ where P_i' is the set of perturbed points corresponding to the points in P_i. These are the sets we used in our experiments; see Figure 7–9.

We set $\varepsilon_0 = \delta$, and according to our description $N = 2 \cdot 276{,}992 = 553{,}984$. For the grid-based algorithm, setting σ to $\varepsilon_0\sqrt{N/n}$ would yield too many grid cells (18 times the number of points). This would occupy a lot of memory and take a lot of time to initialize the LinkedList entries. Instead, we set σ to $\frac{AvgDis}{\sqrt{2}}\sqrt{N/n}$, which means that we have roughly the same numbers of grid cells and points.

3.2.1 Time Consumption

Basically, we get similar results as in Section 3.1.1. An interesting difference is that although the grid-based algorithm is still the fastest, the factor decreases to roughly 2. There are two reasons. One is that the ratio of σ to ε changes. When $n = N$, it is $3\sqrt{2}$ for the case study on real-world data while it is 1 for the case study on random data. This leads the grid-based algorithm to check more points in the case study on real-world data. The other reason is that the size of the real-world output dominates the running time a little bit more. There are on average 10.8 close points for one point in the case study on real-world data when $N = 553{,}984$ and $\varepsilon = \delta$, while the number is 7.2 for the case study on random data when $N = 200{,}000$ and $\varepsilon = 0.003$. Also note that now the construction time of the DT is less than the running time of the two implementations of the SL algorithm. The DT-based algorithm with radius r_1 is faster than that with radius r_2 by a factor of roughly 3.

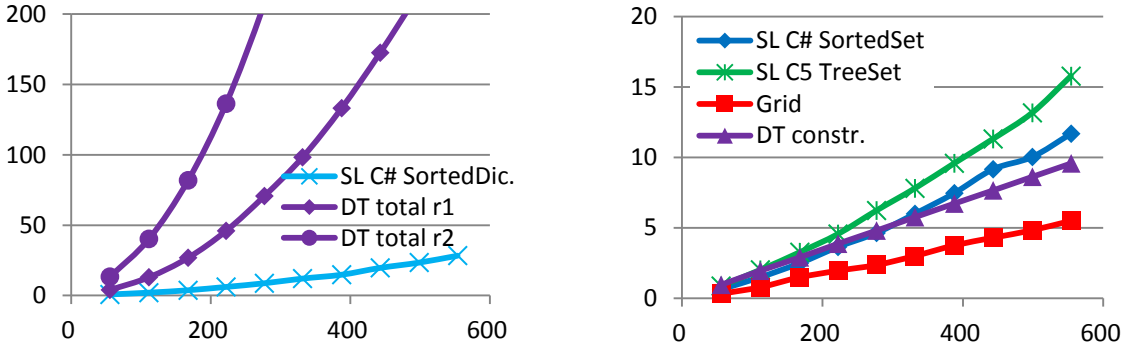


Figure 7. Time consumption of the algorithms for $\varepsilon = \delta$. The DT-based algorithm took $262\ s$ with radius r_1 ("DT total r1") and $784\ s$ with radius r_2 ("DT total r2") for $n = 553{,}984$. The axes and the notations are as in Figure 3.

3.2.2 Output Size and Memory Consumption

Also, the curves of the output size perform as expected; see Figure 9(a). For the grid-based algorithm, when we try to promise that there are roughly the same numbers of entries and points, we need more memory compared to the case study on random data. However, the grid-based algorithm still uses less memory than the DT-based algorithm by a factor of 1.1, and it also still uses less than the SL implementations by a factor of 1.6.

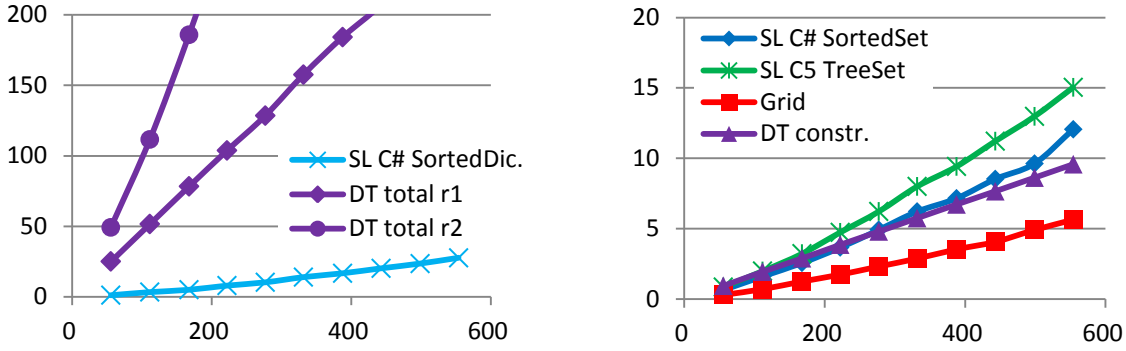
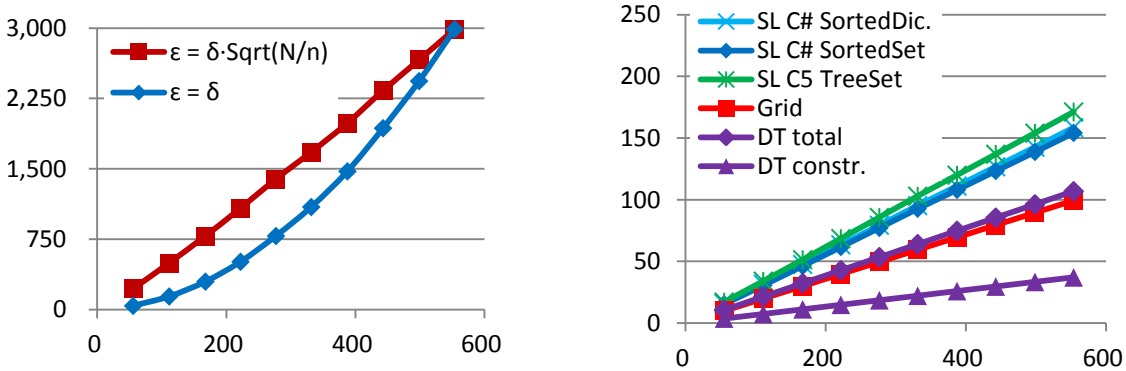


Figure 8. Time consumption of the algorithms for $\varepsilon = \delta\sqrt{N/n}$. The DT-based algorithm took $267\ s$ with radius r_1 ("DT total r1") and $810\ s$ with radius r_2 ("DT total r2") for $n = 553,984$. The axes and the notations are as in Figure 3.



(a) Output size. (b) Memory consumption.

Figure 9. Output size and memory consumption of the algorithms. The axes and the notations are as in Figure 5.

4. Concluding Remarks

Although the grid-based algorithm was the clear winner of our comparison, we were more interested in the results of the three implementations of the SL algorithm. The SL paradigm can be used to solve many problems (e.g., computing the Voronoi diagram) for which the grid approach would not work. When implementing the SL algorithm, it was tempting to use the data structures available in C# (for example, the method *where* of SortedDictionary), but we have seen that it is worth to read the fine print.

Even from the slowest algorithm, based on the DT, we have learned something. By comparison with the other implementations, we noticed that the radius-ε BFS missed a few close pairs in the case study on random data (just 5 out of the 718,775 close pairs that were reported in the 200,000-point instance for $\varepsilon = 0.003$). Then we conjectured that a radius of $(1+\sqrt{2})\varepsilon/2$ is sufficient, which was supported by our experiments where we found all pairs of close points. We also proved that a radius of $(1+\sqrt{7})\varepsilon/2$ is sufficient. Of course, enlarging the radius slowed down our code. It turned out, however, that a radius of $(1+\sqrt{2})\varepsilon/2$ is sometimes necessary.

5. Acknowledgments

We thank Thomas van Dijk for pointing us to the grid-based algorithm. This research was partly supported by the China Scholarship Council (CSC).

6. References

BOX, G., & MULLER, M. (1958). A note on the generation of random normal deviates. *The Annals of Mathematical Statistics, 29*(2), 610-611.

BUCHIN, K. (2009). Delaunay triangulations along space-filling curves. In A. Fiat, & P. Sanders (Ed.), *Proc. 17th Annu. Europ. Symp. Algorithms (ESA'09). 5757*, pp. 119-130. Springer.

CORMEN, T. H., LEISERSON, C. E., RIVEST, R. L., & STEIN, C. (2009). *An introduction to algorithms* (Third edition ed.). MIT Press.

DE BERG, M., CHEONG, O., VAN KREVELD, M., & OVERMARS, M. (2008). *Computational geometry: algorithms and applications* (Third edition ed.). Berlin Heidelberg: Springer-Verlag.

KOKHOLM, N., & SESTOFT, P. (2007). The C5 generic collection library. A .NET 2.0 collection library that supports advanced functionality. *Dr. Dobb's Journal*, 50-56.

XIA, G. (2013). The stretch factor of the delaunay triangulation is less than 1.998. *SIAM Journal on Computing, 42*(4), 1659-1620.

Biography

Dongliang Peng MEng *received a bachelor degree in Mapping Engineering and a master degree in Cartography and Geographic Information Engineering from Central South University, respectively in 2009 and 2012. He is currently a PhD student in the Institute of Computer Science, University of Würzburg. His research focuses on morphing algorithms for maps, cartographic generalization, and algorithms for GIS.*

Prof. Dr. Alexander Wolff *received a PhD degree in Computer Science from Freie Universität Berlin in 1999. He did his habilitation at the Faculty of Informatics of Karlsruhe University (now KIT). He is currently the vice dean of the Faculty for Mathematics and Computer Science, University of Würzburg. His research focuses on graph drawing, computational geometry, geometric networks, algorithms for GIS, and graph algorithms.*