Separation of Voronoi Areas

Martin Gebert, Sascha Schreckenbach, Jonathan Sharyari

8th February 2013

Abstract

Given a set $\mathcal{R}$ of N red points, we consider a set $\mathcal{B}$ of blue points to be a *solution* if in the Voronoi diagram for $\mathcal{R} \cup \mathcal{B}$, no two red points have coinciding delimiters. In this paper we experimentally investigate the claim that an minimal solution always needs as many blue points as there are red points. Although this claim has not been proven, we have found no counter-examples to this claim, further strengthening our belief that the claim is in fact true.

Contents

1 Introduction

1.1 Voronoi Diagram

Given a set of points in space, a Voronoi diagram is a partitioning of the space into a set of Voronoi regions or *cells*. The Voronoi region corresponding to a point p consists of all points that are closer to p than to any other point in space.

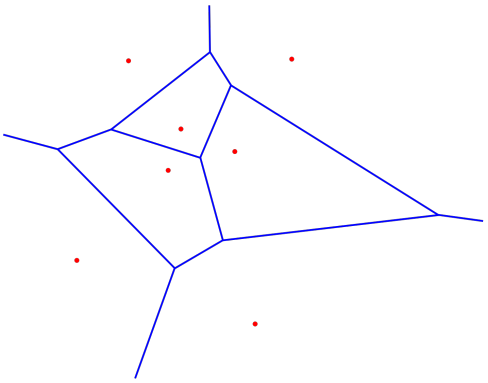

Figure 1: Example of a Voronoi diagram.

1.2 Delaunay triangulation

A Delaunay triangulation for a set of points in space is a triangulation of the points, with the added property that no point is inside the circumcircle of any triangle in the triangulation, illustrated in figure 2. The Delaunay triangulation is dual to the Voronoi diagram, illustrated in figure 3.

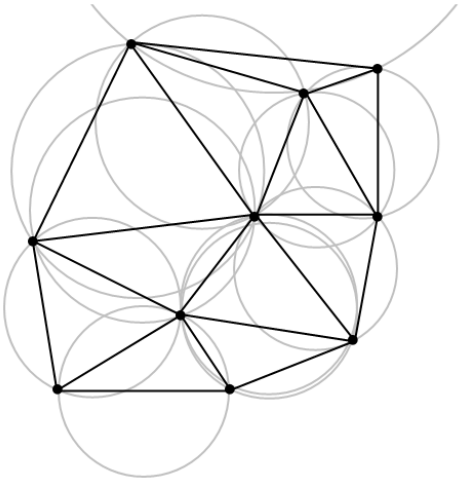

Figure 2: A Delaunay triangulation, with a set of corresponding Delaunay circles.

1.3 Arrangement

Given a set of planar curves, the arrangement is the subdivision of the plane into zero-dimensional, one-dimensional and two-dimensional cells, called *vertices, edges* and *faces* respectively, induced by the given curves. 1

1.4 The Set Cover Problem

Given a set $\mathcal{A}$ of sets of numbers, the set cover problem is the problem of finding a minimal subset $\mathcal{B}$ of $\mathcal{A}$, so that the union of the sets in $\mathcal{B}$ is equal to the union of all elements of $\mathcal{A}$.

Example

Let $\mathcal{A} = \{\{1,2\}, \{2,3\}, \{2,5\}, \{3,5\}\}$. The union of all sets of $\mathcal{A}$ is $\{1,2,3,5\}$.
The subset $\mathcal{B}_1 = \{\{1,2\}, \{2,3\}\}$ has the union $\{1,2,3\}$ and does not cover the set $\mathcal{A}$.
The subset $\mathcal{B}_2 = \{\{1,2\}, \{2,3\}, \{2,5\}\}$ has the union $\{1,2,3,5\}$ that covers $\mathcal{A}$, but it is not minimal.
A minimal solution is $\mathcal{B}_3 = \{\{1,2\}, \{3,5\}\}$.
The set cover problem is known to be NP-complete.4

2 Problem formulation

Given a set $\mathcal{R}$ of red points, find a set $\mathcal{B}$ of blue points such that in the Voronoi diagram of $\mathcal{R} \cup \mathcal{B}$, no two regions corresponding to points in $\mathcal{R}$ are incident to each other. For the Delaunay triangulation of $\mathcal{R} \cup \mathcal{B}$, this means that there is no edge connecting two red points. The conjecture that $|\mathcal{B}| = |\mathcal{R}|$ is sufficient in an optimal solution is to be experimentally explored.

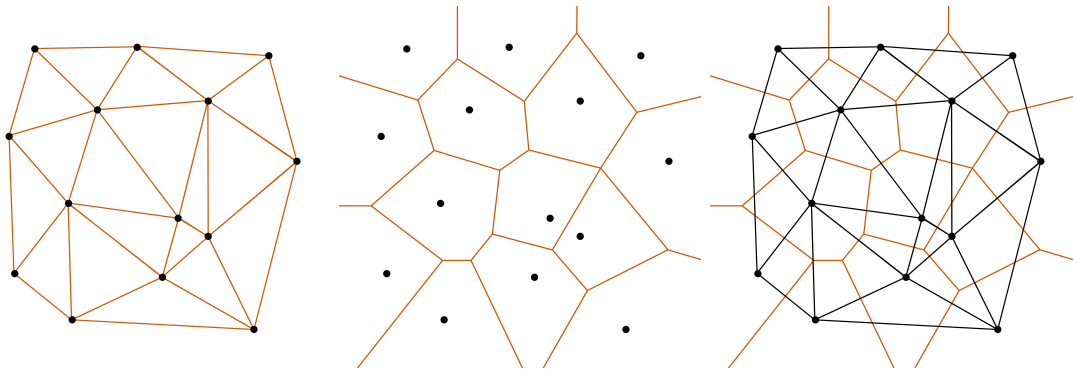

Figure 3: A triangulation of a set of points (left), the corresponding Voronoi diagram (middle) and the overlapping of the triangulation and the Voronoi diagram, emphasizing the dual relationship (right).

3 Background

3.1 Upper and lower bounds

Given a set $\mathcal{R}$ of red points with $N = |\mathcal{R}|$, it has been shown 2 that at least N-1 points are needed to solve this problem. The case where only N-1 blue points are needed is exemplified in the picture below, and always applies to the special case when all points lie in row. In the general case, it has been shown that for $N > 2$ there are cases where at least N blue points are required to solve the problem.

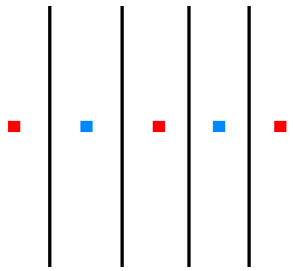

Figure 4: Example of points not in general position, having a solution with N-1 blue points.

In the general case, it has been proven 2 that at most $3N/2$ blue points are needed to solve the problem, and in the case where the points in $\mathcal{R}$ are in convex position, $5N/2$ points are sufficient. It has been conjectured that N points are sufficient to solve the problem in the general case.

3.2 Complexity

Although not yet shown, the problem is suspected to be NP-hard 2, as a similar problem has been proven to be NP-hard. 3

4 Algorithms

4.1 General outline

1. For the set $\mathcal{P}$ of points (both red and blue) find the Delaunay triangulation and the Voronoi diagram.

2. For each edge in the Delaunay triangulation that connects two red points, find a circle with these two points on its perimeter and the circle's centre on the corresponding Voronoi edge and add these to the (initially empty) set of circles $\mathcal{C}$.

3. Calculate the arrangement $\mathcal{A}$ of the circles $\mathcal{C}$.

4. Find a set of points $\mathcal{P}^*$, so that for every face in the arrangement $\mathcal{A}$, exactly one point is located in the interior. For every point $\rho \in \mathcal{P}^*$ and every circle ς in $\mathcal{C}$, determine whether ρ is in the interior of ς.

5. Find a minimum subset $\mathcal{P}^{**}$ of $\mathcal{P}^{*}$, so that every circle in $\mathcal{C}$ is covered by at least one point.

6. Set $\mathcal{P}$ to $\mathcal{P}^{**} \cup Red(\mathcal{P})$, where $Red(\mathcal{P})$ denotes the set of red points in $\mathcal{P}$. Calculate the Delaunay triangulation for $\mathcal{P}$.

7. For every red edge still in the triangulation, calculate the length of its dual in the Voronoi graph. If no red edges exist in the triangulation, terminate. If the sum of the calculated lengths is sufficiently small (smaller than 300*N), use random search to find an optimal solution. Otherwise, restart from 2.

4.2 Finding circles corresponding to unsatisfied edges

Following the terminology used by the CGAL-project, a **line** is a line unbounded on both sides, whereas a line that is bounded on one side is called a **ray** and when bounded on both sides it is called a **segment**.
The approach for choosing a circle depends on which type of Voronoi edge is to be blocked. Given a Voronoi edge corresponding to the neighbouring points p_1 and p_2, there are the following cases:

1. For a **line**, the smallest possible circle is chosen. This is the circle with its centre in the middle of p_1 and p_2, and a radius such that p_1 and p_2 are on the perimeter of the circle. A line only occurs in the Voronoi diagram in the trivial case where all points lie in a straight line.

2. A **ray** in the Voronoi diagram is always associated with two points on the convex hull of the Delaunay triangulation. There are arbitrarily large circles with the points p_1 and p_2 on their perimeter and their centre on the ray. In this case, one of these circles is randomly chosen.

3. For a **segment**, the smallest circle with p_1 and p_2 on its perimeter, and its centre on the midpoint of the segment is chosen.

4.3 Finding a point in the interior of a face

A relatively difficult task is that of finding a point in the interior of an arbitrary face, given two point p_1 and p_2 known to be on the boundary of the face. This can be solved using the following algorithm, which is exemplified in figure 5.

1. Find the point p_m in the middle of p_1 and p_2.

2. Find the line l that is going through p_m and is perpendicular to the edge between p_1 and p_2.

3. Find the segment s that is the intersection of l and the smallest rectangle large enough to contain all circles of the arrangement. This step is needed because the the type of CGAL-arrangement used does not support lines and is described here only for the sake of completeness.

4. Find the segment that is the intersection of the segment s and the face itself.

5. Any point except the endpoints of this segment are inside the face. The midpoint of the segment is chosen.

4.4 Finding a minimal subset of points

The problem of finding a minimal subset P of a set of points $\mathcal{P}^*$, so that at least one point in $\mathcal{P}$ is in the interior of each circle in $\mathcal{C}$, is exactly that of the set covering problem. The problem is easily stated in terms of an integer programming problem:

For every point $\rho_i \in \mathcal{P}^*$, let $x(\rho_i)$ be a boolean, set to 1 if the point $\rho_i \in P^*$ and 0 otherwise. Also, for every circle $\varsigma_j \in \mathcal{C}$, let $\varsigma_j(\rho_i)$ be 1 if point ρ_i lies in the interior of ς_j and 0 otherwise.

1 Minimize: $\sum_{\rho \in \mathcal{P}^*} x(\rho_i)$

2 Constraint: $\sum_{\rho \in \mathcal{P}^*} \varsigma_j(\rho_i) \geq 1$ **for** each $\varsigma \in \mathcal{C}$

Stated in this way, the problem can be solved relatively efficiently with an integer programming solver, such as Gurobi.

4.5 Refining a near-optimum solution through random search

Random search is a direct search method that does not require the derivative of the function to be minimized. It is algorithmically simple, and in general terms work as follows:

To minimize a function $f : \mathbb{R}^n \to \mathbb{R}$, start with a (potentially random) solution $x : \mathbb{R}^n$ and do the following:

```
1  while (f(x) > 0 )
2    Generate a new candidate solution y in the neighbourhood of x
3    if f(y) < f(x)
```

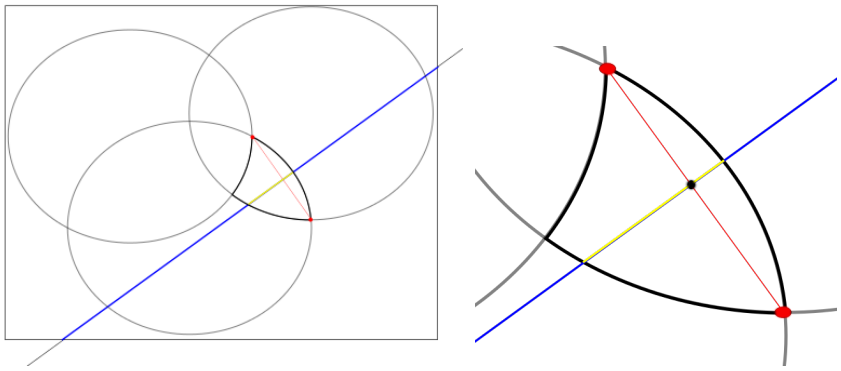

Figure 5: The graphic shows the application of the algorithm descibed in this section to an exemplary face. The black point in the right picture is the point p_m in step 1. The blue line segment (right and left) is that mentioned in step 3 and the yellow segment (right) is the intersection mentioned in step 4.

4	replace the current solution x with y.

There are several candidates for choosing the objective function f, for example the number of red Voronoi edges. For this problem, we have chosen the sum of the squared lengths of all red voronoi edges.

5 Results

5.1 Graphical interface

In order to easily specify new problems, and to visualize the process of solving a problem, a minimalistic graphical interface was implemented. It has methods for adding, moving and removing points. It automatically updates and draws the Voronoi diagram corresponding to the set of points used in the current iteration of the above mentioned algorithm, and also the a set of Delaunay circles corresponding to the red edges in this Voronoi diagram.

Additionally, methods for saving and loading configurations were added.

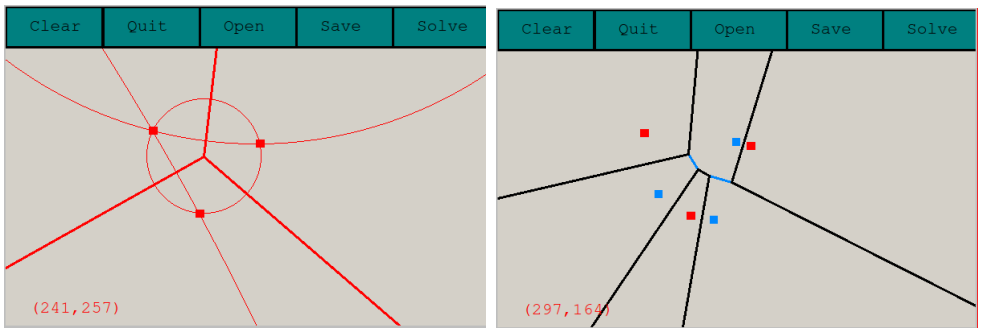

Figure 6: An example of a problem configuration (left) and a found solution to the same problem (right).

5.2 Command line parameters

It's also possible to start the program without using the GUI. To do so, either the name of an existing file or "random" followed by an integer N has to be provided as command line parameter. If the first command line parameter is "random", a problem consisting of N randomly placed points is generated and solved. Afterwards, a result will be printed as a series of comma-separated numbers with the pattern: the number of red points, the number of blue points used to solve the generated problem, the time in ms to find the solution, the number of iterations of phase 1 (algorithm 4.1) and the number of iterations of phase 2 (randomized search).

If a filename is provided, the problem saved in the named file is solved (without GUI) and the same information as above is printed.

Example

```
 user@computer: /path/$ ./separator random 6
6, 6, 2674, 7, 9355
```

```
user@computer: /path/$
```

This output shows that six blue points were used to solve the problem. To do so, phase 1 iterated 7 times and phase 2 iterated 9355 times. It took 2,67 seconds to find the solution.

5.3 Automated testing and debugging

The GUI-less mode was designed to be used for automatation, such as the provided by the `randomtest` and `reruntest` scripts.

The `randomtest` script will randomized problems of a specified size, a certain number of times (this is specified in the script). If a problem could not be solved within 5 minutes, it is considered to have *timed out*, and is stored in the subfolder *testing*.

The script `reruntest` will run all tests in the *testing* subfolder again, and remove them after completion. If a test again times out, a new file will be created in the same folder. This is to ensure that a correct solution is found for all problems.

When running with GUI enabled, the last used configuration is always saved in the file *latest.cnfg*. This means that in case of unexpected behavior, for example a crash, the configuration on which it occured can always be recovered and examined.

5.4 Speed

The problem of finding a blocking set of blue points is computationally hard since the set-cover problem is NP-hard, but also by its own, assuming the conjuncture in 2 is true. This complexity is reflected in the time it takes to solve a problem, shown in figure 4.

The figure shows the average and median time of solving a problem with between 5 to 13 points. Due to the existence of local minima in the search space explored by the random search algorithm, there is a risk that a run takes unusually long time. Such statistical *outliers* can greatly impact the total average, but do not impact the median value. Thus both the average and median execution time is shown in the graph below.

The graph is based on 120 randomly generated problems for each problem size, excluding the runs that ended with a timeout. The execution time was measured on a machine with an Intel Core 2 Duo 2.00GHz CPU and with 2GB of RAM storage.

5.5 Timeout and non-optimal solutions

During solving, there is a risk of a timeout occuring or that a solution is found that is believed not to be optimal (more blue points than red points). In the statistics which was shown above, runs that ended with a timeout have been excluded as the impact on time is too large (a timeout occurs after 300 seconds). The total number of timeouts and non-optimal solutions are shown below in figure 8.

It can be seen that the number of non-optimal solutions is either relatively constant or slowly increasing. The phenomenon is uncommon (not more than 6% in the statistics above), thus a solution to the problem is to simply re-solve the problem automatically, if a non-optimal solution is detected. This is indeed the approach generally used in this project, although this functionality was disabled in order to measure the frequency of the problem.

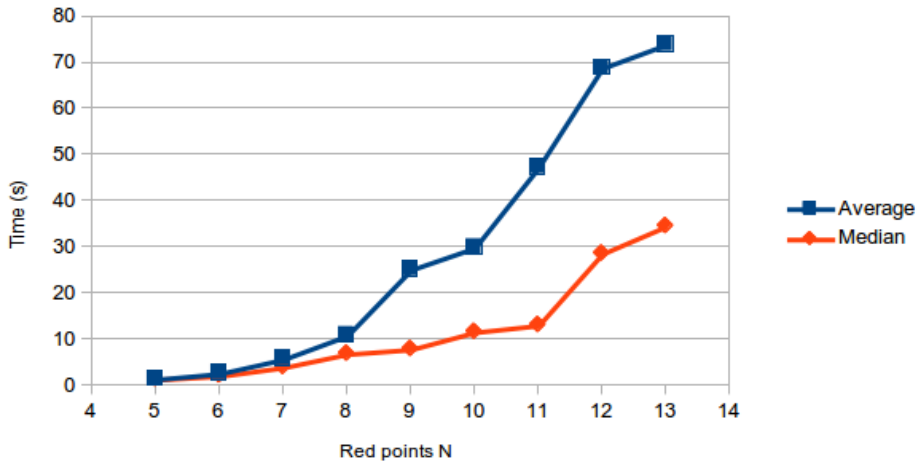

Figure 7: Graph showing average and the median time to find a solution, for problems of different size. The shown values are based on 120 runs per problem size.

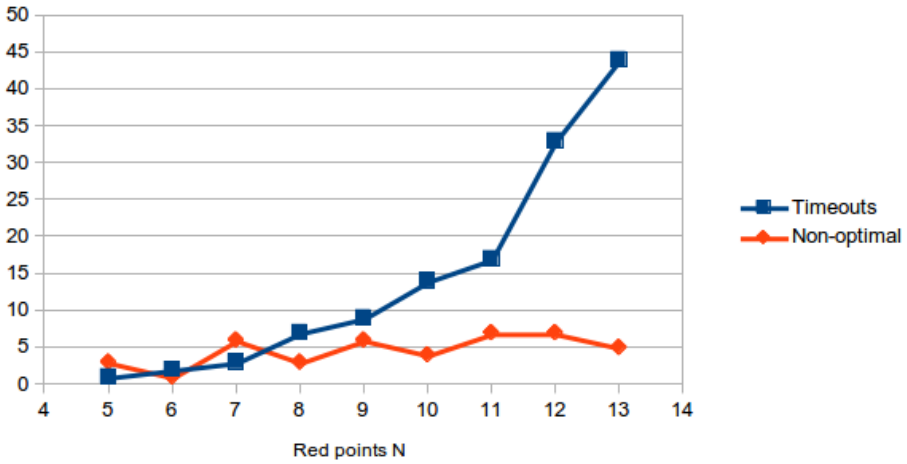

Figure 8: Graph showing number of timeouts and non-optimal solutions. The result is based on 120 runs per problem size.

The number of timeouts on the other hand seems to be increasing exponentially. A timeout can occur in two different ways; either the algorithm gets stuck in the first phase (the algorithm described in section 4.1), or the second phase (randomized search). To get some insight in this matter, the number of iterations in phase 1 and phase 2 where measured. See figure 9 and 10.

One important conclusion is that the average number of iterations in phase 1 is increasing very slowly. This means that relatively few iterations are needed until a near-optimal solution is found, even for quite difficult problems. This low number indicates that phase

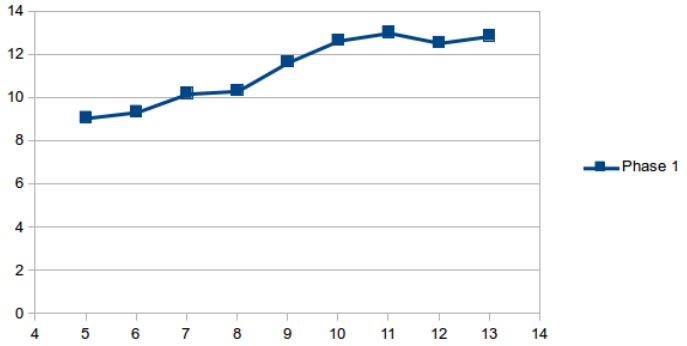

Figure 9: The average number of iterations in phase 1.

1 is not the cause of timeouts, and in fact, only one (1) timeout during testing occurred because of the algorithm being stuck in the first phase (the problem had 9 points, 51 iterations in phase 1 and never entered phase 2). This leads us to believe that timeouts generally occur in phase 2.

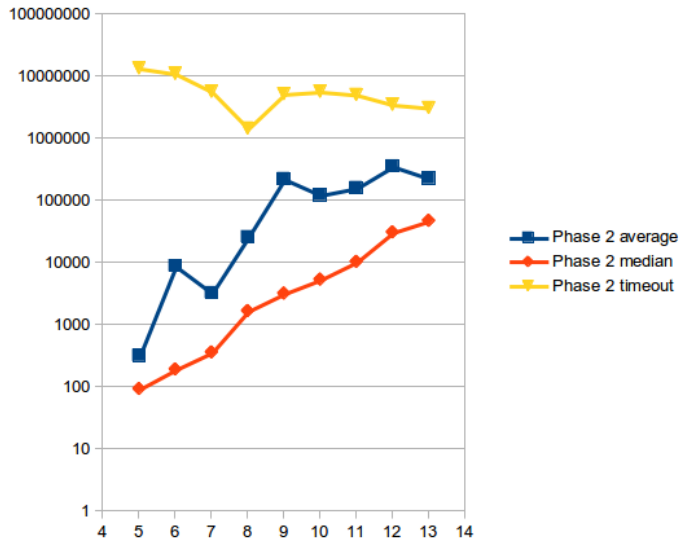

Figure 10: Logarithmic scale graph showing the average and median number of iterations in phase 2, and the average of iterations before a timeout.

It can be seen that the median value is always much lower than the average. From this we assume that in the general case only a low number of iterations are needed until an optimal solution is found, but occasionally solutions are found after a long time of search. These singular cases greatly increase the average, but the median value is not affected as much.

This assumption is further strenghtened by the average number of iterations in the

cases where a timeout occurred. This number is much higher than the average, and is presumably just bounded by the timeout time (300 seconds). It is likely that in most cases, no solution would be found no matter how long the randomized search phase would continue.

This means that there are some near-optimal solutions that do not lie in the neighbourhood of a true optimal solution, or that the landscape between the two is irregular making the optimal solution difficult or impossible to reach with a simple randomized search. One solution could be to use more sophisticated search algorithms, such as tabu search or a genetic algorithm. In this case however, we believe that a better solution could be to try to detect when the algorithm cannot converge. The statistics show that the median number of iterations is relatively low in the cases where an optimal solution is found (i.e. when a solution is found, it is found quickly). Thus if a solution is not found within an estimated number of iterations (for example a constant factor higher than a known median value), the algorithm could go back to phase 1, and find a new near-optimal solution which might or might not be closer to an optimum.

5.6 Correctness of the calculated solutions

Inexact calculations

The numbers used in the algorithm implementation are mostly floating point numbers. While calculations involving floating point numbers are usually not precise, the CGAL library offers the possibility to perform 100% exact calculations, although with the disadvantage of a notable decrease in performance. For this reason, only step 2 (see section 4.1) of the algorithm is done with full precision.

It is theoretically possible that a set of blue points is found, that is believed to covers all circles, when it in reality does not (or vice versa). Note that the Delaunay triangulation that is later constructed using the found points does not depend on the calculated circles and their precision. It is when this Delaunay triangulation is free from red-red neighbours that the program terminates, and thus the correctness of the final solution is not affected by the inexactness of the intermediate calculation, although it is possible that more iterations are performed than what in fact is necessary.

The solution returned are not necessarily correct, since the construction of the Delaunay triangulation also uses inexact calculations. It is therefore possible that a solution returned by our program is in fact a false solution, or that a true solution is overlooked. This may lead to a solution using more blue points than necessary will be accepted and returned. The only way to guarantee that the returned solutions are always correct and minimal would be to do all calculations in an exact way. As exact calculations are costly, it is preferable to perform inexact calculations and in the rare cases where a non-optimal solution is found, try solving the problem again from the beginning.

Finding a point in the interior of a face

Another design issue was that of finding a point in every face of the arrangement (step 4 in section 4.1). The algorithm presented in section 4.3 provides satisfying results, but is relatively complex.

An alternative approach is to simply choose two points on the boundary of a face, and chose the middle point as result, without checking whether it really lies in the considered face. In fact, such a point will be within the face 50% of the cases, since the boundary of the face in a restricted area is either convex or concave with the same probability.

This latter approach is computationally easy, but does not fully fulfill the requirement of the algorithm. As explained above, it cannot lead to incorrect solutions being returned by our program, but it might happen that the returned solution is not minimal. Again, this is a rare phenomenon and the performance gain of using the inexact method is large, and therefore this approach is preferable. The methods were both implemented in a manner such that switching between them is easy.

The former approach is slower to the extent that a comparison in speed is not possible, other than for very small problems (up to five red points). For bigger problems, a large proportion of tests end with a timeout, making automated testing cumbersome.

6 Conclusions

The purpose of this project has been to experimentally explore the conjecture, that in an optimal solution there are as many blue points as red points, except for the case where all red points lie in a row. It must be noted that this claim can be falsified through experimentation by finding a counter-example to the claim, but it can never be experimentally proven. During testing, no counter-examples to the claim have been found, leaving us to believe that this conjecture is in fact true.

7 Discussion

During the course of this project, we have used the developed tool to solve a large amount of problems. Due to the exponential complexity of this problem (shown in figure 7), we have been limited to problems of a relatively small size. To more fully investigate this claim, the solver would likely need to be improved.

One way to do this is to more fully investigate how and when randomized search is at most effective. During our testing, a randomized search phase was entered when the total length of the red segments in the voronoi diagram was smaller than 300*N, where N is the number of red points. This is a very rough estimation, which if too large leads to a local minima, and if too small means the problem could be solved faster. A second parameter, the amount of randomization used, can also be optimized. It is likely that using a similar but more sophisticated technique, such as tabu search, would be more effective.

Another way to improve our solver would be to add more fast heuristics that are likely to find a solution, so that the slow but (almost) exact algorithm described in section 4 would be used rarely.

8 Tools

In this project, a set of tools and libraries were used, most notably the Computational Geometry Algorithms Library (CGAL) provides a vast number of efficient algorithms for different areas within computational geometry. In this project, we have made extensive use of the methods for calculating Voronoi diagrams, triangulations and arrangements.

The Gurobi optimzer was used for solving the integer programming problem (see section 4.4. It provides an easy-to-use C++-API.

To easily build the simple GUI for our program, we used the QT library.

Finally, we used *git* for revision control and to simplify our collaboration. The source code for the project is available at
`https://github.com/martin-mfg/voronoi-thesis-tester`

9 references

1. Ron Wein, Efi Fogel, Baruch Zukerman, and Dan Halperin

   CGAL manual chapter 20, 2D Arrangements

   http://www.cgal.org/Manual/3.3/doc_html/cgal_manual/Arrangement_2/Chapter_main.html

2. O. Aichholzer, R. Fabila-Monroy, T. Hackl, M. van Kreveld, A. Pilz, P. Ramos, und B. Vogtenhuber

   Blocking Delaunay Triangulations.

   Proc. Canadian Conference on Computational Geometry, CCCG 2010, Winnipeg, August 9/11, 2010.

3. M. de Berg, D. Gerrits, A. Khosravi, I. Rutter, C. Tsirogiannis and A. Wolff.

   How Alexander the Great Brought the Greeks Together while Inflicting Minimal Damage to the Barbarians.

   Proc. 26th European Workshop on Computational Geometry, pages 73-76, 2010.

4. Richard M. Karp Reducibility Among Combinatorial Problems. In: R. E. Miller, J. W. Thatcher (Hrsg.): Complexity of Computer Computations. Plenum Press, New York 1972, S. 85-103.

images

- Delaunay circumcircles, GNU Free Documenation Licence, Nü es

  http://commons.wikimedia.org/wiki/File:Delaunay_circumcircles.png

- Delaunay triangulation picture 1-3, public domain, user Capheiden

  http://upload.wikimedia.org/wikipedia/de/1/17/Voronoi-Delaunay.svg

  http://upload.wikimedia.org/wikipedia/de/4/48/Voronoi-Diagramm.svg

  http://upload.wikimedia.org/wikipedia/commons/1/1f/Delaunay-Triangulation.svg