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 witnessed a large increase in both the quantity and availability of geographic data. This new availability of such large quantities of data has prompted a similar rise on the number of applications to capture, store, manipulate and analyse this data. A lot of these applications share the need to visualise the geographic information in such a way that it can be easily understandable by a human. This is usually done by displaying points of interest on a map so that their relative position or direction can be easily interpreted without much thought by the user.
One obstacle when representing large amounts of geographic data, is that the sheer volume of points to display can be overwhelming for a human, as well as computationally intensive to render for a machine. As such, there is a need to develop and implement a viable way to reliably calculate and display a subset of geographic points, whilst keeping a degree of representability of the larger set, so that as little information as possible is absent when the representative subset is shown.
This project aims to research and develop a real-time algorithm that can analyse geographic data provided by a geographic information system (GIS) infrastructure developed and maintained at Smartgeo. More precisely, the developed algorithm will have to be able to aggregate and select geographic points according to a given a set of criteria.
This thesis aims to research, develop, and analyse different algorithms to choose a representative subset of geographic points, whilst being able to dynamically change that set of points via zooming or panning over a geographic region containing a large amount of geographical data. Heuristic algorithms will have their solution quality and speed benchmarked against implicit enumeration algorithms, and the one with the best results will be implemented in the web framework via the *WMS* and *WFS* standards.
This report is organized as follows: Chapter 2 - Theory and Definitions defines the base theoretical concepts, such as a notion of representativeness, as well as some useful structures used in the algorithms. Chapter 3 - State of the Art analyses previous related

work. Chapter 4 describes the implicit enumeration algorithms, as well as an analysis on their time and space complexities. Chapter 5 - Future Work describes the direction of the heurisitc algorithms to be developed in the second half of the thesis. Finally, Chapter 6 - Conclusion will provide the conclusions reached by this thesis so far.

Chapter 2

Theory and Definitions

This chapter is an introduction to and explanation of the base concepts used further in this report. The chapter starts by establishing the formal definition of the problem at hand. Then it proceeds to detail the algorithmic and geometric concepts to be used in the different approaches described in the following chapters.

2.1 Definitions of Coverage

Representation consists of finding a subset of points in a larger set. The subset chosen should be able to keep some specified properties of the original set, such as density, or general distribution. As such there can be many ways to define representativeness. For the purposes of this thesis, we will use the definition of *coverage*.

Given a set of points N in $\mathbb{R}^2$, we must choose a subset, P, that best matches our definition of representativeness. The size of P, however, is constrained to a size k, which will specify how many points can be afforded to be displayed in a section of a map.

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, this 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. Given the geographic nature of the input data, this thesis will use the Euclidean norm as the distance notion.

Finding the most 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. The coverage value of a subset if determined by the highest coverage value of its points. It can thus be more formally described as:

$$\max_{n \in N} \min_{p \in P} \|p - n\| \tag{2.1}$$

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 avoiding overlapping coverage areas whenever possible. We can then finally define our problem as the minimising the coverage:

$$\min_{\substack{P \subseteq N \\ |P| = k}} \max_{n \in N} \min_{p \in P} \|p - n\| \tag{2.2}$$

This is known in the field of optimisation as the *p-centre* problem, and is an example of a facility location problem [**?**]. For the 1-dimensional case, it can be calculated in polynomial time [**?**]. However, for any other number of dimensions it is a *NP-hard* problem, and cannot be solved in polynomial time [**?**].

2.2 Algorithmic Concepts

2.2.1 Branch-and-Bound

Minimising coverage, as we have seen, is a *NP-hard* combinatorial problem [**?**]. One possible way of solving the problem is to us implicit enumeration algorithms such as *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 branch-and-bound 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 usually slower to compute, so a compromise has to be made in order to obtain the fastest possible algorithm.

2.3 Geometric Concepts and Geometric Structures

In the following, we explain some geometric concepts that are used in the following chapters, in order to simplify the explanation of more complex algorithms in further chapters.

2.3.1 Nearest Neighbour Search and Point Location

A common concept in computational geometry is point location. A point location algorithm finds the correct region on a plane that contains a given point p. Depending on the nature and shape of the regions, point location algorithms may differ in approach. In this thesis, most point location problems consist of finding the closest centroid to a given point, i.e. a nearest neighbour search algorithm.

Given a point p, a *nearest neighbour search* algorithm returns the closest point to p in a given set. Since we need to find the closest centroid to a given point in order to find the correct coverage value, 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 computational overhead.

Point location algorithms direct the nearest neighbour search to smaller regions, bypassing any regions that are too distant from p, 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 et al. [1]. A *k-d tree* partitions the space using a divide and conquer approach to define orthogonally aligned half-planes. This approach takes $\mathcal{O}(\log n)$ time to achieve point location queries. However, a *k-d tree* needs to be periodically updated in order to keep its efficiency and cannot be constructed or deconstructed incrementally without considering this overhead.

2.3.2 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 2.1 shows a partitioning of a plane using a Voronoi Diagram for a set of points:

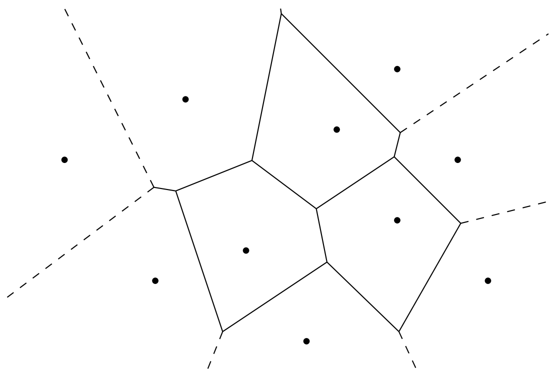

FIGURE 2.1: Example of a Voronoi Diagram

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.

2.3.3 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.

A Delaunay triangulation maximizes the minimum angle of its triangles, avoiding long or slender ones. 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 2.2 shows the Delaunay triangulation of the same set of points used in Figure 2.1:

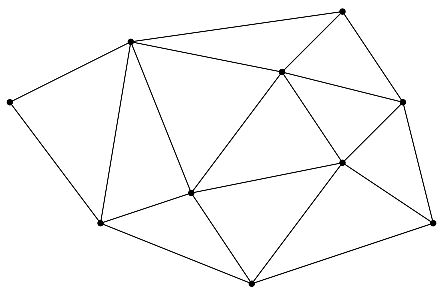

FIGURE 2.2: Example of a Delaunay Triangulation

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 2.3 shows the overlapping of the Voronoi diagram in 2.1 and the Delaunay triangulation in 2.2. The Delaunay edges, in black, connect the points at the centre of the Voronoi cells, with edges in red, to their neighbours.

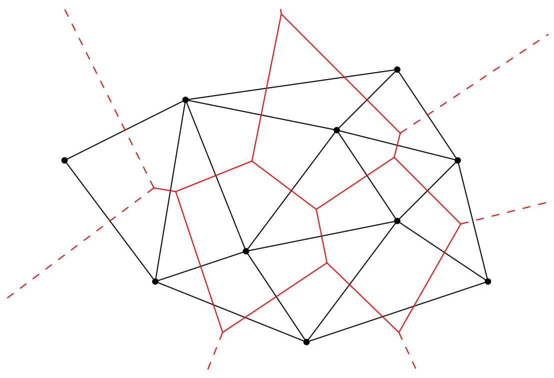

FIGURE 2.3: Overlap of a Voronoi Diagram and its Delaunay Triangulation

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.

2.3.3.1 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 $\mathcal{T}$.

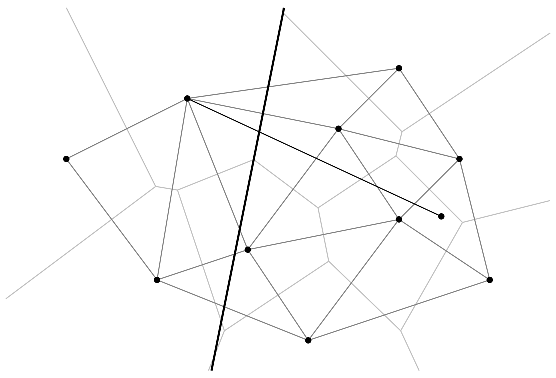

Figure 2.4: Example of the Greedy Routing Algorithm

Theorem 2.1. *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. □

2.3.3.2 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 et al. [2], and is ilustrated in Figure 2.5.

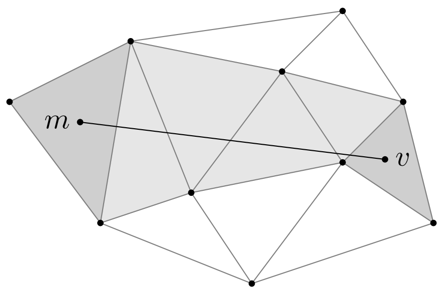

Figure 2.5: Example of the Walking Algorithm

2.3.3.3 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.
Building a Delaunay triangulation can be done incrementally. Starting with a valid triangulation, points can be added, creating and updating existing triangles. An efficient way to do so is to use the Bowyer-Watson algorithm.

Starting with a valid Delaunay triangulation $\mathcal{T}$, we find the triangle t with vertices a,b and c that contains the vertex to insert v using a point location algorithm, such as the line walking algorithm described in the previous section. We then follow algorithm 1 for each vertex v to be included in the triangulation:

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:     $d \leftarrow$ Adjacent($b, c$)
3:     if $d \neq \varnothing$ 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($a, b, c$)
```

The algorithm starts by removing the triangle t that contains the new vertex v, and recursively checks adjacent triangles whose circumcircle contains v using the *DigCavity* function. Any triangle that contains v in its circumcircle, violates the Delaunay rule, and must also be deleted and have its sides recursively checked, until no adjacent triangles violate the Delaunay rule.
For inserting n points, this algorithm has an expected time complexity of $\mathcal{O}(n \log n)$ and is described in more detail by Shewchuk [3].

2.3.3.4 Deconstruction

Deconstructing a Delaunay triangulation usually takes reversing the construction algorithm to remove points from the triangulation. However, as we will explain in a later chapter, 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 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.

2.3.3.5 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 direction, that connects p_j to p_i. They both contain information about the triangle that they face, and thus, are part of. Triangles are defined by three half edges. All the half edges in the triangle share two of the vertices of the triangle, and are all sorted in a counter-clockwise order. Figure 2.6 further illustrates the concept of the half-edges.

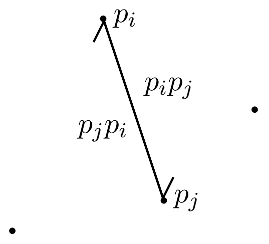

FIGURE 2.6: Example of a Representation of the Half-Edge Structure

This structure makes it easier to store the changes to the triangulation at 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 (for construction and deconstruction), with no need to manage the triangles directly. The half-edge structure helps to obtain the triangulation neighbours for any vertex v, since previously these are all the end points to the half-edges starting at v, which is useful when implementing the greedy routing algorithm described in 2.3.3.1.

2.3.4 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 2.7 demonstrates the first few orders of approximation:

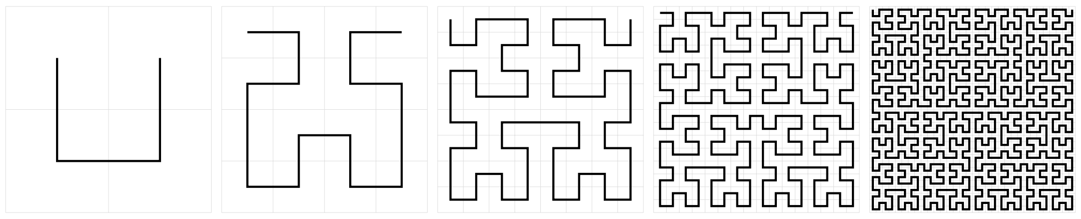

FIGURE 2.7: First Five Orders of Hilbert Curve Approximation

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 3

State of the Art

The *p-center* problem is a well known problem in the optimisation field of study. As such, several approaches have been explored over the years.

3.1 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:

$$\begin{aligned}
&\text{minimise} && D && && (3.1)\\
&\text{subject to} && \sum_{j=1}^{N} y_j = k && && (3.2)\\
& && \sum_{j=1}^{N} x_{ij} = 1 && i = 1, \ldots, N && (3.3)\\
& && \sum_{j=1}^{N} d_{ij} x_{ij} \leq D && i = 1, \ldots, N && (3.4)\\
& && x_{ij} \leq y_j && i = 1, \ldots, N; j = 1, \ldots, N && (3.5)\\
& && x_{ij}, y_j \in \{0, 1\} && i = 1, \ldots, N; j = 1, \ldots, N && (3.6)
\end{aligned}$$

In this formulation, $y_j = 1$ if point j is a centroid and $y_j = 0$ if it is a non-centroid. $x_ij = 1$ 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 3.2 ensures that k centroids are chosen. Constraint 3.3 limits the assignment of one point to more than one centroid. Constraint 3.4 ensures that all active distances are lower than the limit we are minimising. Constraint 3.5 limits points to being assigned only to centroids, where $y_j = 1$.

Constraint 3.6 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.

Other, more elaborate formulations can be used. Elloumi et al. [4] explore a new formulation to obtain tighter bounds in the LP relaxation. They also limit the values that the solution can take, by enumerating all different values of distances between points, and sorting them in decreasing order.

3.2 Incremental Approach

A different approach using procedural programming is described by Cambazard et al. [1]. Their method describes algorithms to insert and remove centroids from a set of points, and update the centroid assignment in the geometrical area surrounding the changed centroid. This method is a form of local search, that allows for small modifications on an already valid solution, until a similar solution is deemed optimal is found.

In order to minimise computation time, Cambazard et al. [1] maintain the selected centroids in a *kd-tree*. Using a *k-d tree* reduces the number of comparisons needed for the point location steps in the algorithm. To keep them from loosing efficiency in insertions and removal of points, the trees need to be balanced from time to time.

Chapter 4

Optimal Minimum Coverage Algorithms

This chapter covers two possible algorithms that solve the coverage problem described in 2.1. Both algorithms use a similar incremental branch-and-bound approach for implicit enumeration of the centroid subsets.
The first algorithm uses simple loops over arrays for point location queries, while the second one builds and uses a Delaunay triangulation and its properties for that purpose.

4.1 Branch-and-Bound

A more sophisticated approach to the problem is to use a branch-and-bound method. In this approach, the assignment of non-centroids to their correct centroids is built incrementally.
At each step of the recursive tree, one of the points is considered. A decision is then taken of whether the point is a centroid or a non-centroid. According to which decision is taken, the objective function and the centroid assignment is updated accordingly. This is done until all the centroids have been chosen.

4.1.1 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, the established non-centroids which are closer to c than their current centroid must be checked, 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 now maximises the objective function.
This step compares all non-centroids to the new centroid c, taking $\Theta(N)$ time.

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 $\Theta(K)$ time.

After a branch is fully calculated, it is necessary to backtrack to the parent state, either by removing a centroid, or a non-centroid.

Removing a Centroid Removing a centroid c means 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 $\Theta(NK)$ 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 $\Theta(N)$ time at the expense of additional $\Theta(N)$ memory space.

Removing a Non-Centroid In order to remove a non-centroid n, we only need to update the objective function. If point n maximises the objective function, the second farthest point from its centroid, the new maximiser, must be found.
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 $\Theta(N)$ 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 $\Theta(1)$ time at the expense of additional $\Theta(1)$ memory space.

4.1.2 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 larger than the upper bound calculated, then there is no purpose in further exploring the current branch. In a minimisation problem, the upper bound can be the best solution found until that point in time.

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 larger than the best value found, then there is no possible assignment that will improve the current solution in the current branch, and the branch can be pruned.

4.2 Delaunay Assisted Branch-and-Bound

Most of the operations in the branch-and-bound approach described in section 4.1 have at least linear time complexity for both the best and expected cases. We can speed these up by implementing incrementally built Delaunay triangulations, which can be used to accelerate point location queries. 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 section 2.3.3.3 takes an estimated $\mathcal{O}(\log K)$ for a uniformely distributed set of points [3].
After a centroid c is 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 in

regions by distance to 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 [5] [6], 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 still takes $\mathcal{O}(N)$ time to complete, and in the worst case scenario it can still require a check through all non-centroids, which can all be 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 search through all the non-centroids must be done, to see which one is now the farthest away from its centroid.

Inserting a Non-Centroid Since there is a triangulation built, using the centroids as its vertices, finding the closest centroid c to a new non-centroid n is simply a matter of using the greedy routing algorithm to find c [7].
The greedy routing algorithm has a worst-case time complexity of $\mathcal{O}(K)$. This happens when the search starts from the farthest centroid from n, and all centroids are either in the direction of n, or are neighbours of the centroids that are. The last centroid returned by the greedy routing algorithm can be used to start the new query. Since the points are inserted ordered by a Hilbert curve approximation, each consecutive point should minimise the position variation from the last. This means that, ideally, each inserted non-centroid n will be 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 $\mathcal{O}(6\frac{N}{K})$, 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 $\mathcal{O}(\log K)$ time [3], and removing it will take exactly the same higher level

operations (in reverse order), it can also be done in expected $\mathcal{O}(\log K)$ 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 those that 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 section 4.1, 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.

4.3 Algorithm Comparison

Algorithm	Insert		Remove	
	Centroid	Non-Centroid	Centroid	Non-Centroid
Branch-and-Bound	$\Theta(N)$	$\Theta(K)$	$\Theta(N)$	$\Theta(1)$
Delaunay Assisted	$\mathcal{O}(K) + \mathcal{O}(6\frac{N}{K})$	$\mathcal{O}(\log K)$	$\mathcal{O}(N)$	$\mathcal{O}(1)$

Table 4.1: Expected time complexities for the various operations in a uniformly distributed set

Chapter 5

Future Work

5.1 Integration in a Visualisation Framework

The final objective of this study is to integrate an algorithm in a web application that can display the most representative set as possible. There is, however, a time constraint to do this. The feedback time on the application needs to be as small as possible while still delivering an acceptable set of points, in order to not lose engagement from the user.

The application will display a rectangular window, showing a cut of geographical region containing a set of points. The algorithm chosen will need to be able to choose a representative set of points within the cut quickly, as well as be able to recalculate a new set points for a new cut, resulting from panning or zooming the display window over the region.

The algorithm serve as the middleware responsible for filtering the response of a GIS server to a Web Map Service, or WMS request. WMS lists the geographic coordinates of the points to be represented in an image by the coordinates mapped into orthogonally organised pixels on an image displaying the cut of the region requested by the application. The candidate algorithms will be tested and benchmarked using data from the Open Street Map project. The project features large quantities of open source geographic data, as well as a versatile API for fetching data in the WMS standard.

5.2 Heuristic Approaches

Optimal solution algorithms, and their slow time performance, makes them a poor choice for real-time applications. As such, heuristic algorithms that provide good but not optimal solution in faster time are more likely the most appropriate approach. Since a lot of complex structures have already been explored and implemented in the implicit enumeration approaches, a lot of the concepts and methods can be repurposed and reused when experimenting and researching heuristic approaches.

5.3 Uniformity

Another measure of quality for a solution is its *uniformity*. Uniformity is defined mathematically in a set of points as the distance between the closest pair of points. The most representative subset U of a larger set N relative to its uniformity will be the subset of N that has the highest value for the distance of its closest pair. Like coverage, the concept of uniformity is frequent in the field of optimisation. Maximising uniformity can be formally defined as:

$$\max_{U \in N} \min_{\substack{u_i, u_j \in U \\ u_i \neq u_j}} \|u_i - u_j\| \tag{5.1}$$

Where N is the initial set of points in $\mathbb{R}^2$, U is the most uniform subset in N, and $\|\cdot\|$ is the Euclidean distance between two points. The maximum uniformity is the most representative set.

Chapter 6

Conclusion

Bibliography

[1] Hadrien Cambazard, Deepak Mehta, Barry O'Sullivan, and Luis Quesada. A computational geometry-based local search algorithm for planar location problems. In *Integration of AI and OR Techniques in Contraint Programming for Combinatorial Optimzation Problems*, pages 97–112. Springer, 2012.

[2] Nina Amenta, Sunghee Choi, and Günter Rote. Incremental constructions con brio. In *Proceedings of the nineteenth annual symposium on Computational geometry*, pages 211–219. ACM, 2003.

[3] Jonathan Richard Shewchuk. *Lecture notes on Delaunay mesh generation.* Department of Electrical Engineering and Computer Science, University of California at Berkeley, 1999.

[4] Sourour Elloumi, Martine Labbé, and Yves Pochet. A new formulation and resolution method for the p-center problem. *INFORMS Journal on Computing*, 16(1):84–94, 2004.

[5] Atsuyuki Okabe, Barry Boots, Kokichi Sugihara, and Sung Nok Chiu. *Spatial tessellations: concepts and applications of Voronoi diagrams*, volume 501. John Wiley & Sons, 2009.

[6] Ling Hu, Wei-Shinn Ku, Spiridon Bakiras, and Cyrus Shahabi. Verifying spatial queries using voronoi neighbors. In *Proceedings of the 18th SIGSPATIAL International Conference on Advances in Geographic Information Systems*, pages 350–359. ACM, 2010.

[7] Prosenjit Bose and Pat Morin. Online routing in triangulations. In *Algorithms and Computation*, pages 113–122. Springer, 1999.