University of Coimbra

M.Sc. Thesis

Visualisation and Analysis of Geographic Information: Algorithms and Data Structures

Author:
João Valença

Supervisor:
Prof. Dr. Luís Paquete
Eng. Pedro Reino

Department of Informatics Engineering

January 2015

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Recent years have seen an exponential growth in informatics in a lot of fields. One such field is that of Geographic Information.
With now ubiquitous technologies such as portable GPS devices, Satellite imagery and many more sophisticated technologies, there are very many ways of retrieving very large quantities of Geographic Data. So much so, that the size of the data available has become too great to be able to be easily interpreted into information.
This means there is now a need to provide ways to efficiently convert very large amounts of geographical data into smaller, more manageable information.

Chapter 2

State of the Art

2.1 Geographic Information Algorithms

2.2 Incremental Approach

2.3 ILP

Chapter 3

Theory and Definitions

3.1 Background and Definitions

In order to be able to quantify representativeness one must first define it. Starting with a set of geographic points N, we must choose a subset, P, that best matches our definition of representativeness. The size of P, however, should not be larger than a given value k, which will specify how many points we can afford to display.

For any point in N not in P, there must be a point in P that best represents it. In a geographic or geometric plane, where points are merely described by their coordinates, the notion of representativeness may be defined by the distance, i.e. the point p that best represents q is the closest point closest to q, under some notion of distance.

This way, finding the best representative set P in N will mean that every point in N will be assigned to the point in P that best represents it. This means that each point in N will be assigned to the point in P closest to it. This definition of representativeness is referred to as *coverage*. The points in P are called *centroids*.

The coverage value of a given centroid, is defined by the circle around that centroid with the radius defined by the distance between itself and the farthest non-centroid point assigned to it. It can thus be more formally described as:

$$\max_{n \in N} \min_{p \in P} \|p - n\| \tag{3.1}$$

Where N is the initial set of points in $\mathbb{R}^2$, P is the centroid subset and $\|\cdot\|$ is the Euclidean distance, which we will use as our notion of distance. The most representative subset,

however, is the one with the minimum value of coverage. This means all points will be assigned to the closest centroid, minimising the coverage of all centroids and avoid coverage areas overlapping whenever possible. We can then finally define our problem as the minimising the coverage:

$$\min_{\substack{P \subseteq N \\ |P|=k}} \max_{n \in N} \min_{p \in P} \|p - n\| \tag{3.2}$$

This is known in the field of optimisation as the *p-centre* problem, and is an example of a facility location problem. It is a *NP-hard* problem, and cannot be solved in polynomial time.

3.2 Defining the Problem in Integer Linear Programming

A simple and straight-forward approach to the problem is to model it in integer linear programming as follows:

$$\text{minimise} \quad D \tag{3.3}$$

$$\text{subject to} \quad \sum_{j=1}^{N} y_j = k \tag{3.4}$$

$$\sum_{j=1}^{N} x_{ij} = 1 \qquad i = 1, \ldots, N \tag{3.5}$$

$$\sum_{j=1}^{N} d_{ij} x_{ij} \leq D \qquad i = 1, \ldots, N \tag{3.6}$$

$$x_{ij} \leq y_j \qquad i = 1, \ldots, N; j = 1, \ldots, N \tag{3.7}$$

$$x_{ij}, y_j \in \{0, 1\} \qquad i = 1, \ldots, N; j = 1, \ldots, N \tag{3.8}$$

In this formulation, $y_j = 1$ if point j is a centroid and $y_j = 0$ if it is a non-centroid. $x_i j = 1$ if the point i is assigned to the centroid j, and $x_{ij} = 0$ if it is not. d_{ij} is the Euclidean distance between points i and j. Constraint 3.4 ensures that k centroids are chosen. Constraint 3.5 limits the assignment of one point to more than one centroid. Constraint 3.6 ensures that all active distances are lower than the limit we are minimising. Constraint 3.7 limits points to being assigned only to centroids, where $y_j = 1$. Constraint 3.8 defines both x_{ij} and y_j as binary variables, in order to properly represent selection and assignment.

It is worth noting that this formulation minimises the objective function by selecting the best possible set of centroids, but it only minimises the maximum coverage. This way, only the farthest point from its centroid has the guarantee that it is connected to its closest centroid.
Every other point, however, can be linked to any centroid so long as it is closer to it than the distance defined by the objective function, since $d_{ij}x_{ij}$ only has to be lower than D, but not include the lowest possible values.
Likewise, it can also produce the result where one centroid is assigned to another centroid, and not itself, as long as they are close enough together. These cases have no effect on the outcome of the final coverage value or the centroid selection, but are rather counter-productive, since we want to minimize the coverage of all centroids, with minimal overlapping of the covered areas.
In order to best display the results, a simple post-processing step can be applied, where each point will be strictly assigned to the closest centroid. This can be easily computed in $\mathcal{O}((N-k)k)$ time in case there is a need for a clearer display of the assignment.

3.3 Algorithmic Concepts

3.3.1 Branch and Bound

Minimising coverage, as we have seen, is a *NP-hard* combinatorial problem. These problems can be approached using Branch and Bound algorithms.

These algorithms solve the problems by recursively dividing the solution set in half, thus *branching* it into a rooted binary tree. At each step of the subdivision, it then calculates the upper and lower bounds for the best possible value for the space of solutions considered at that node. This step is called *bounding*. In the case of a minimisation problem, it would be the upper and lower bounds for minimum possible value for the objective function in the current node. These values are then compared with the best ones already calculated in other branches of the recursive tree, and updated if better.
The idea is to use these values to *prune* the search tree. This can be done when the algorithm arrives at a node where the lower bound is larger than the best calculated upper bound. At this point, no further search within the branch is required, as there is no solution in the current branch better than one that has already been calculated.
In the case that the global upper and lower bound meet, the algorithm has arrived at the best possible solution, and no further computation must be done.

These algorithms are very common in the field of optimisation and can be very efficient, but their performance depends on the complexity and tightness of its bounds. Tighter bounds accelerate the process, but are generally slower to compute, so a compromise has to be made in order to obtain the fastest possible algorithm, and fast bound calculation is a priority for most approaches.

3.4 Geometric Concepts and Structures

3.4.1 Nearest Neighbour Search

A common step of geometry-based algorithms to solve the problem is performing a nearest neighbour search. Since we need to find the closest centroid to a given point, this operation will be one of the most used, and so we need a fast and flexible way of determining which of the centroids is closest, in order to reduce overhead. Nearest neighbour search algorithms find a point q in N closest to p according to a definition of distance.

3.4.2 Point Location

A concept commonly associated with nearest neighbour search is point location. One way to efficiently make a large number o point location queries is to partition the plane into regions. Point location algorithms direct the nearest neighbour search to smaller regions, discarding any points that are too distant from the query, thus reducing the number of calculations necessary to get the proper point location. Common structures used for point location are *k-d trees* as described in [[Cambazard]]. A *k-d tree* partitions the space using a divide and conquer approach to define orthogonally aligned half-planes in order to achieve point location queries in $\mathcal{O}(\log n)$ time. A *k-d tree*, however, needs to be periodically updated in order to keep its efficiency and cannot be constructed or deconstructed incrementally without considering this overhead.

3.4.3 Voronoi Diagrams

Voronoi diagrams partition the space into regions, which are defined by the set of points in the space that are closest to a subset of those points. Definitions of distance and space may vary, but on our case we will consider the $\mathbb{R}^2$ plane and the Euclidean distance.
Figure 3.1 shows a partitioning of a plane using a Voronoi Diagram for a set of points:

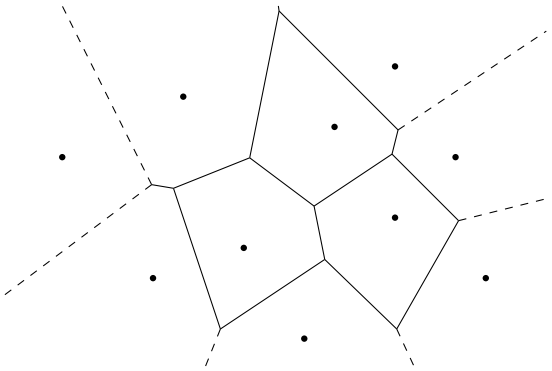

FIGURE 3.1: Example of a Voronoi Diagram

Dashed lines extend to infinity. Any new point inserted in this plane is contained in one of the cells, and its closest point is the one at the centre of the cell.
Each edge is the perpendicular bisector between two neighbouring points, dividing the plane in two half planes, containing the set of points closest to each of them.
The construction of Voronoi diagrams can be done incrementally, but in order to obtain fast query times, one needs to decompose the cells into simpler structures.

3.4.4 Delaunay Triangulations

Another useful structure for geometric algorithms is the Delaunay triangulation. A Delaunay triangulation is a special kind of triangulation with many useful properties. In an unconstrained Delaunay triangulation, each triangle's circumcircle contains no points other inside its circumference.
It maximizes the minimum angle in its triangles, avoiding long or slender triangles. The set of all its edges contains both the minimum-spanning tree and the convex hull. The Delaunay triangulation is unique for a set of points, except when it contains a subset that can be placed along the same circumference. Figure 3.2 shows the Delaunay triangulation of the same set of points used in Figure 3.1:

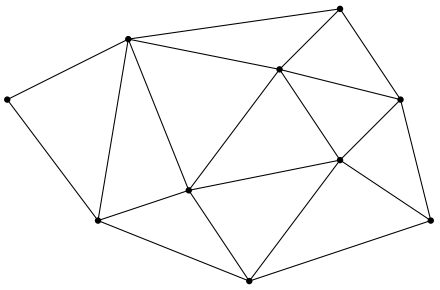

FIGURE 3.2: Example of a Delaunay Triangulation

More importantly, the Delaunay triangulation of a set of points is the dual graph of its Voronoi Diagram. The edges of the Voronoi diagram, are the line segments connecting the circumcentres of the Delaunay triangles. When overlapped, the duality becomes more obvious. Figure 3.3 shows the overlapping of the Voronoi diagram in 3.1 and the Delaunay triangulation in 3.2. The Delaunay edges, in black, connect the points at the centre of the Voronoi cells, with edges in red, to their neighbours.

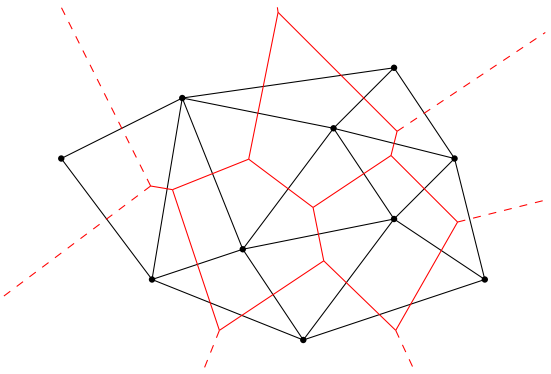

FIGURE 3.3: Overlap of a Voronoi Diagram and its Delaunay Triangulation

Unlike its counterpart, the Delaunay is much simpler to build incrementally. It is also easier to work with, whilst still providing most of the Voronoi diagram's properties, including the ability to calculate both point location and nearest neighbour searches.

3.4.4.1 Greedy Routing

In order to quickly calculate the nearest neighbour to a point in a set, one can make use of the Delaunay triangulation.
Consider a triangulation $\mathcal{T}$. In order to find the closest vertex in $\mathcal{T}$ to a new point p, we must start at an arbitrary vertex of $\mathcal{T}$, v, and find a neighbour u of v whose distance to p is smaller than the distance between p and v. Repeat the process for u and its neighbours. When we reach a point w such that no neighbours of w are closer to p than w is, we have found the closest point to p in $\mathcal{T}$.

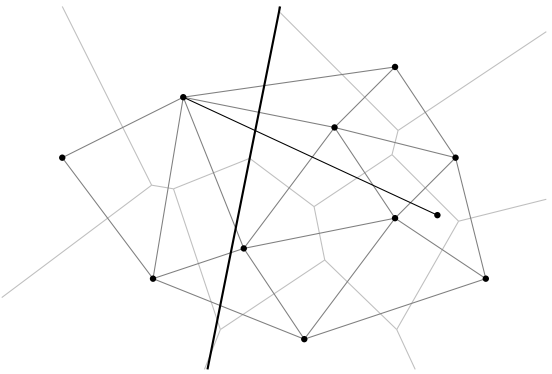

FIGURE 3.4: Example of the Greedy Routing Algorithm

Theorem 3.1. *There is no point set whose Delaunay triangulation defeats the greedy routing algorithm.*

Proof. For every vertex v in a triangulation $\mathcal{T}$, let the perpendicular bisector of the line segment defined by v and u be called e if there is at least one neighbour of v, u closer to p than v is. The line e intersects the line segment (v, p) and divides the plane in two open half planes: h_v and h_u. Note that the half plane h_u contains p.
Delaunay edges connect the Voronoi neighbours and their bisectors define the edges of the Voronoi cells, which are convex polygons.
Repeating the process recursively for u, if a point w is found, whose neighbourhood contains no points closer to p than itself, then p is contained within all possible open half planes containing w, defined by w and all its neighbours. Point p is then by definition located in point w's Voronoi cell. This means that w is the point in $\mathcal{T}$ closest to p. □

3.4.4.2 Line Walking

Another point location algorithm to consider is the line walking algorithm. This algorithm finds a triangle t in a triangulation $\mathcal{T}$ that contains a given point v.
Starting at any triangle s, with the geometrical centre m, if point v is not contained in s, then the line segment (v, m) intersects a finite set of triangles. The line segment (v, m) intersects two edges of each triangle in this set, with the exception of s and t where (v, m) only intersects one edge each. By iterating through each triangle choosing the neighbour that contains the next edge that intersects (v, m), triangle t can be found in $\mathcal{O}(n)$ time.
This algorithm was described by Amenta, Choi, and Rote [1], and is ilustrated in Figure 3.5.

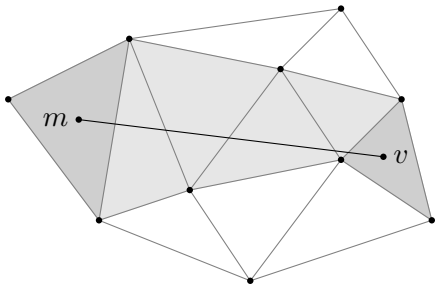

FIGURE 3.5: Example of the Walking Algorithm

3.4.4.3 Construction

There are many algorithms to construct a Delaunay triangulation. The particular conditions of our approach to the coverage problem impose some restrictions to the choice of the algorithm to use.
In order to implement an incremental branch and bound approach, our triangulation algorithm has to not only be efficient, but also to be able to be done incrementally. The Bowyer-Watson algorithm allows us to do both.
Starting with a valid Delaunay triangulation $\mathcal{T}$, we find a triangle $t = \triangle_{abc}$ that contains the vertex to insert v using a point location algorithm, such as the walking algorithm. We then use algorithm 1:

Algorithm 1 Bowyer-Watson Algorithm

```
1: procedure InsertVertex(v, a, b, c)
2:     DeleteTriangle(a, b, c)
3:     DigCavity(v, a, b)
4:     DigCavity(v, b, c)
5:     DigCavity(v, c, a)

1: procedure DigCavity(a, b, c)
2:     d ← Adjacent(b, c)
3:     if d ≠ ∅ then
4:         if inCircle(a, b, c, d) then
5:             DeleteTriangle(w, v, x)
6:             DigCavity(a, b, d)
7:             DigCavity(a, d, c)
8:         else
9:             AddTriangle(a, b, c)
```

For inserting n points, this algorithm has an expected time complexity of $\mathcal{O}(n \log n)$.

3.4.4.4 Deconstruction

Deconstructing a Delaunay triangulation usually takes an algorithm as complex as the construction. However, since in our case the deconstruction has to be incremental, and the first point to remove from the triangulation is necessarily the last one to be inserted, we can use a much simpler approach.
At each step of the construction, all created and removed edges and triangle from the triangulation can be stored in a LIFO structure, or a stack. When the last inserted point is to be removed, recreating the previous state of the triangulation is only a matter of rolling back and retrieving the information from the stack. This also means no geometrical calculations have to be done, and the old edges and triangles are quickly put back in place, with no new memory allocation needed.

3.4.4.5 Half-Edge Structure

A useful structure to use when building and managing triangulation meshes is the half-edge structure. The half-edge structure represents one orientation of each edge in the triangulation. This means that for each pair of points (p_i, p_j)connected in a triangulation $\mathcal{T}$, there are two directed half-edges: one represents the edge from p_i to p_j, and the other represents the opposite face, that connects p_j to p_i. They both contain information about the triangle that they are part of. The triangles each of them form with two other half-edges contains both p_i and p_j, and are thus, adjacent. The way to tell which triangle is which is by defining the triangles by listing their points in a counter-clockwise order. Figure 3.6 further illustrates the concept of the half-edges.

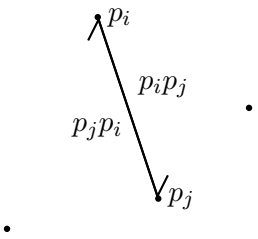

FIGURE 3.6: Example of a Representation of the Half-Edge Structure

This structure makes it easier to store the changes to the triangulation in each step, since they contain the information about the triangles themselves. This means that only the half edges need to be stored in the stack, with no need to deal with the triangles directly. The half-edge structure helps to obtain the triangulation neighbours for any vertex v,

since these are all the end points to the half-edges starting at v, which is useful when implementing the greedy routing algorithm described in 3.4.4.1.

3.4.5 Hilbert Curves

Most of the point location algorithms aforementioned have linear time complexity, and most of the worst case scenarios include searching across the plane. These occur when the starting search position is random and does not make use of the spatial organisation of the data. In order to fully take advantage of these approaches, the points should be sorted is such a way that minimizes the distance between consecutive point such as a Hilbert curve.

Hilbert curves are a kind of fractal space-filling curves that generally minimize the Euclidean distance between points close on the curve.

True Hilbert curves map a 2-dimensional space in a 1-dimension line. This line has an infinite length, which makes mapping 2-dimensional points to it infeasible. Instead, discrete approximations are used. Since the true curve is fractal, the approximations are defined by the number of fractal steps it takes in order to reach them. Figure 3.7 demonstrates the first few orders of approximation:

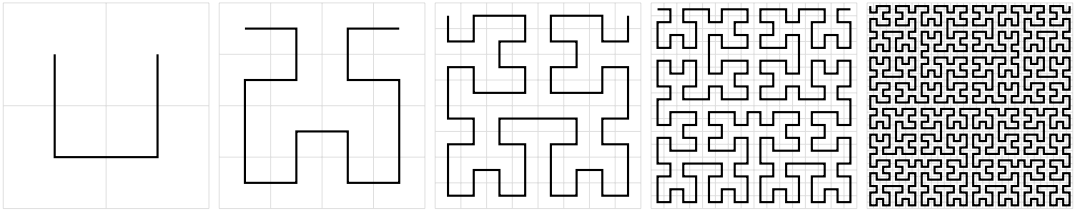

FIGURE 3.7: First Five Orders of Hilbert Curve Approximation

Since the coordinates of the points in our problem are continuous rather than discrete, the points must first be mapped into a square grid with tile size and number appropriate to the Hilbert approximation chosen.

In order to sort an array of 2-dimensional points to follow a Hilbert approximation, each point should be assigned the 1-dimensional coordinate of the square tile that contains that point. The array is then sorted using the square coordinates along the Hilbert approximation as a key.

This means that there are cases where more than one point will share the same discrete approximation coordinates, but this has little effect on the performance of the point location algorithms, as long as the grid is fine enough to separate most of the points. The space must be partitioned into a grid of 2^n squares in height and width and the grid must be able to contain all points.

Chapter 4

Optimal Minimum Coverage Algorithms

This chapter covers three possible algorithms that solve the coverage problem described in 3.1. All these algorithms calculate the subset P of size K from a set N, in a way that minimises the maximum coverage value for all points in P, over the remaining points in N.

4.1 Combinatorial Approach

The simplest way to implement a solution to the coverage problem is to enumerate all combinations of K elements from N. Each of these combinations is the list of candidate centroids, and every point in N is assigned to its closest centroid. This assignement step takes $\mathcal{O}(\mathcal{NK})$ for each combination of K elements.
Finding the minimum coverage value in this approach is trivial, since it is just a matter selecting the combination of centroids that has the minimum distance between the farthest point from its centroid.
This approach takes $\mathcal{O}(K!)$ time complexity, and takes no advantage from previously calculated candidate solutions.

4.2 Branch and Bound

A more sophisticated approach to the problem is to use a branch and bound method.
In this approach, we build the assignment between centroids and non-centroids incrementally.

At each step of the recursive tree, we consider one of the points. We then take a decision of whether the point is a centroid or a non-centroid. According to which decision we take, the objective function and the centroid assignment is updated accordingly. We do this until all the centroids have been chosen.

4.2.1 Branching

As stated above, branching the tree involves updating the assignment between new points and/or new centroids, as well as update the objective function. The following algorithms explain in detail how to do so.

Inserting a Centroid To insert a centroid c, we must check which points are closer to c than their current centroid, and change their assignment to c. Since non-centroids only change assignment to centroids closer to them, inserting a centroid means that the objective function either decreases in value or stays the same.
After inserting a centroid, if the farthest non-centroid is reassigned, all non-centroids must be checked to see which one is the new maximiser of the objective function.
This step compares all non-centroids to the new centroid c, taking $\mathcal{O}(N)$ time at both the worst and best cases.

Inserting a Non-Centroid Inserting a non-centroid n only requires finding which of the current centroids is the closest to n. Updating the objective function is a matter of testing whether the distance between n and its centroid is larger than the current maximum.
Inserting a non-centroid cannot produce a better objective function, since it will either decrease or maintain the current value.
This step compares the distance between point n and all centroids, taking $\mathcal{O}(K)$ time.

Removing a Centroid After a branch is fully calculated, it is necessary to backtrack to the parent state. This means removing a centroid c, and redistributing the values assigned to c to their respective closest centroids in the remaining set.
The value function either increases or maintains, since the distance for all points previously assigned to c will increase, potentially above the current value for the objective function.
Removing a centroid c means comparing all non-centroids assigned to c to all the other centroids. This step takes $\mathcal{O}(NK)$ time to execute. Alternatively, if the assignment state is saved before inserting the centroid, recovering it requires only retrieving the

state, which means, in the worst and best cases copying an array of size N, which takes $\mathcal{O}(N)$ time at the expense of additional $\mathcal{O}(N)$ memory space.

Removing a Non-Centroid In order to remove a non-centroid n, we only need to update the objective function. If n is the maximiser of the objective function, we must find the second farthest point from its centroid, which is the new maximiser.
Removing a non-centroid can either decrease or maintain the value of the objective function.
Removing a non-centroid n means that the next farthest point from its centroid must be found. This can be done by checking all distances between the non-centroids and their respective centroids, taking $\mathcal{O}(N)$ time. Alternatively, one can save the previous value for the objective function, as well as the maximiser. Retrieving the previous value this way can be done in $\mathcal{O}(1)$ time at the expense of additional $\mathcal{O}(1)$ memory space.

4.2.2 Bounds

At all points in the branching, we calculate a lower bound for the value of the objective function the current state's sub-branches. If the lower bound is higher than the best value calculated, then there is no purpose in further exploring the current branch.

Lower Bound After each insertion, centroid or non-centroid, one can assume that, the best case scenario, all the points not yet inserted will be centroids. This would hypothetically decrease the value the most. If this value is higher than the best value of the objective function, then there is no permutation of centroids/non-centroids that will improve the current solution, and the branch can be pruned.

4.3 Delaunay Assisted Branch and Bound

Most of the operations in the branch and bound approach described in 4.2 have at least linear time complexity for both the best and expected cases. We can speed these up by implementing incrementally built Delaunay triangulations. To aid the calculations, the points are pre-processed and sorted by a Hilbert Curve approximation of a sufficiently high order.

Inserting a Centroid In order to take advantage of Delaunay triangulations, each time a centroid is chosen, it must be included in the Delaunay triangulation. This means that the triangulation must be updated. Inserting a point in a triangulation with K vertices using the Bowyer-Watson algorithm described in 3.4.4.3 takes an estimated $\mathcal{O}(\log K)$ [2].

After having a centroid c included in a Delaunay triangulation, it is possible to know which other centroids are its Voronoi neighbours. This is due to the duality between Delaunay triangulations and Voronoi diagrams. Since Voronoi diagrams partition the space by distance to a set of points (in this case, the centroids), we only need to check the subset of non-centroids assigned to the direct neighbours of c to find which points should change assignment to c. This property lowers the expected number of comparisons to make. Since the average number of Voronoi neighbours per centroid in any given diagram cannot exceed 6 [3], the number of points to be compared in a uniformly distributed set of non-centroids should not include all non-centroids, but only a small fraction of them. Despite the lower number of comparisons, the worst-case time complexity is still $\mathcal{O}(N)$ time to complete, and in the worst case scenario it can still require a check through all non-centroids, which can all be assigned to the neighbours of c.

If the objective function maximiser is assigned to c, all non-centroids can be candidates to become the new maximiser, so a linear check must be done.

Inserting a Non-Centroid Since there is a triangulation in place, finding the closest centroid c to a new non-centroid n is simply a matter of using the greedy routing algorithm to find this centroid.

The greedy routing algorithm has a worst-case time complexity of $\mathcal{O}(K)$. This happens when the search is started in the furthest centroid from n, and all centroids are either in the direction of n, or are neighbours of the centroids that are. Since the points are inserted ordered by a Hilbert curve approximation, each consecutive point should minimize the position variation from the last, and the last centroid returned by the greedy routing algorithm can be used to start the new query. This means that, ideally, each inserted non-centroid n is close to its respective optimally positioned centroid c, and it will only need to calculate the distances to the neighbours of c in order to guarantee that c is indeed the correct centroid. The aforementioned property of the average 6 neighbours for each centroid means that the expected time for a query starting at the right centroid is $\mathcal{O}(1)$, and this is heuristically approximated by the Hilbert curves.

The time complexity of inserting one non-centroid is still $\mathcal{O}(K)$ for the worst case. However, the insertion of a large number of uniformly distributed and Hilbert-sorted points *should* behave closer to constant time per point.

Removing a Centroid Removing a centroid c means removing it from the Delaunay triangulation and redistributing all points assigned to c across its neighbours.
Since all points are inserted in the triangulation in a last-in first-out order, removing a point from a triangulation is a matter of retrieving the previous state. We can do this by storing all new edges and triangles in a stack upon construction, and retrieve them upon removal, without the need of recalculating anything. Since inserting a centroid c takes expected $\mathcal{O}(\log n)$ time [2], and removing it will take exactly the same higher level operations (in reverse order), it can also be done in expected $\mathcal{O}(\log n)$ time, without the need to do extra calculations.
Likewise, redistributing the points assigned to c takes retrieving the previous state. Each change in assignment can be saved in a stack upon insertion, and retrieving it can be done by popping the stack.
This step also takes $\mathcal{O}(N)$ time, since all points can change assignment. Using a stack limits the number of operations to only the ones changed upon insertion.

Removing a Non-Centroid Removing a non-centroid n only requires recovering the second farthest point if n is currently the farthest point, otherwise, no operations besides erasing n's assignment, taking $\mathcal{O}(1)$ time and memory.

Despite having the same worst-case time complexity as the branch and bound algorithm described in 4.2, the expected time complexity for the Delaunay assisted approach is lower. This approach should have better performance when higher numbers of centroids are needed.
This is especially true since maintaining a valid Delaunay triangulation through all the centroid permutations, as well as the Hilbert sorting, takes a computing cost. This extra overhead will have a negative impact in the performance in the smaller instances of the problem.

4.4 Results?

Chapter 5

Heuristic Algorithms and Future Work

5.1 Approximating the Problem

5.2 Panning and Zooming

5.3 Scalable Algorithms

5.4 Usable Structures

5.5 Uniformity

Chapter 6

Conclusion

Bibliography

[1] Nina Amenta, Sunghee Choi, and Günter Rote. Incremental constructions con brio. *Proceedings of the Nineteenth Annual Symposium on Computational Geometry (San Diego, California)*, pages 211–219, June 2003.

[2] Shewchuk.

[3] find.