· U C ·

Visualisation and Analysis of Geographic Information: Algorithms and Data Structures

Master's Degree in Informatics Engineering
Dissertation

July 5, 2015

João dos Santos Valença
Department of Informatics Engineering
University of Coimbra
valenca@student.dei.uc.pt

Advisors

Prof. Dr. Luís Paquete
Department of Informatics Engineering
University of Coimbra
paquete@dei.uc.pt

Eng. Pedro Reino
Smartgeo Solutions
Lisbon, Portugal
pedro.reino@smartgeo.pt

Jury

Prof. Dr. Francisco Araújo
Department of Informatics Engineering
University of Coimbra
filipius@uc.pt

Dr. Alcides Fonseca
Department of Informatics Engineering
University of Coimbra
amaf@dei.uc.pt

Abstract

In recent years, Geographic Information Systems (*GIS*) have witnessed a large increase in data availability. There is a need to process a large amount of data before it can be managed and analysed. This project aims to develop a *GIS* application operating through a Web platform in order to allow for a low cost and simplified integration, management and manipulation of georeferenced information. Special emphasis is given to the implementation of efficient clustering algorithms for finding a representative set of points in a map. The approaches covered in this thesis include exact algorithms for solving the *k-centre* problem, as well as approximation algorithms and heuristic methods to solve the *geometric disk cover* problem. The algorithms are experimentally evaluated in a wide range of scenarios.

Keywords

Geographic Clustering, Computational Geometry Algorithms, Coverage Problems, Real-Time Applications.

Resumo

Nos últimos anos, os Sistemas de Informação Geográfica (GIS) têm assistido a um grande aumento na quantidade de dados disponíveis. De facto, existe uma necessidade de encontrar uma maneira eficiente de processar grandes quantidades de dados para que tanta informação possa ser facilmente gerida e analisada. Este projeto visa desenvolver uma aplicação GIS para uma plataforma Web, de modo a obter uma integração simples e de baixo custo que manipule e analise dados georeferenciados. Uma ênfase especial é dada à implementação de algoritmos para encontrar um conjunto representativo de pontos num mapa. As abordagens descritas nesta tese incluem algoritmos exactos para resolver o problema do *k-centre*, bem como algoritmos de aproximação e métodos heurísticos para resolver o problema de *geometric disk cover*. Os algoritmos são experimentalmente avaliados numa grande seleção de cenários.

Palavras-chave

Clustering Geográfico, Algoritmos de Geometria Computacional, Problemas de Cobertura, Aplicações em Tempo Real.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In recent years, there has been a large increase in the amount and availability of geographic data which 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 understood 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 by the user.

One obstacle when representing large amounts of geographic data is that the sheer number 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 representativeness of the larger set.

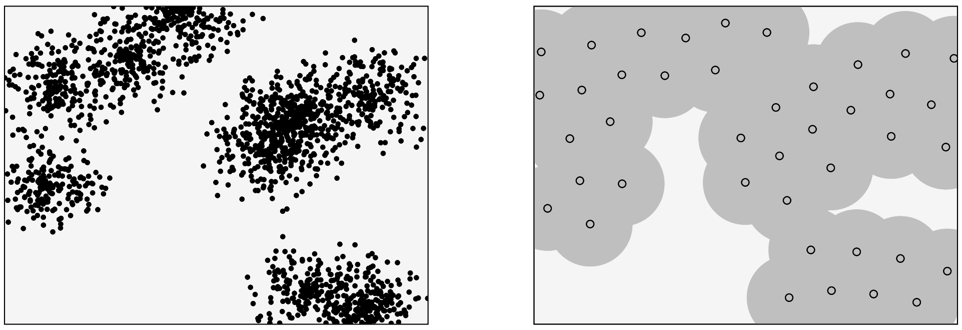


Figure 1.1. Example of a starting set (left), and a representative subset (right).

The purpose of SGP-GIMS, a QREN-funded project that involves the University of Coimbra and *Smartgeo*, is 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 must be able to aggregate and select geographic points according to a given a set of criteria. Figure 1.1 shows an example of a representative subset of an original, larger set, as well as one possibility for a representative set.

This thesis aims to design and and analyse different algorithms that allow 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. The algorithm that is deemed the most suitable to solve the task is integrated in a web framework via the *geojson* and *WFS* (web feature service) geographic data communication standards. Figure 1.2 shows the basic concept of the architecture for the web application.

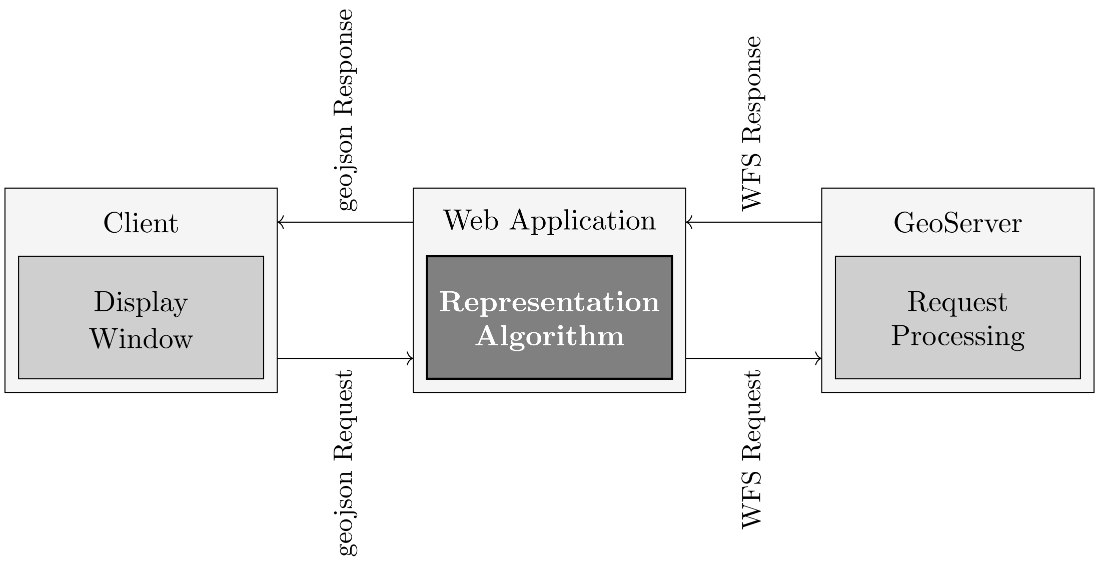


Figure 1.2. Proposed Program Architecture

The application is meant to function as an independent module capable of being decoupled and used for different clients and/or servers.

Some well-known web applications and services perform similar operations that already select points from large sets. Currently, there are two ways of handling the problem: to preprocess the data into layers of representation, and to project the points into a limited resolution image format.

Most of these applications rely on having different preprocessed layers of information, which contain similarly ranked points. For example, a map of a continent would only request the layer containing the capitals of the visible countries, whilst a map of a singular country would only request the layer containing the cities within the viewing window's coordinates. This solution requires that a lot of preprocessed data be stored in a fairly large and robust database. It also skips the representation problem by having points with different importance ranks. If any geographic query returns too many points for the application to render, then it could choose to only display the higher ranked ones or, alternatively, repeat the same query to the layer above to reduce the number of points to be rendered. Since this project requires that no storage space is used, other solutions must be found.

Another approach is done by projecting all points to a limited resolution image format. Mapping vectorial points into a bitmap format with no aliasing means that all points that are closer together than a pixel are likely be rounded off to the same spot, effectively merging the two points. Explicitly creating an image file may result in a larger file, which may cause problems in a bandwidth-dependant web application. The solution to this would be to round the coordinates of every point to a grid, and only relay the coordinates of the grid that would contain any points. This method, however, would mean a loss of precision increasing with the size of the grid. Furthermore, the points selected would be grid-aligned, and not necessarily correspond with any of the original, vectorial set. This would make for a visual pattern, which would be obvious for a human user. This solution is therefore also not suitable for our purpose.

Since none of the two approaches were suitable for the project, particular algorithms were developed to properly solve the problem. The first approach interpreted the representation problem as an optimization problem known as *k-centre* [12]. This problem consists of finding k points from a large set that optimizes a given measure of representation, namely coverage.

The cardinality constraint means that the number of representative points must be known before the algorithm is ran, which is not always possible. Because of this particular limitation, and coupled with performance issues, a second approach was considered. In this new approach, the coverage distance is defined a priori and the goal is to find the smallest set of points such that no point in the set is uncovered. This is known as a *Geometric Disk Cover* [11] problem, and our specific formulation has one extra constraint: the disks need to have their centres on points of the original set. Because of the performance issues met in the first approach, this problem was solved using an approximation algorithm, which compromises the quality of the result (within a controlled threshold) in order to be performed in a more reasonable time.

This report is organized as follows: Chapter 2 defines the base theoretical concepts, such as a notion of representativeness, as well as some useful data structures used in the algorithms. Chapter 3 describes the implicit enumeration algorithms implemented for the minimum coverage set problem, as well as an analysis on their time and space complexities. Chapter 4 describes the approximation algorithm for the geometric disk coverage problem, as well as heuristic speed-ups and a space and time complexity analysis. The chapter ends with a proposed solution to the issue of panning the viewing window. The work developed in this thesis was presented as a poster on the 18th International Conference on Geographic Information Systems (AGILE 2015) [26].

Chapter 2

Concepts, Definitions and Notation

This chapter gives an introduction to the base concepts used further in this report. The chapter starts by establishing the formal definition of the problems 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

In the context of this thesis, the representation problem 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 general distribution. As such there can be many ways to measure representation. For the purpose of this thesis, we use the definition of *coverage*.

In two dimensions, coverage measures the area occupied by the union of a collection of shapes in space. Since our method for representing points is based on the Euclidean distance, the shapes are circles, since their circumference delimits all the possible points within a smaller distance of the given radius. The points chosen to represent a larger set of points are called *centroids* as they can ultimately cover a circular area centred around them. As such, any point p is covered by centroid c if and only if it is contained in a geometric disk of a given radius centred around c.

Although the points represent geographic locations, in this thesis the metric that would measure their distance on the surface of the globe, the geodesic distance, is not used. The triangular inequality property does not apply to geodesic distances as a sphere (or an approximation of thereof) is not an Euclidean space, and it would add an unnecessary layer of complexity to computing the coverage. Because of this, in this thesis, the coordinates of the points are the planar projection of the geographic coordinates to their location counterparts, as implemented by the WFS web mapping communication standard. Therefore, the Euclidean norm is used as the spatial distance notion.

2.2 The k-centre Problem

One of coverage problems approached in this thesis is the k-centre problem. Given a set of points N in $\mathbb{R}^2$, the goal is to choose a subset P that best matches our definition of representation. The size of P, however, is constrained to a size k, which specifies how many points can 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. This notion of representation may be defined by the distance, i.e. the point p that best represents q is the closest point to q, or the one that minimises the Euclidean distance between the pair.

Finding the most representative set P in N means that every point in N is assigned to the point in P closest to it. This definition of representation is referred to as *coverage* and the points in P are called *centroids*.

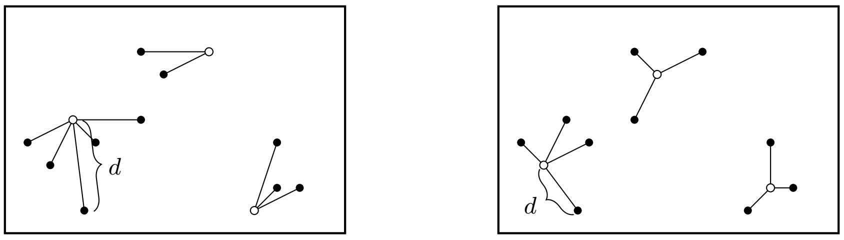
The coverage value of a given centroid is defined by the circle around that centroid with the radius defined by the distance between itself and the farthest non-centroid point assigned to it. The coverage value of a subset is determined by the highest coverage value of its points. It can thus be more 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. The most representative subset, however, is the one with the minimum value of coverage. This means all points are 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 *k-centre* problem, and is an example of a facility location problem [13]. Figure 2.1 shows two possible centroid assignments, each with its own value of coverage, d.



Non-optimal Assignment Optimal Assignment

Figure 2.1. Different assignments for the same set of points

The *k-centre* problem is a well known problem in the optimisation field of study. As such, several approaches have been explored over the years. For the 1-dimensional case, the minimum coverage value can be calculated in polynomial time [27]. However, for any other number of dimensions it is a *NP-hard* problem, and cannot be solved in polynomial time [20]. One approach to the problem is to model it in integer linear programming as follows:

$$\begin{aligned}
&\text{minimise} && D && && (2.3)\\
&\text{subject to} && \sum_{j=1}^{N} y_j = k && && (2.4)\\
& && \sum_{j=1}^{N} x_{ij} = 1 && i = 1,\dots,N && (2.5)\\
& && \sum_{j=1}^{N} d_{ij}x_{ij} \le D && i = 1,\dots,N && (2.6)\\
& && x_{ij} \le y_j && i = 1,\dots,N; j = 1,\dots,N && (2.7)\\
& && x_{ij}, y_j \in \{0,1\} && i = 1,\dots,N; j = 1,\dots,N && (2.8)
\end{aligned}$$

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 $x_{ij} = 0$ otherwise; d_{ij} is the Euclidean distance between points i and j. Constraint (2.4) ensures that k centroids are chosen. Constraint (2.5) limits the assignment of one point to more than one centroid. Constraint (2.6) ensures that all active distances are lower than the limit we are minimising. Constraint (2.7) limits points to being assigned only to centroids. Constraint (2.8) 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 as opposed to itself, as long as they are close enough together and it does not affect the coverage of the whole set. 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 is 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. [15] 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.

2.3 Geometric Disk Coverage

Another coverage problem considered in this thesis is the geometric disk coverage problem. Given a set of objects N and a fixed distance d, the goal is to minimise the cardinality of the set of disks of radius d centred around a set of centroids P contained in N, such that all points in N are covered. This is also a variation of the class of facility location problems [13], and shares some similarities with the k-centre problem, described in the previous section.

The geometric distance cover problem can be seen as a dual problem to the minimum cover subset. In fact, an Integer Linear Programming formulation can be obtained by swapping the objective function for one of the constraints:

$$\begin{aligned}
&\text{minimise} && k && && (2.9)\\
&\text{subject to} && \sum_{j=1}^{N} y_j \leq k && && (2.10)\\
& && \sum_{j=1}^{N} x_{ij} = 1 && i = 1,\dots,N && (2.11)\\
& && \sum_{j=1}^{N} d_{ij}x_{ij} \leq D && i = 1,\dots,N && (2.12)\\
& && x_{ij} \leq y_j && i = 1,\dots,N; j = 1,\dots,N && (2.13)\\
& && x_{ij}, y_j \in \{0,1\} && i = 1,\dots,N; j = 1,\dots,N && (2.14)
\end{aligned}$$

In this formulation, the value for the minimum distance D is given as a parameter and k is the value to be minimized. The value for $y_j = 1$ 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 $x_{ij} = 0$ otherwise; d_{ij} is the Euclidean distance between points i and j. Constraint (2.10) ensures that k or less centroids are chosen. Constraint (2.11) limits the assignment of one point to more than one centroid. Constraint (2.12) ensures that all active distances are lower than the limit we are minimising. Constraint (2.13) limits points to being assigned only to centroids. Constraint (2.14) defines both x_{ij} and y_j as binary variables, in order to properly represent selection and assignment.

2.4 Algorithmic Concepts

2.4.1 Branch-and-Bound

The two problems in this thesis are *NP-hard* combinatorial problems [11, 20]. One possible way of solving these problems is to use 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 bounds can be used 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.

2.4.2 Approximation Algorithms

Much like heuristic methods, approximation algorithms do not compute the optimal solution to a given problem for the sake of time efficiency. Unlike heuristics, however, these algorithms are designed to compute a solution that differs from the optimal by a given predictable factor [28].

For example, a 2-approximation algorithm for a minimising problem never reaches any solution that is more than double the optimal value for any given input. By compromising the quality of the solution, the algorithms can be run in polynomial time.

In this thesis, we explore approximation algorithms to solve the *set cover problem.* The set cover problem is a well-known combinatorial problem: given a Universe U of n elements, and sets $S_1, \ldots, S_k \subseteq U$, one must find the smallest cardinality collection of sets whose union covers U. Calculating the optimal solution to the set cover problem is *NP-hard* [11], and thus cannot be solved in polynomial time. However, by using the greedy approach of iteratively picking the set that contains the most uncovered points, it is possible to achieve an approximated solution in polynomial time. Assuming the instance of the problem has an optimal solution of m sets, the greedy algorithm guarantees that its solution is bound above by $m \log_e n$. This statement has the proof described in detail in Appendix A.2.

2.4.3 Heuristic Techniques

Optimal solution algorithms, even very optimized ones, are oftentimes still too inefficient to be used in any practical, time constrained application. One possible strategy to solve an *NP-hard* problem is to use a heuristic method to speed the process up. A heuristic is a rule or method that allows an algorithm to find a solution to problems faster than it would by calculating every option. Heuristics do this by compromising the quality of the result, and so they do not calculate the optimal solution to the problems. However, they are useful in applications where the time is a more important factor than quality.

*Local search*is a heuristic method to finding solution to *NP-hard* problems. This approach starts by finding a valid solution to a problem. Then, it tries to find similar valid solutions by slightly modifying the original solution, thus searching the local space of candidate solutions. After a local optimum is found or a time bound is elapsed, the best candidate is returned as the local solution to the problem. This method offers no quality guarantee, but are very efficient in practical implementations. Chapter 3 mentions a Local Search approach to the k-centre problem, as described by Cambazard et al. [10].

2.5 Geometric Concepts and Data 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.

2.5.1 Nearest Neighbour Search and Point Location

A common concept in computational geometry is point location. A point location algorithm finds the 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 is 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. [10]. 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.

2.5.2 *k*-Dimensional Trees

A k-dimensional tree, or k-d tree, is a space partitioning structure used for point location and nearest neighbour queries. A k-d tree is a binary tree, where each node represents an axis-aligned hyper-rectangle. Each node specifies an axis and splits the set of points based on whether their coordinate along that axis is greater than or less than a particular value [5]. The axis chosen to split each subgroup is chosen via rotation, which in the 2-dimensional space means that each level alternates between the vertical and horizontal axis.

During construction, the splitting point is chosen to be the point whose relevant coordinate best divides the group into two subgroups. The best possible choice for the splitting point at each level is the point which has the median of the relevant coordinate in the group. This ensures that each node divides the number of points in half so that the resulting tree becomes balanced, and that each point is reachable by performing $\mathcal{O}(\log n)$ operations [5]. Figure 2.2 illustrates a k-d tree partitioning a 2-dimensional space.

Constructing a k-d tree requires selecting a pivot, which divides the set into two groups: the points whose relevant coordinate is smaller than the pivot, the points whose relevant coordinate is larger than the pivot. The same process is then repeated recursively for each of the groups, alternating the relevant coordinate between both axes. Each recursive call fixes one pivot,

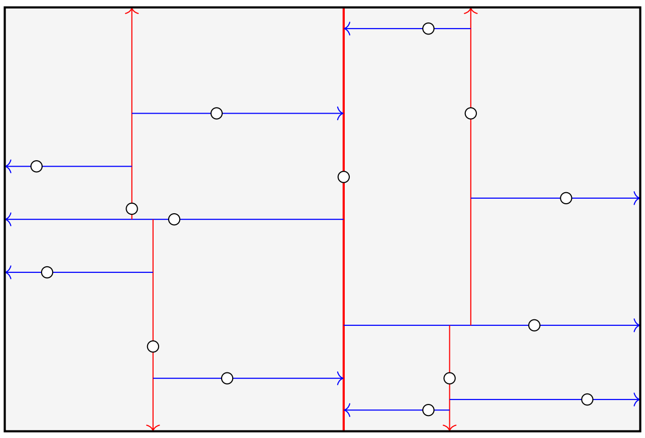


Figure 2.2. Example of a k-d Tree

which ideally contains the median value of the relevant coordinate. The time complexity of the construction of a k-d tree relies on the pivot selection function. Calculating the median usually requires sorting a list of n and then picking the $n/2$-th element. Since sorting algorithms typically take $\mathcal{O}(n \log n)$, building a k-d tree would take $\mathcal{O}(n^2)$ time. To achieve linear time complexity the median of medians algorithm described in Appendix A.1 can be used.

The median of medians algorithm does not necessarily return the actual median, which results in slightly unbalanced trees. However, the query complexity in a k-d tree constructed using this method still achieves $\mathcal{O}(\log n)$ time complexity. This occurs because the median of medians always outputs a value between the 30th and 70th percentile [6]. This guarantees that at each level of the k-d tree the group of points covered by each of the children nodes is substantially smaller than the parent node by a constant factor, and there is never a redundant node that covers the same set of points the parent does. At each level of a given query, each decision discards at least 30% of the points. This constant reduction factor maintains the $\mathcal{O}(\log n)$ time complexity for the queries.

2.5.3 Orthogonal Aligned Range Search

Some geometric algorithms require knowing which subset of points are contained in a given area. This operation is known as range search. A range search can be used to find all the neighbours of a given point that are within a fixed radius. This is most efficiently done by performing a orthogonally aligned range search on the square that encloses the range circle. Performing a orthogonally aligned range search query on a set of points can be done using a k-d tree structure or a line sweep.

k-d Tree Range Search

Performing a range search on a k-d tree can be done recursively. Each level of the k-d tree divides the plane in a different axis, alternating between horizontal and vertical divisions. Starting at the root, only the sides (left and/or right) that intersect the queried rectangle are checked for points contained inside it. Figure 2.3 illustrates a 2-dimensional k-d tree.

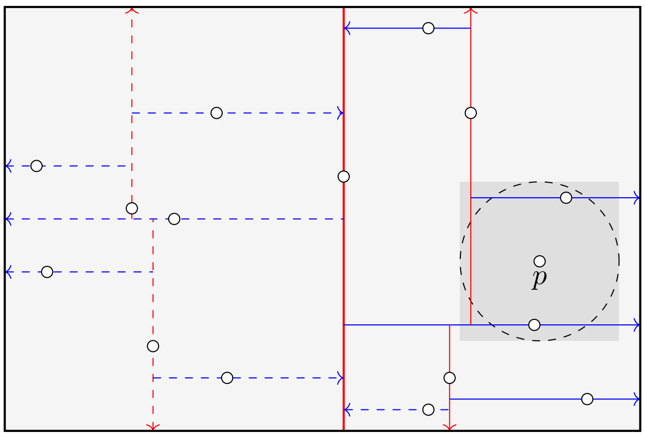


Figure 2.3. The dashed lines are never queried, since the rectangle does not intercept their parent. This query searches for all points within a fixed distance of p.

If the rectangle queried is small enough, this method eliminates most candidates from being checked, thus improving on a brute force algorithm. However, since the rectangle can be big enough to cover all points, the worst case is still the same, since returning the whole set of points never takes less than linear time to compute. The expected time complexity is given by $\mathcal{O}(\sqrt{N} + m)$ for queries in a 2-dimensional space, where m is the number of points inside the region [19].

If the rectangle to be queried is a square of side $2d$ centred around a given point p, this query can be used to limit the number of points to be tested for being within a fixed radius d around p.

Line Sweep Range Search

A Line Sweep method can also be used to find all the points in a orthogonally aligned rectangle. The line sweep algorithm starts by sorting the points on one of the coordinates, usually the x coordinate. Then, an imaginary vertical line starts sweeping each pair of points, until the distance between them is larger than the width of the rectangle.

If the rectangle to be queried is a square of side $2d$ centred around a given point p, this query can be used to limit the number of points to be tested for being within a fixed radius d around p. Doing so is a matter of performing the line sweep twice, once in each direction. A special property of this algorithm is that it can find all neighbours of all points within a distance by performing the sweep starting on every point. This method does not require both directions to be swept, since this definition of neighbourhood is mutual. So if q is a neighbour to q, then q is a neighbour to p. Each of the queries still take $\mathcal{O}(n)$ time for each point, since each swipe can contain all of the other points

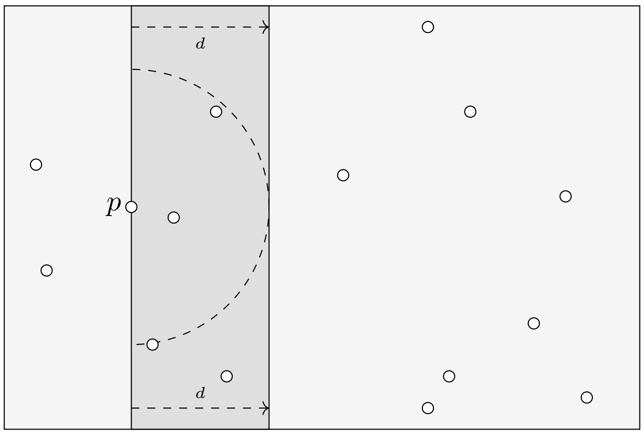


Figure 2.4. Illustration of the Line Sweep method. This query searches for the points within a given distance of p that are to the right of p

2.5.4 Voronoi Diagrams

Voronoi diagrams [21] 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 consider the $\mathbb{R}^2$ plane and the Euclidean distance. Figure 2.5 shows a partitioning of a plane using a Voronoi Diagram for a set of points: Dashed lines extend to

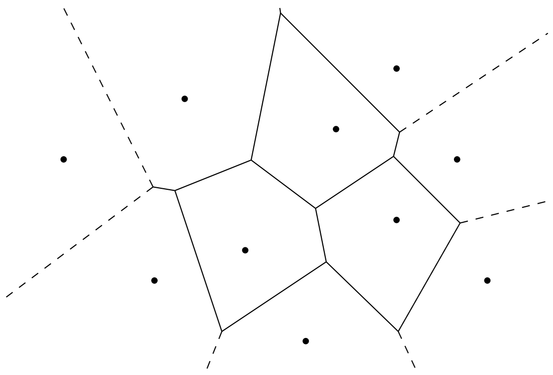


Figure 2.5. Example of a Voronoi Diagram

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.

2.5.5 Delaunay Triangulations

Another useful structure for geometric algorithms is the Delaunay triangulation [21]. A Delaunay triangulation [14] 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 2.6 shows the Delaunay triangulation of the same set of points used in Figure 2.5. More importantly, the Delaunay triangulation of a

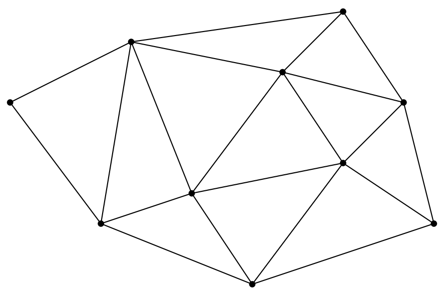


Figure 2.6. Example of a Delaunay Triangulation

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 2.7 shows the overlapping of the Voronoi diagram in Figure 2.5 and the Delaunay triangulation in Figure 2.6. The Delaunay edges, in black, connect the points at the centre of the Voronoi cells, with edges in blue, to their neighbours. Unlike its

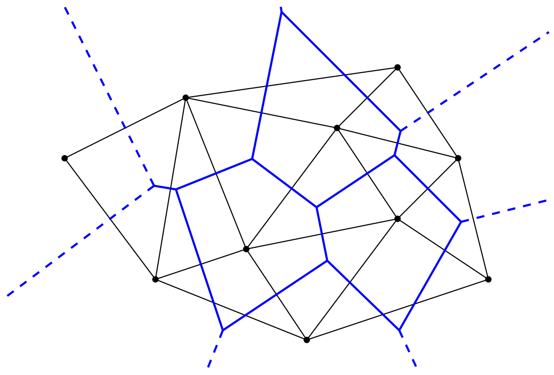


Figure 2.7. Overlap of a Voronoi Diagram and its Delaunay Triangulation

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.

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

Algorithm 2.1. Bowyer-Watson Algorithm

```
1: function 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: function 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($a, b, c$)
```

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 [9].

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 2.1 for each vertex v to be included in the triangulation.

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 (calculated with the *InCircle* function), violates the Delaunay rule, and must also be deleted and have its sides recursively checked, until no adjacent triangles violate the Delaunay rule. Whenever the *DigCavity* function reaches a set of three points whose circumcircle does not contain v, it creates the triangle by creating counter-clockwise half-edges between those three points (*AddTriangle*). 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 [24].

Deconstruction

Deconstructing a Delaunay triangulation usually consists of reversing the construction algorithm to remove points from the triangulation. However, as we explain in a later chapter, in our case the deconstruction has to be incremental. Since the first point to be removed 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 performed, and the old edges and triangles are quickly put back in place, with no new memory allocation needed.

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, connecting 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 2.8 further illustrates the concept of the half-edges.

This structure makes it easier to store the changes to the triangulation at each step, since they

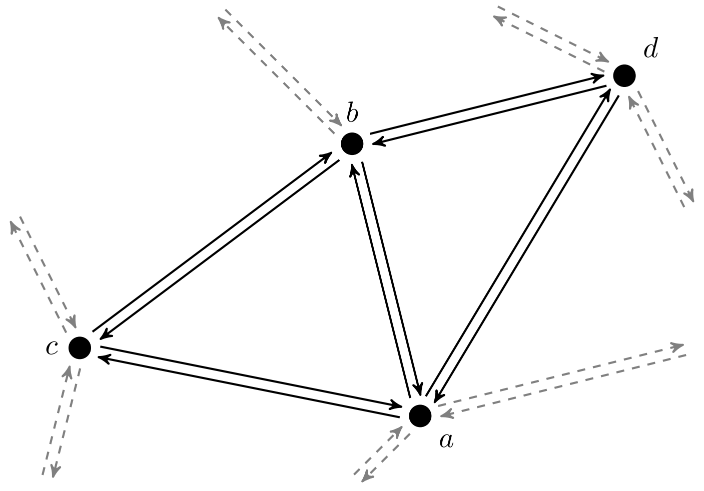


Figure 2.8. Illustration of the Half-Edge Structure

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 it keeps all the edges starting at any given point easily accessible. All neighbours to any point v are the end points to the half-edges starting at v. This property is useful when efficiently implementing the greedy routing algorithm described in the following Section.

Greedy Routing

In order to quickly calculate the nearest neighbour to a point in a set, one can make use of the Delaunay triangulation with Greedy Routing [8]. Consider a triangulation $\mathcal{T}$. In order to find the closest vertex in $\mathcal{T}$ to a new point p, 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 a point w is reached such that no neighbours of w are closer to p than w is, the closest point to p in $\mathcal{T}$ has been found. In the following, we show that the greedy routing algorithm is correct: In Figure 2.9, the search for the closest vertex to

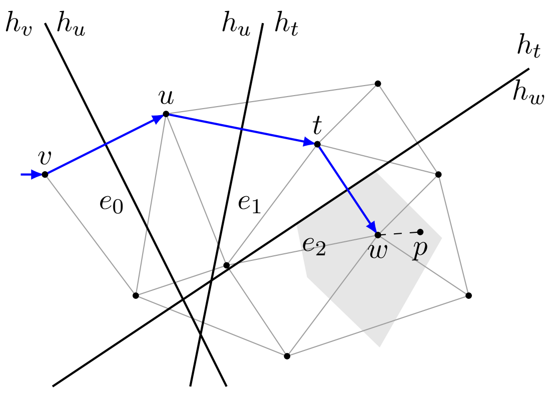


Figure 2.9. Example of the Greedy Routing Algorithm

p starts at point v. From there, point u, which is closer to p than v is, is found. The step is repeated, following the blue path until point w is reached. Since no neighbour of w is closer to p than w is, then p must be within the Voronoi cell of point w (shaded light grey).

Theorem 2.1. *[8] 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 any neighbour 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. $\square$

Line Walking

Another point location algorithm to consider is the line walking algorithm [4]. 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 triangle 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. [4], and is illustrated in the following figure,

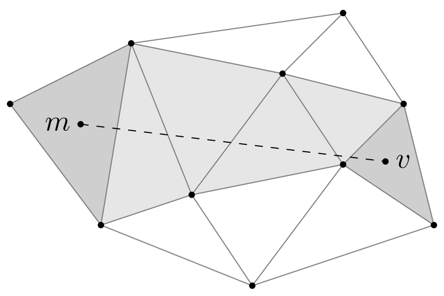


Figure 2.10. Illustration of the Walking Algorithm

where the dark shaded triangles represent the starting and finishing triangles, and the light shaded triangles the path the algorithm takes to find the final triangle that contains the vertex v.

After finding this triangle, the Bowyer-Watson algorithm described in Section 2.5.5 can be used to update the new triangulation, which now includes v.

2.5.6 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. To fully take advantage of these approaches, the points should be sorted is such a way that the distance between consecutive points is minimised.

Hilbert curves are a kind of fractal space-filling curves [23] 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 2.11 demonstrates the first few orders of approximation: Since the coordinates of the points in

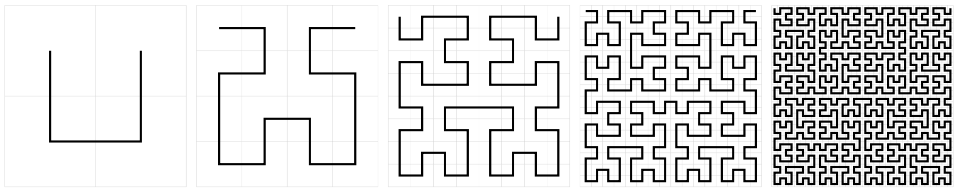


Figure 2.11. First Five Orders of Hilbert Curve Approximation

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. Sorting an array of 2-dimensional points to follow a Hilbert approximation, each point should be assigned the 1-dimensional coordinate of the square tile containing it. The array is then sorted using the square coordinates along the Hilbert approximation as a key. There are cases where points share the same discrete approximation coordinates, but this has little effect on the performance of the point location algorithms, providing 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 containing all points.

Chapter 3

Exact Algorithms for the k-Centre Problem

This chapter describes two possible algorithms that solve the *k-centre* problem. The goal of this problem is to, given a set N of n points, find the subset of centroids of cardinality k that minimises the farthest distance of a point in N to its closest centroid.

Both algorithms use an incremental branch-and-bound approach for implicit enumeration. The first algorithm is a naïve implementation of a branch-and-bound algorithm and uses simple loops over arrays for point location queries. The second algorithm builds and uses Delaunay triangulations to achieve more efficient point location queries. At the end of this chapter, these algorithms, as well as an implementation of the Integer Linear Programming formulation described in Section 2.2, are experimentally tested in a set of benchmark instances.

3.1 Naïve Branch-and-Bound

One possible approach to this problem is to use a branch-and-bound method. Our approach is based on the wotk of Cambazard et al. [10] for solving a related k-centre problem. Their method describes algorithms to incrementally insert and remove centroids from a set of points, and update the centroid assignment only in the geometrical area surrounding the changed point. This method allows for small modifications on an already valid solution, and the techniques for updating the objective function upon insertion and removal of points can be used in an incremental branch-and-bound method to speed up the computation between branches, without having to calculate the full point assignment for every iteration of the algorithm, whilst still implicitly enumerating all possible combinations.

To solve the problem incrementally, at each step of the recursive tree, one of the available 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 and a solution is found at the end of the branch. The best solution in all the branches is the optimal value for the *k-centre* problem.

3.1.1 Branching

As stated above, branching the tree involves updating the assignment between new points and/or new centroids, as well as updating the objective function. The following procedures 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. Figure 3.1 illustrates this operation.

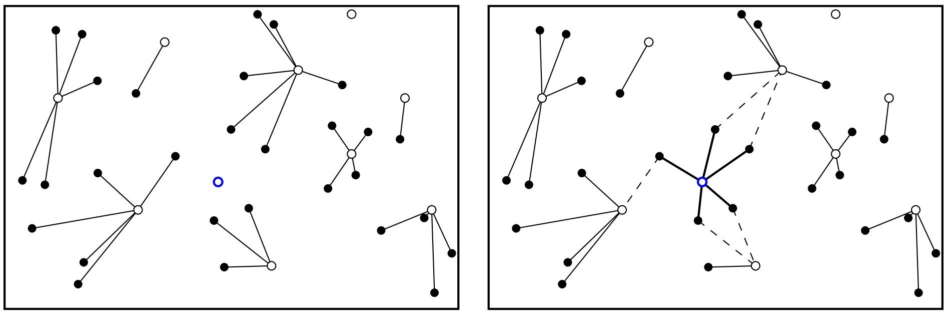


Figure 3.1. Illustration of a centroid insertion. The new point, in blue, changes the surrounding points' assignments, dashed, to connect to the new closest centroid.

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 either decreases or maintains the current value. This step compares the distance between point n and all centroids, taking $\Theta(k)$ time. Figure 3.2 illustrates this step.

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

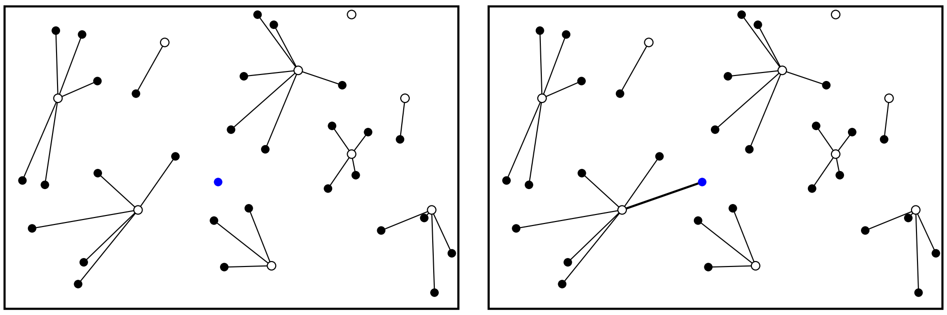


Figure 3.2. Illustration of a non-centroid insertion. The new non-centroid, in blue, is assigned to the closest centroid.

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 increases, 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. Figure 3.3 illustrates this step.

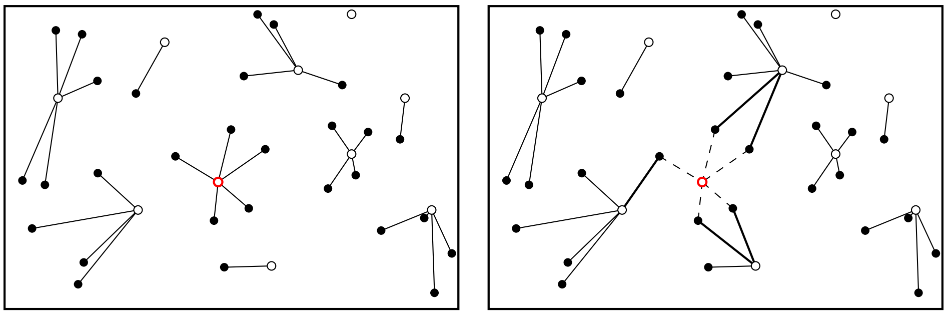


Figure 3.3. Illustration of a centroid removal. The old assignments, in thicker lines, are restored after removing the centroid, in red.

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 can be done in $\mathcal{O}(1)$ time at the expense of additional $\Theta(1)$ memory space. Figure 3.4 illustrates this step.

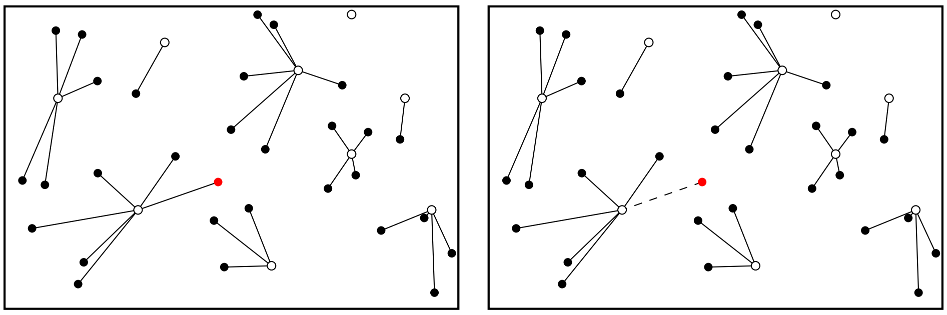


Figure 3.4. Illustration of a non-centroid removal.

3.1.2 Bounds

At all steps in the branching, the lower bound for the value of the objective function in the current branches is calculated. If the lower bound is larger than an already calculated upper bound, 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 are 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 improves the current solution in the current branch, and the branch can be pruned. Figure 3.5 illustrates the computation of the bound in one branch of the recursion.

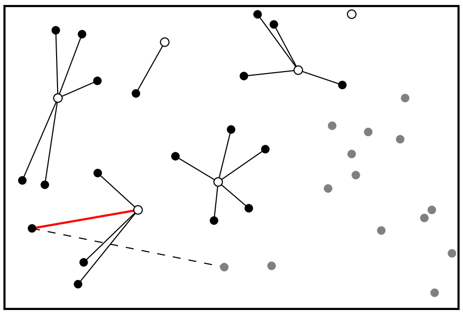


Figure 3.5. Illustration of the lower bound calculation. The dashed line represents the value for the lower bound. Since the value is larger than the current objective function, in red, if there has been found a branch with a better solution, then the current branch is pruned.

3.2 Delaunay Assisted Branch-and-Bound

Most of the operations in the branch-and-bound approach described in Section 3.1 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 2.5.5 takes an estimated $\mathcal{O}(\log K)$ for a uniformly distributed set of points [24]. 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 into 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 six [17, 21], 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. Figure 3.6 illustrates this operation.

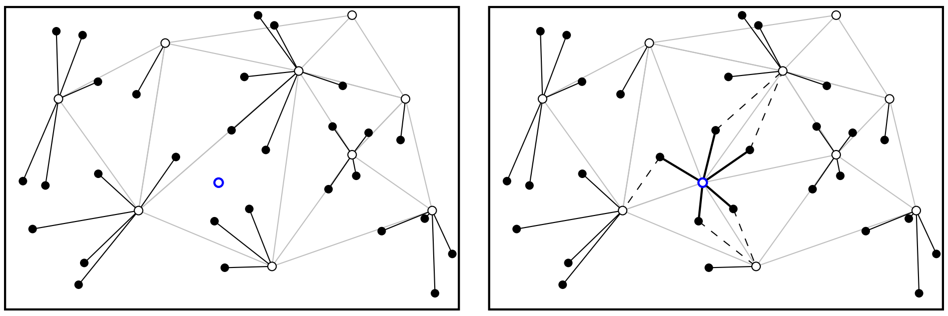


Figure 3.6. Illustration of a centroid insertion using the geometric method. The Delaunay triangulation is updated and only the neighbours need to be checked for assignment updates.

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 [8]. 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 is close to its respective optimally positioned centroid c, and it only needs 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 six neighbours for each centroid means that the expected time for a query starting at the right centroid would be $\mathcal{O}(1)$. This represents the best case scenario, and 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 points *should* behave closer to $\mathcal{O}(\sqrt{K})$ time per point. If a rectangular area has N points, the longest path would be a diagonal. The diagonal, like the sides, has close to $\sqrt{N}$ number of points, in an area with a sufficiently good uniformity of points in it. Figure 3.7 illustrates this operation.

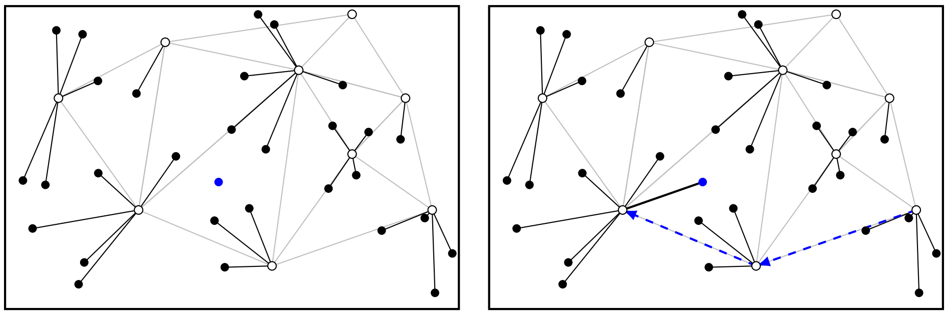


Figure 3.7. Illustration of a non-centroid insertion using the geometric method. The greedy routing is used to limit the number of points to be checked to be the closest centroid.

Removing a Centroid Removing a centroid c means removing it from the Delaunay triangulation and redistributing all points assigned to c across its neighbours. Figure 3.8 illustrates this operation. Since all points are inserted in the triangulation in a LIFO 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 [24], and removing it takes 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. However, using a stack limits the number of operations to only those that changed upon insertion, which in an uniform distribution, means an expected time complexity of $\mathcal{O}(N/K)$.

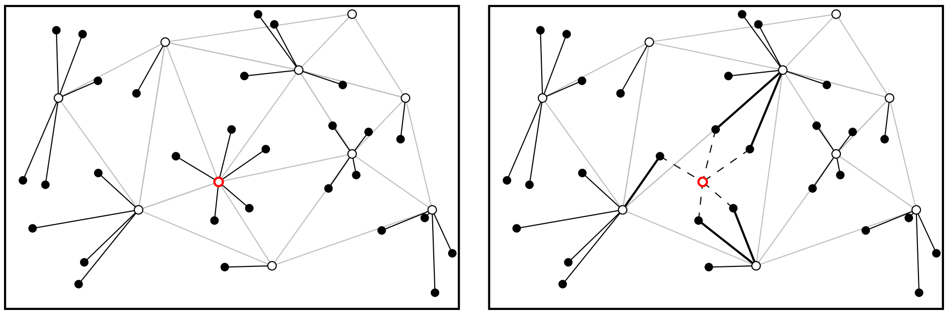


Figure 3.8. Illustration of a centroid removal using the geometric method. This operation is done by retrieving the previous state of the triangulation. Only the points covered by neighbour points of the removed centroid, in red, need to be considered for the assignment update.

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. Figure 3.9 illustrates this operation.

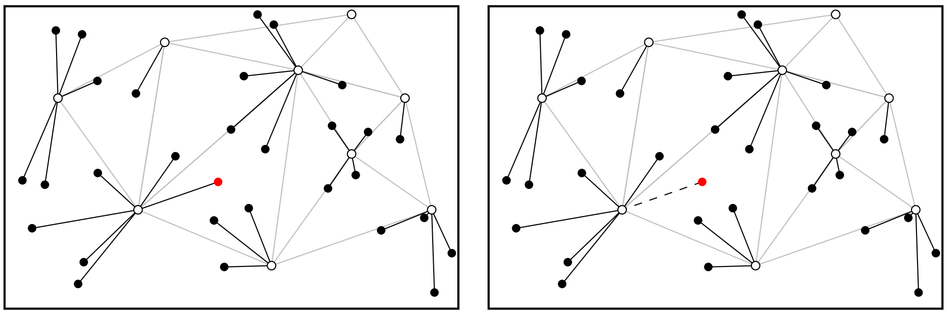


Figure 3.9. Illustration of a non-centroid removal using the geometric method.

Bounds The same lower bound described in Section 3.1.2 can be applied in this approach. Both algorithms have the same time complexity of $\mathcal{O}(N)$ for computing the bound.

These steps occur at each iteration of the branch-and-bound algorithm, and each is performed potentially 2^N times for both approaches. Despite having the same worst-case time complexity as the branch-and-bound algorithm described in Section 3.1, the expected time complexity for the Delaunay assisted approach is smaller. This approach should have better performance when a large number 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 has a negative impact in the performance in the smaller instances of the problem.

3.3 Algorithm Comparison

3.3.1 Time Complexity

Each of the procedures mentioned in the previous Section are performed at each step of the recursive tree. At each step the bound is also calculated, which takes $\mathcal{O}(N)$ time. Table 3.1 shows the time complexities for each procedure in both algorithms. The values presented at the row corresponding to the average case of the Delaunay-assisted algorithm are conjectured and need to be shown in a more formal manner.

3.3.2 Experimental Results

In this section, we analyse empirically the time spent calculating the solutions to different sizes of the problem.

Algorithm	Insert		Remove	
	Centroid	Non-Centroid	Centroid	Non-Centroid
Naïve BB	$\Theta(N)$	$\Theta(K)$	$\Theta(N)$	$\mathcal{O}(1)$
Del. Assisted BB Average Case	$\mathcal{O}(\log K + N/K)$	$\mathcal{O}(\sqrt{K})$	$\mathcal{O}(N/K)$	$\mathcal{O}(1)$
Del. Assisted BB Worst Case	$\mathcal{O}(K + N)$	$\mathcal{O}(K)$	$\mathcal{O}(N)$	$\mathcal{O}(1)$

Table 3.1. Time complexities for the various operations in a uniformly distributed point set

Methodology and Set-up

The test cases are sets of uniformly distributed points generated with a fixed seed. Each test was repeated 10 times with different sets of points. The same sets and machine were used to test the three algorithms.

Both branch-and-bound approaches were implemented using *C++* and compiled using *g++ 4.9.2*. The integer linear programming version had the data preprocessed using *python 2.7.9* and was solved using the *GLPK LP/MIP v4.55* solver. The programs were ran on a machine with a Intel i7 Dual-Core, 2GHz processor, with a 8 GB, 1600 MHz memory and Arch Linux 3.14.4 x64 as its operating system.

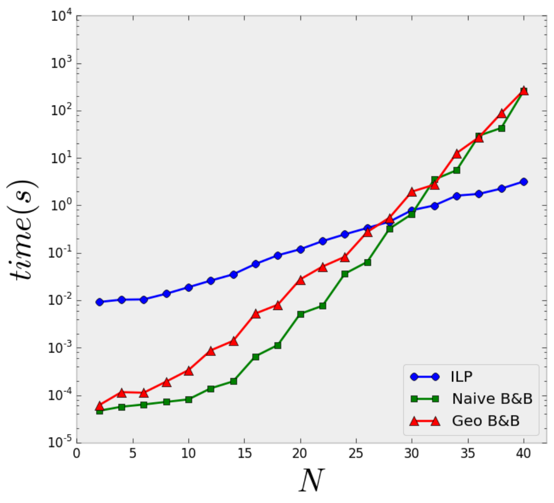
Effect of N

The first test conducted analysed the variance in performance relative to changes in the value of N. It was done by changing N, with K taking fixed fractional values of N. Figure 3.10 shows the results of these tests. The measures were taken in seconds and account for the pre-processing steps and the solving time, but not the input reading or output writing times. The tests were stopped at the half-hour mark.

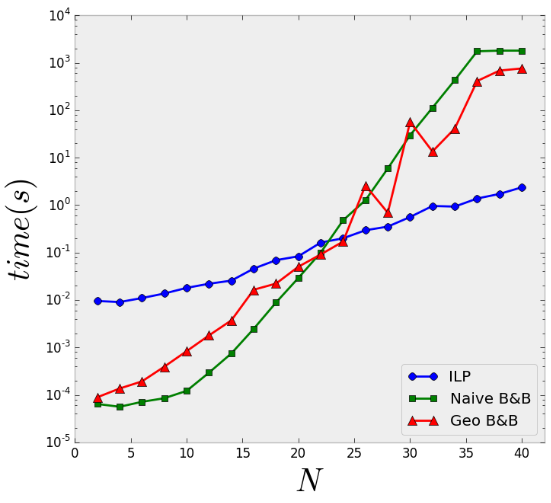
As it can be seen, the problem solving time increases exponentially with the value of N, as expected. The Integer Linear Programming Approach performs faster for larger values of N. Comparing the branch-and-bound approaches, these tests show that the Delaunay-assisted algorithm steadily approaches and surpasses the naïve branch-and-bound algorithm as the instance size grows.

Effect of K

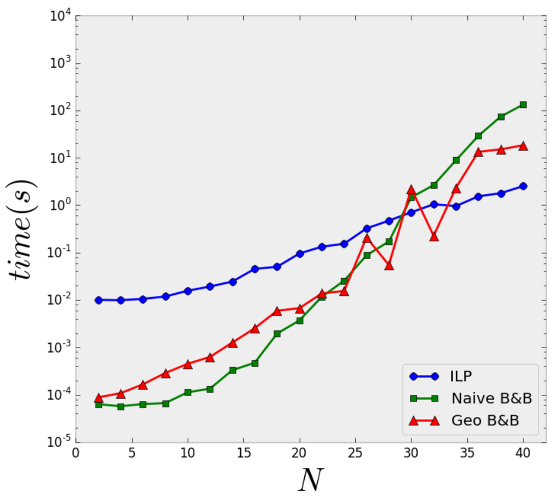
In this experiment, we analysed the performance of the three algorithms in dependence of parameter K. We fixed N and varied K from 2 to N by steps of 2. Figure 3.11 show the results for the $N = \{10, 20, 30, 40\}$, respectively. The tests were stopped at the half-hour mark. Figure 3.11 shows the results of these tests.



(a) $K = 0.25N$



(b) $K = 0.5N$

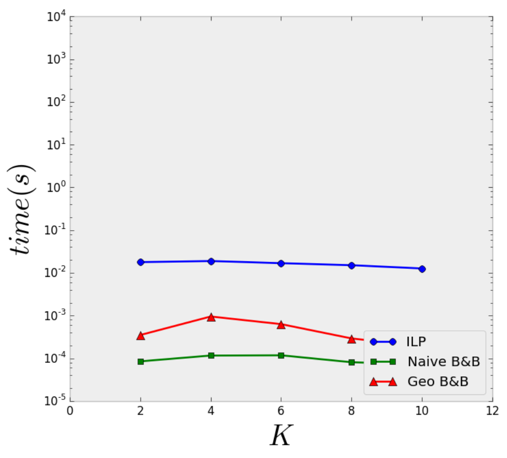


(c) $K = 0.75N$

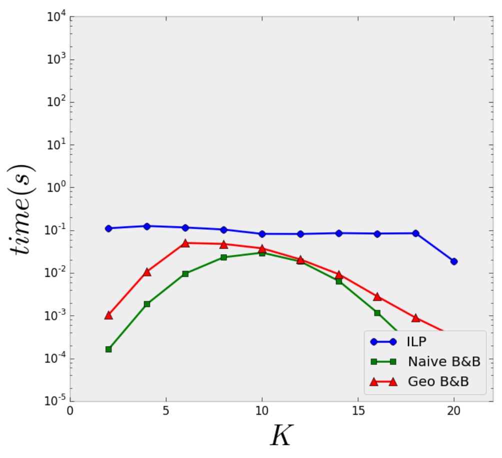
● – Integer Linear Programming ▲ – Delaunay Assisted B&B ■ – Naive B&B

Figure 3.10. CPU-time for different values of K with varying values of N

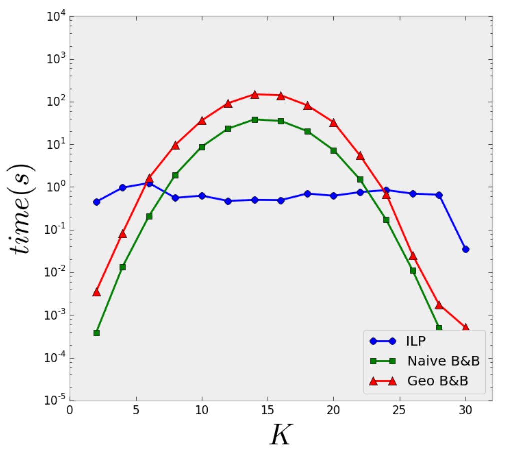
The Integer Linear Programming approach seems to be the fastest approach for most cases, and seems independent to the value of K. The exceptions to this seem to be smaller values of N, as well as the smallest and largest values of K. This happens because the implicit enumeration methods only need to enumerate a very small number of combinations. As for the branch-and-bound algorithms, the Delaunay-assisted approach is slower than the Naïve implementation in these test cases. However it should be noted that for each N, the Delaunay-assisted algorithm peaks before the expected value of $K = N/2$. This is justified due to the fact that the Delaunay triangulation has an overhead which can take advantage for in larger values of K. Furthermore, the fact that the Naïve implementation had no test for the middle values of K for $N = 40$



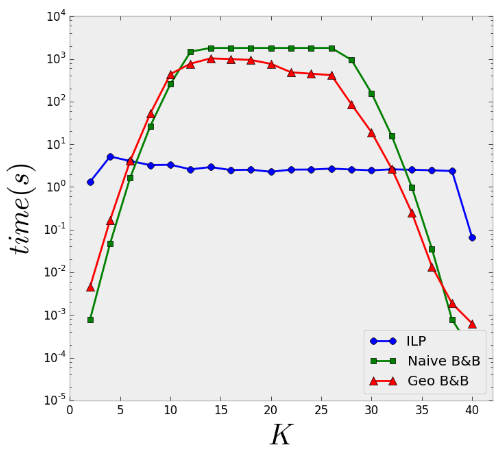
(a) $N = 10$



(b) $N = 20$



(c) $N = 30$



(d) $N = 40$

● – Integer Linear Programming ▲ – Delaunay Assisted B&B ■ – Naive B&B

Figure 3.11. CPU-time for different values of N with varying values of K

that ended before the time-out mark is noteworthy. It is also worth noting that The Delaunay-assisted approach showed a lot more variance between tests, often taking much lower values than the mean. However, for the tests performed, two runs had values much larger than the Naïve approach, approximating the Delaunay-assisted algorithm's mean to the Naïve approach.

The time required for each test limited the number of tests performed. Because of this, the results may not be statistically meaningful. This could mean that the Delaunay-assisted approach is only preferable for values of N and K to which neither approach is usable in real-time. Due to the small number of tests for large values of N, this result may not be statistically meaningful, but it is noteworthy.

3.4 Discussion

The algorithms in this chapter have some drawbacks when used in a web application. Not only were the running times too large to be used in real-time, taking minutes and even hours for small instances with 40 points at most, but they also required some a priori knowledge about the number of clusters on a given window. This is infeasible, since there is no efficient way to infer how many clusters there are in a new region, without testing for all possible values of K. The algorithm should be able to calculate the final number of selected points, constrained to a minimum distance factor between them. This can be achieved by solving *Geometric Disk Cover* problem described in Section 2.3. The next chapter describes in detail a way to solve this problem in regards to the final application.

Chapter 4

Geometric Disk Cover

The goal of the geometric disk cover problem is to, given a set N of points and a minimum distance d, find the smallest number k of circles of radius d centred around points in N such that no point is left uncovered. This chapter describes an efficient way to approximate the optimal solution to this new problem. The number of points selected by the algorithm is bound above by $m \ln n$ points, where m is the optimal value of k.

4.1 Approximation Algorithm

Calculating the optimal solution for Geometric Disk Cover is a *NP-hard* problem [11]. However, there are ways of finding a reasonable approximation to the optimal solution in a short amount of time. Briefly, this approach requires two steps. The first step is the Proximity Graph Building step, which builds a graph connecting all pairs of points that are close together. The second step is a Set Cover step, which uses an approximation algorithm to cover the graph built in the former step. The following sections describe these steps in more detail.

4.1.1 Proximity Graph Construction

The first step is to create a graph connecting all pairs points that are within a distance of each other. This means constructing a graph in which every point is a vertex, and every edge connects two vertices whose points are within the given distance, also known as a proximity graph [18]. The naïve approach to building this graph would be to test all the distances between each pair of points, taking $\mathcal{O}(n^2)$ operations. Alternatively a line sweep algorithm or a series of range searches on a k-d tree could be used to speed up the process. The line sweep algorithm would require a $\mathcal{O}(n \log n)$ sorting algorithm, followed by $\mathcal{O}(n^2)$ comparisons. However, the latter is limited by the number of points within the sliding window, which should only contain a fraction of the total number of points on the map, since the distance chosen is a fraction of its dimensions. The k-d tree range search, on the other hand, requires a $\mathcal{O}(n \log n)$ construction of a k-d tree using a median of medians algorithm. This is followed by n queries consisting of finding the points within a square of side $2d$ centred around each of the points. Each one of these operations take $\mathcal{O}(\sqrt{N} + m)$ time, where m is the number of returned points [19]. Even though the complexity of the k-d tree range search is theoretically faster than the line sweep method, it comes with an extra overhead of handling a more complex structure. As such, both methods are analysed from an experimental point of view in this thesis. After this operation, the graph connecting all neighbours is built. Figure 4.1 illustrates one such graph.

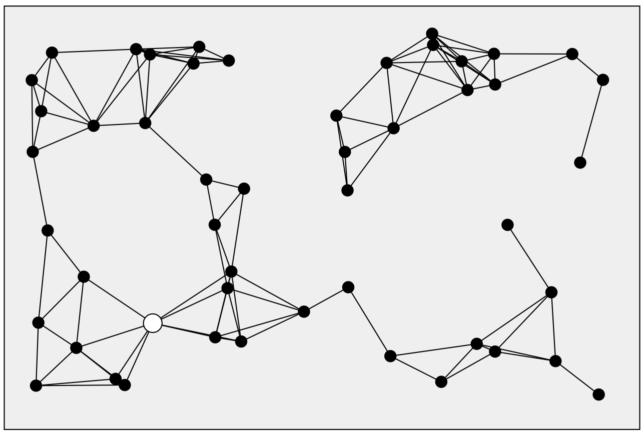


Figure 4.1. Illustration of the proximity graph. The white circle represents the point with the largest number of neighbours.

This graph is represented via adjacency lists. The adjacency lists have an advantage over an adjacency matrix as linked lists can be used to considerably reduce the memory footprint in sparse graphs. It also allows for faster sequential access to the neighbours of any given point. These linked lists share similarities to the half-edge structure, in which each node represents a unidirectional edge that contains a pointer to its counterpart.

It is noteworthy that the graph can reach $\mathcal{O}(n^2)$ edges if the points are all very close together and/or the minimum distance is large enough. This case coupled with the fact that the adjacency linked lists occupy more space per edge than an adjacency matrix, the space needed to keep the graph in memory can be potentially too high for some machines to handle larger instances of the problem.

4.1.2 Set Cover

The second step to the algorithm is to chose a small subset of vertices whose neighbours unions are equal to the whole set of graph's vertices. This is known as *set cover*, and its solution can be approximated using a greedy approach [28]. Starting with the whole uncovered graph, the point with the largest number of uncovered neighbours is selected. This is done iteratively until no uncovered points remain on the graph. This approach approximates the number of points to within $m \ln n$ points, were m is the optimal number.

At each step of this algorithm, one point p and all its neighbours and respective edges must be removed from the graph. This is done by iterating through the adjacency lists of the neighbours of p and deleting all the connections to their neighbours (second-degree neighbours of p). This operation takes exactly $\mathcal{O}(n)$ time where n is proportional to the number of edges to be deleted. Figure 4.2 illustrates the graph after the first iteration:

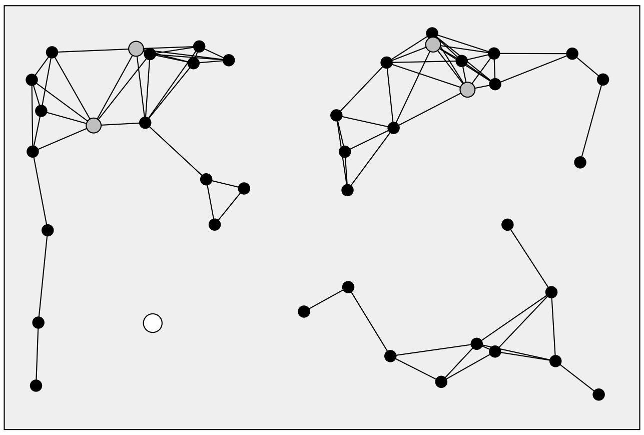


Figure 4.2. Illustration of the state of the proximity graph after the first iteration. Any of the gray points is to be removed in the next iteration, as they have the largest amount of neighbours.

After all the points are covered, the collection of selected centres makes the subset of centroids that is displayed as the final output. The cardinality of this set does not exceed the approximation as described above.

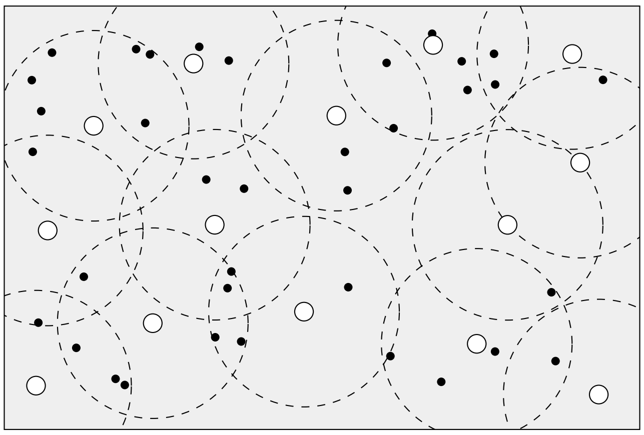
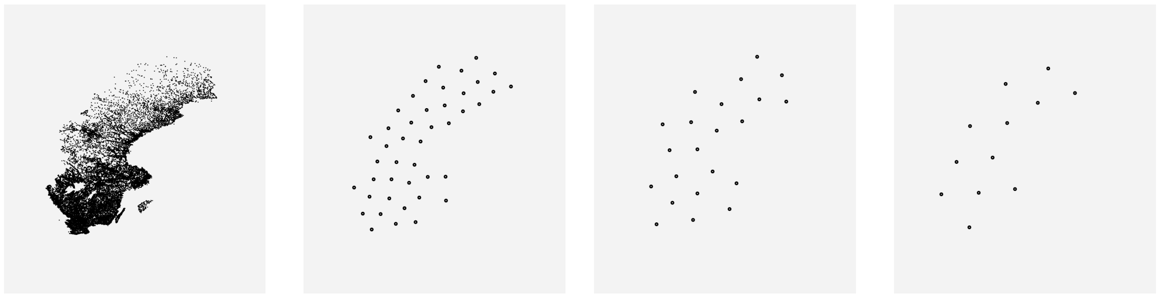


Figure 4.3. Illustration of the state of the final chosen set. The green points are the final representative set.

4.1.3 Results

This section reports both approaches (k-d trees and line sweeping) to the graph building portion of the algorithm. The implementations differ from each other only in the range search algorithm they use to build the proximity graph. This is done because the graph building stage has the highest time complexity, and is therefore expected to be the bottleneck. In fact, running a code profiler shows that the graph building stage takes 98% of the CPU-time, with the time taken to run the set cover algorithm stage being negligible.

For the tests in this section, the given value of d is the minimum distance between points, therefore radius of the covering disks. It is given as a percentage of the largest dimension of the window, as this value is easier to adjust and analyse by a human user. Figure 4.4 shows the outputs for different values of d for a set of points of interest in Sweden.



(a) Original Set **(b)** $d = 10\%$ **(c)** $d = 15\%$ **(d)** $d = 20\%$

Figure 4.4. Selected subsets from both graph building algorithms. The points are selected starting from green to red. The data was obtained from the University of Waterloo, Ontario, Canada[2].

By analysing images of a few real-world examples, the values for the minimum distance d were agreed to be given as a value between 10% and 15% of the largest dimension of the window. These values give the subsets that were deemed as the easiest to look at without losing sense of shape of the area. The performance tests still use the value of 20% to test the algorithms under less favourable conditions, since the larger the value of d, the denser and more time is required to build the proximity graph.

Since the algorithms in this chapter are more efficient than the ones in the previous chapter, new tests were needed to benchmark them. These tests use random inputs to benchmark the performance of the algorithms. Both uniform and clustered inputs are tested. The uniform inputs are a simple collection of N points whose Cartesian coordinates are randomly chosen between 0 and 100. The clustered inputs are generated by choosing a random number of points s between 10 to 20 points. Each of these points acts as a centre and is chosen by randomly generating two Cartesian coordinates between 0 and 100. Around each of these, N/s points were generated by randomly choosing an angle Θ (between 0 and 2π) and a distance ρ (between 10 and 20), which represent the polar coordinates around their respective central point. These tests generate circular clusters, with more density towards the centre.

Each test is performed 30 times, with shared seeds between the algorithms. The times listed do include the input scanning. All algorithms are implemented using ANSI C89 and compiled using *gcc 5.1.0*. The programs were ran on a machine with a Intel i7 Dual-Core, 2GHz processor, with a 8 GB, 1600 MHz memory and Arch Linux 3.14.4 x64 as its operating system.

The first test performed benchmarked both line sweep and k-d tree range search algorithms for the proximity graph construction. The results are shown in Figure 4.5.

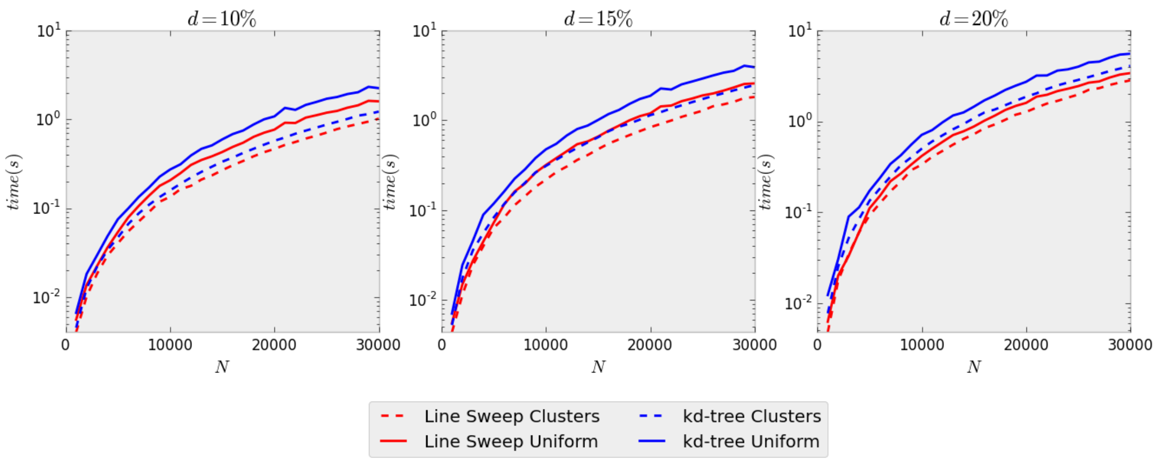


Figure 4.5. CPU-time for both proximity graph algorithms on uniform and clustered inputs for different values of d.

As Figure 4.5 shows, the line sweep method performs faster in both the uniform and the clustered data. The lower expected complexity of the k-d trees does not compensate the larger overhead in their construction. It can also be seen that both algorithms become slower the larger the number of points, as was expected.

Building a proximity graph with 30000 vertices has the potential to need 30000^2 edges, making these algorithms very memory intensive. Even with adjacency lists instead of an adjacency matrix, the inputs can generate a instance where every point must be connected to every other point. In this case, the advantage of the adjacency lists for sparse graphs is lost, and the program exceeds the RAM limit given by the Operating system. This occurs more frequently for larger values of d, and N, with the smallest case recorded being of $N = 25000$ and $d = 0.2$ on a clustered input that generated most clusters on top of each other, concentrating most points within a circle of a small radius, where most of them had over 20000 neighbours. However this case is somewhat rare only occurs under very specific circumstances and did not occur during the tests displayed. Figure 4.6 shows the space used by both algorithms on the tests performed.

The lines in Figure 4.6 overlap, since applying both methods to any given case gives the same proximity graph, as it is unique. The final set, however, may not be the same. The coverage

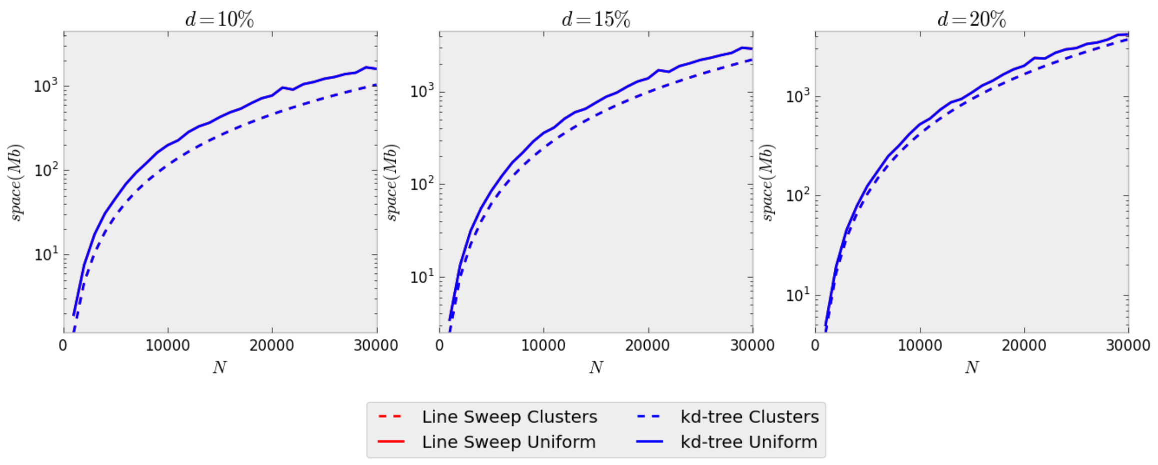


Figure 4.6. Memory used by both proximity graph algorithms on uniform and clustered inputs for different values of d. Note that the lines overlap, as both algorithms build the same graph.

algorithm picks the points with most neighbours. Since two different points may have the same number of neighbours, and the two algorithms sort the points in two different ways, there may occur a case where the sets are completely different, based on which point is selected first in case of a draw. The values still fall bellow the upper bound for the approximation factor, and no algorithm should have the advantage over the other. Figure 4.7 shows the number of points selected by each algorithm in the same instances as above.

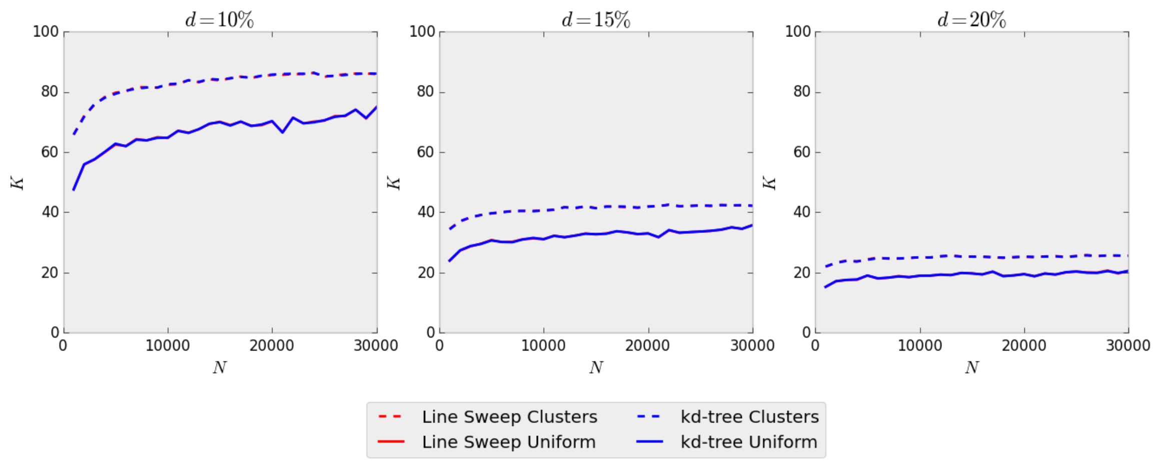


Figure 4.7. Number k of points selected for both proximity graph algorithms on uniform and clustered inputs for different values of d. Note that the lines overlap, since both algorithms are expected to output the same set most of the time.

Figure 4.7 shows that the number of selected points k does not seem to vary very much with N for the same values of d. In fact, the growth of the value of k seems to suggest an asymptotic approach to a fixed value. This happens because there is a limit to how many circles can be placed in an area without any of them containing the centres of the others.

The process of finding the best disposition of circles of the same radius in a given area is known as circle packing. The number of circles in a circle packing instance for a square area is given by the ratio between the circles' radii and the width of the square enclosing them. Since the value for d is given as a fraction of the size of the window, the bound values of k can be calculated for the various values of d. A detailed explanation of how to calculate these values is given in Appendix A.3. The values indicate that k should not be larger than $\{128, 59, 36\}$, for the given values $d = \{0.1, 0.15, 0.2\}$, respectively. With the exception of the latter, these values are yet to be proven to be optimal, but are not expected to be very different, and serve as a good estimate of the maximum number of points that can be selected.

Figure 4.7 also shows that no algorithm has the advantage on the number of points chosen. In fact, the results overlap each other almost completely. This suggests that the best algorithm to use is the line sweep algorithm, as it takes less time and the same space to compute similar results.

4.2 Heuristic Speed-ups

The approximation algorithm runs rather efficiently CPU-time-wise, taking under 1 second for 30000 points for the uniform inputs, and under 3 seconds for the clustered inputs. Nonetheless, they are very memory intensive, which could present an issue for larger instances. This result can be improved by employing some heuristic filtering methods to the inputs. By using these approaches, however, the guarantee of the quality of the approximation is lost. Nevertheless, the results may indicate if the heuristic algorithms should still be considered to handle larger numbers of points.

4.2.1 Random Sampling

The simplest method to speed-up the algorithms is to simply ignore a randomly chosen subset of those points. With the smaller sample of points, the algorithm should run faster and use less memory space. If the points are removed uniformly, then the overall distribution should still be kept. Two versions of this method were tested. In the first, only one in every five points was considered by the algorithm, whereas in the second only one in every ten were. Figure 4.8 shows the time difference between the regular line sweep and the randomly sampled input algorithms.

The graph in Figure 4.8 proves that the sampling is indeed faster. This speed-up is caused by the lower number of starting points for the algorithm to process. The smaller number also affects the memory needed by these algorithms, which require less space to store the graph. This happens because the number of edges tends to grow quadratically in relation to the number of points, meaning that a set with 1/5th of the points would be expected to have around 1/25th of the edges. Figure 4.9 shows the space used by the programs. This time, no cases that exceeded

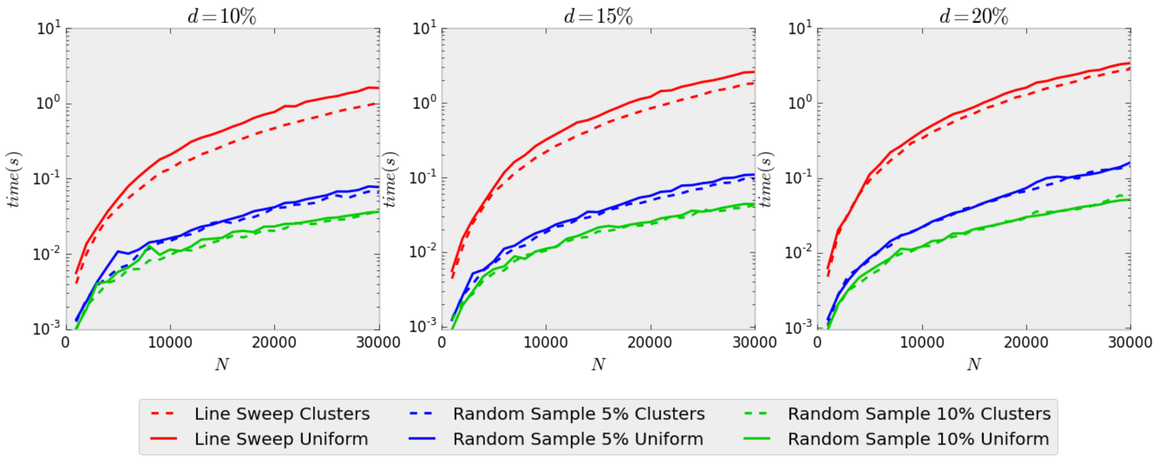


Figure 4.8. CPU-time for the line sweep algorithm, and two instances of the random sampling, with 5% and 10% of the total points, for different values of d.

the memory limit were recorded.

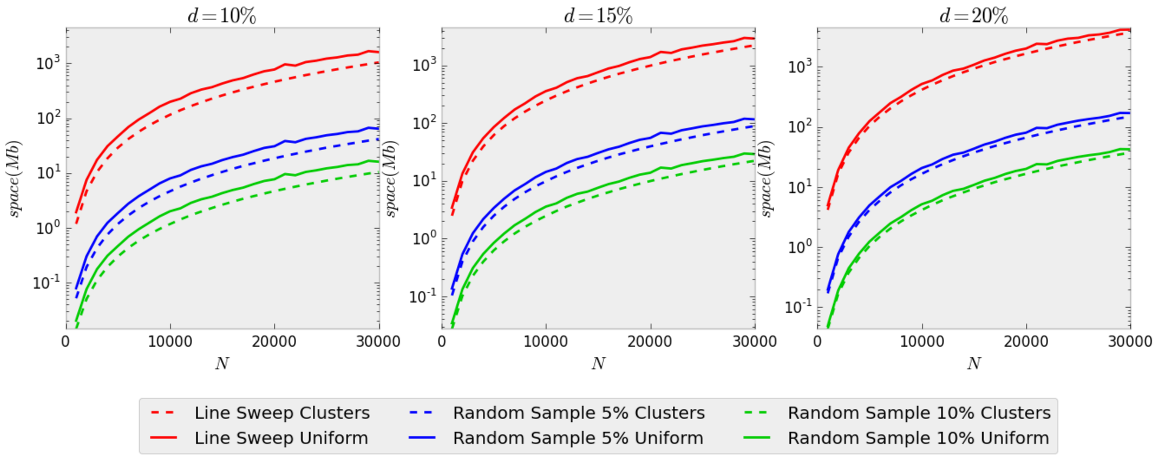


Figure 4.9. Memory used by the line sweep algorithm, and two instances of the random sampling, which sample 5% and 10% of the number points, for different values of d.

Figure 4.9 shows that the random samples use the space expected above. Where the line sweep algorithm uses an average of 4164.93Mb for the clustered inputs, the random samples use 167.83Mb for the 1-in-5 version and 41.44Mb for the 1-in-10 version. The uniform inputs had a similar result, with the line sweep algorithm using an average of 3729.71Mb for the largest inputs, and the random sample algorithms using 149.17Mb for the 1-in-5 version and 37.31Mb, matching our initial prediction of 1/25th and 1/100th the size of the line sweep algorithm to the second decimal case. Upon closer inspection, however, it can be noted that the resulting set is not optimal. Figure 4.10 shows that the number of selected points is inferior.

This can be explained by the fact that the algorithm is not accounting for all points. In fact, whenever the original set has one isolated point, away from any other by more than the coverage

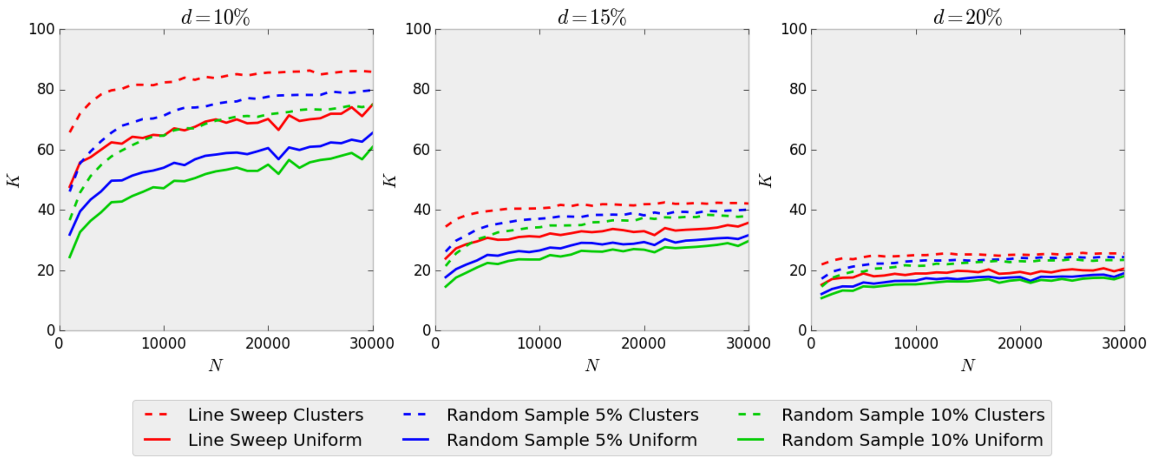


Figure 4.10. Number k of points selected for the line sweep algorithm, and two instances of the random sampling, which sample 5% and 10% of the number points, for different values of d.

distance, it has a 4 out of 5 chance of being discarded and unaccounted for (or 9 out of 10, depending on the version of the algorithm). This means that not all points are being covered, and the number of selected points decreases. Figure 4.11 shows this effect in a real map:

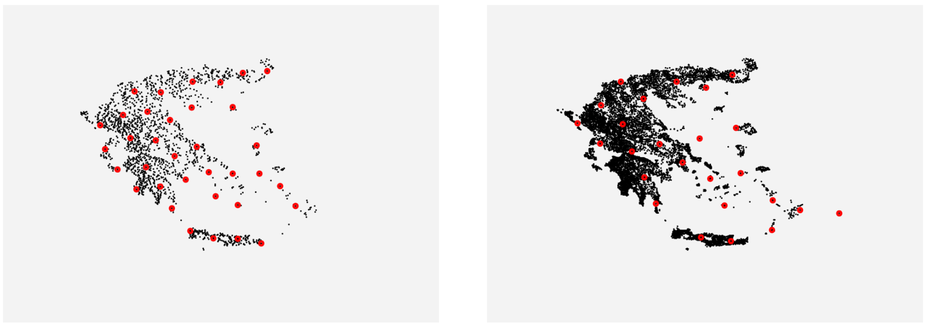


Figure 4.11. Selected subset using the full set (right) and a sampled subset (left). The data was obtained from the University of Waterloo, Ontario, Canada.[1].

As shown, the sampled subset does not cover the most isolated points. In fact, the westernmost point in the map is not present in the sample and, therefore, is not covered. This method is not suitable for the initial requirements, since very isolated points have only a small chance of being represented.

4.2.2 Two-phase filtering

A solution to the random sampling algorithm problems is to perform two passes of the approximation algorithm for the geometric disk cover problem. The first pass over the points is done with a very small radius, since using a small distance speeds the algorithm up, as seen in Figure 4.8. This can be done very quickly, since the range search only has to look in a very small area. This means that many of the points that are close to each other are discarded, but a representative neighbour is always left in its place. Isolated points are then never unaccounted for. The resulting set is much smaller than the original, whilst still keeping a representativeness degree. The second pass, now with the full coverage distance takes advantage of the much smaller set of points for the speed-up. The *CPU* times for three instances of the algorithm can be seen in Figure 4.12. The values of the first pass distance d' are given as a percentage of the final pass distance d. One of the instances uses $d' = 0.05d$, another $d' = 0.1d$ and the other uses $d' = 0.2d$.

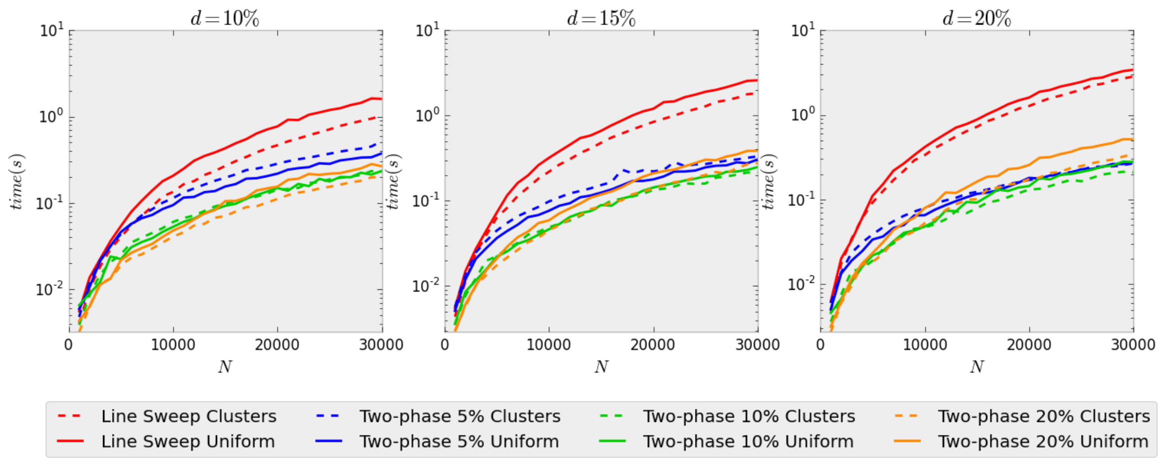


Figure 4.12. CPU-time for the line sweep algorithm, and three instances of the Two-phase filter, with $d' = 5\%$, $d' = 10\%$ and $d' = 20\%$ in relation to d.

As it can be seen, the two-phase filtering is more efficient than the simple line sweep algorithm and does not seem to vary a lot with the total number of points. The first pass eliminates a very large number of redundant points, leaving a similar resulting set for different values of N. The intermediate subset is still representative of the original. The smaller distance requires less edges to be created, and the proximity graph for the first phase is smaller than a one-phase algorithm would have. The proximity graph for the second phase benefits from having less points, which require less edges. Since this algorithm calculates two graphs, both of which sparser than the complete on in the Line Sweep and k-d tree algorithms, the memory limit was not an issue for the inputs tested. Figure 4.13 shows the memory footprint of the three versions of the algorithm.

These algorithms are less predictable concerning the size of the graphs, since they depend a lot from local density of points for the first phase. While they do not have as small a footprint as the random sampling, they still use considerably less than the line sweep and k-d tree range search. They also do not share the issue of leaving isolated points completely uncovered like the random sampling algorithms do, returning better representative subsets. Figure 4.14 shows the variation in size of the final subset chosen by the two-phase filtering algorithms.

The graph shows that despite the two-phase algorithms returning lower numbers of K, they are not as low as the random sampling. Despite accounting for isolated points, the resulting set does not necessarily cover all the points with disks of radii d. Because the first pass transforms the set into disks of radii d', and the second pass only considers their centre point as the measure of cover, then there can be points that are further away from the centroids than d, the maximum

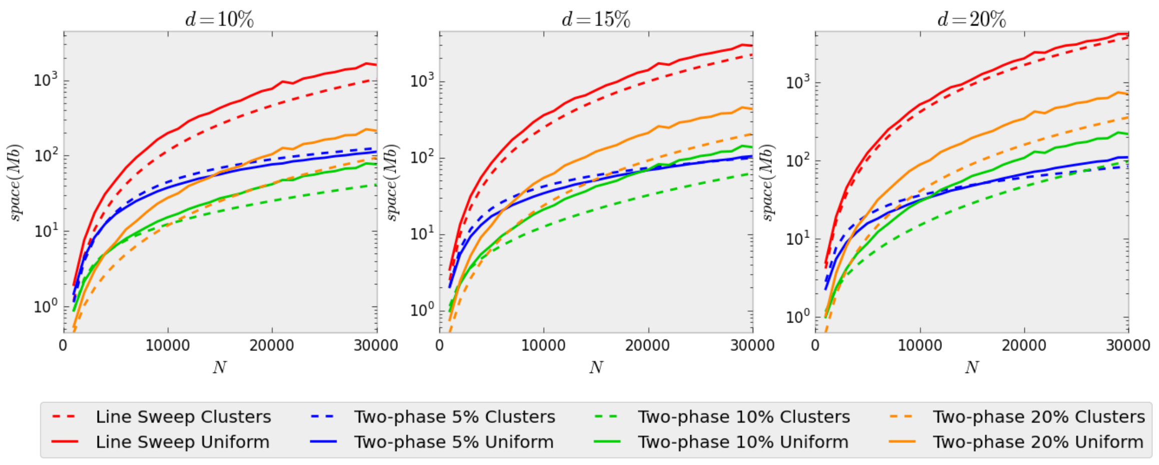


Figure 4.13. Memory used by the line sweep algorithm, and three instances of the Two-phase filter, with $d' = 5\%$, $d' = 10\%$ and $d' = 20\%$ in relation to d. Since two graphs are built, only the larger one is considered for the purposes of this graph, as they are never simultaneously stored in memory.

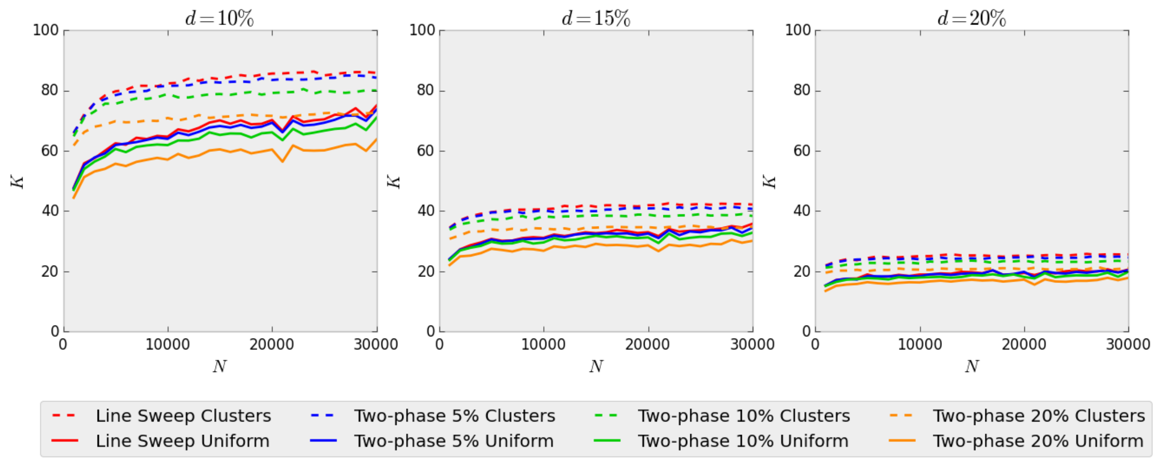


Figure 4.14. Number k of points selected for the line sweep algorithm, and three instances of the Two-phase filter, with $d' = 0.05d$, $d' = 0.10d$ and $d' = 0.2d$ to various valus of d.

distance being $d + d'$, as show in Figure 4.15.

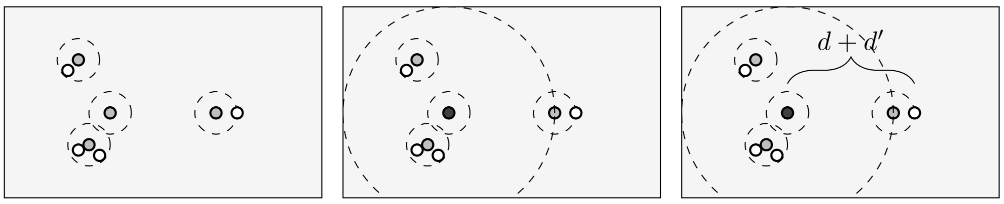


Figure 4.15. Illustration of the worst case scenario for the error of the two-phase algorithm.

This result has some effect on the quality, but it is not nearly as noticeable as the results from the random sampling. Figure 4.16 shows the compared output between the original line sweep algorithm, and the different versions of the two-phase algorithm, as well as their intermediate phases.

This final result shows that the two-phase filter, especially with $d' = 0.1d$, can yield very approximate results at a fraction of the time and space, thus making a good candidate to use with inputs larger than the regular Line Sweep method can handle.

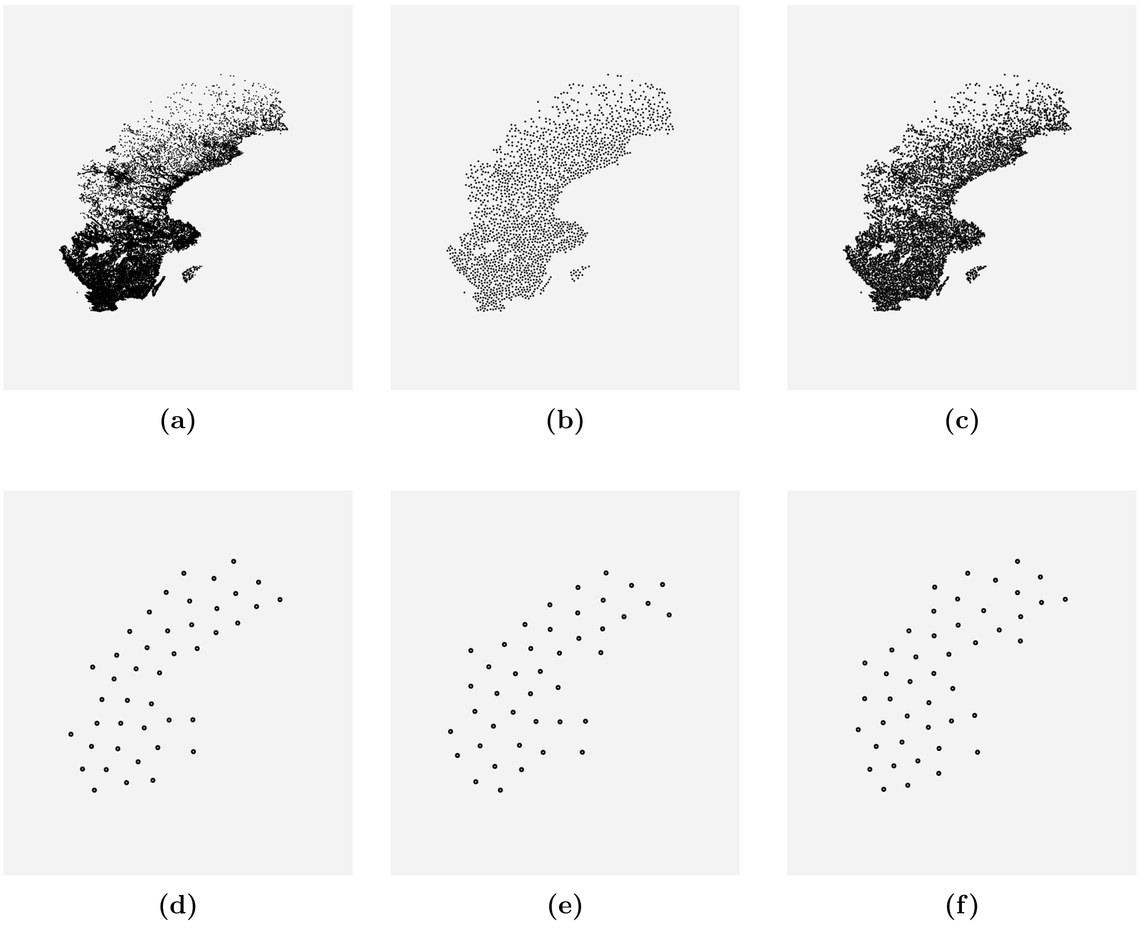


Figure 4.16. Two-phase filter result comparison. (A) Original set; (B) Intermediate set for two-phase filter with $d' = 0.1d$; (C) Intermediate set for Two-phase filter with $d' = 0.05d$; (D) Final set for one pass; (E) Final set for two-phase filter with $d' = 0.1d$; (F) Final set for two-phase filter with $d' = 0.05d$

4.3 Region Panning and Zooming

One of the requirements established for the final algorithm it that it should be able to handle a translation motion in the region and starting point set. This means that in case the new region intersects the previous one, it is important for the user that the points already in display are kept selected, so that each move does not cause the user to lose track of the movement. As such, any selected centroids inside the intersection of the two regions, and all the points covered by them, should be kept unchanged, even if it means losing the guaranteed approximation to the optimal value. Each of the approaches described earlier can be modified to ensure that this property is met.

To handle the panning case, each instance of the program receives the centroids chosen in the previous region. To ensure that the points are still selected is a matter of artificially increasing their number of neighbours by N, then compute the problem regularly. Since the number of points from the old region is bound from above by at most 128, the new number of points is relatively insignificant for instances of 10000 points and more, and the algorithms show no significant difference in performance. Since the covering distance does not change, the old centroids are not close enough together to cover each other, otherwise they would have done so in the old region. This solves the panning issue without needing to add much complexity to the algorithms. In the case of multiple phase filters, the weight of the selected points must be artificially increased in all the phases, to ensure they are kept selected.

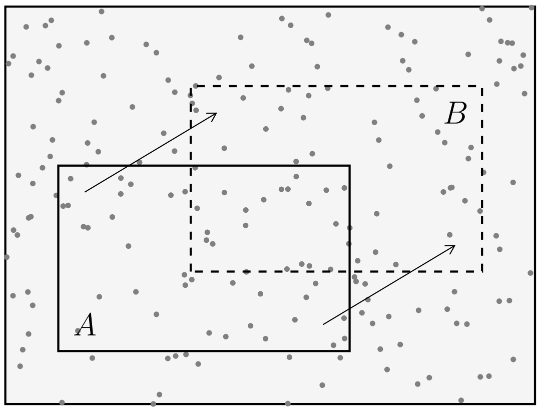


Figure 4.17. Example of a panning motion from region A to region B. Centroids contained in $A \cap B$ calculated in region A must be kept when calculating region B.

In the case of zooming, the scale changes, and so does the minimum distance. The points selected previously are relatively more close together in the new window, and they shift towards the centre of the region. One of the solutions is to simply discard the previous selected points, since they do not have the same representation properties they did in the previous region. Another solution is to do the same process as for the panning. By artificially inflating the weight of the established points, they are given priority over the new window's points. Since the points are now closer together in the window, they may cover each other, resulting in some prioritised points not being selected.

4.4 Discussion

The approximation algorithms in this chapter meet the efficiency requirements established for the web application, and are also able to handle panning and zooming motions without any complexity. In the case the test inputs prove to still be too large to be handled by the approximation

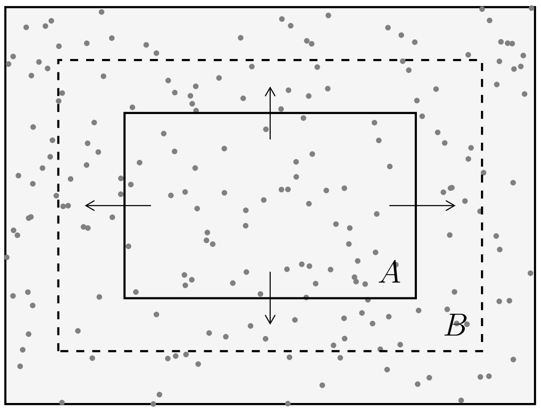


Figure 4.18. Example of a zooming motion from region A to region B. Centroids contained in $A \cap B$ calculated in region A should have selection priority when calculating region B.

algorithms, a good alternative is found in the two-phase filtering approach.

A solution for even larger set of points may require more filter phases, but for the inputs given, no such solution was needed. Furthermore, since the two-phase filtering approaches show similar CPU-times to the random sampling approaches, while showing fewer critical shortcomings, there should be no advantage in using random sampling.

Chapter 5

Conclusion

In this thesis, we designed algorithms to select representative subsets from large quantities of geographic points. The algorithm must be able to handle panning and zooming motions along the geographic region displayed, as well as be efficient enough to be integrated in the back-end of a real-time Web Application.

We started by defining representation as finding the optimal solution to the *k-centre* problem. To solve this problem, two different incremental branch-and-bound approaches were implemented. We then compared the performance of these approaches and one formulation of the problem in integer linear programming to ascertain if the problem could be solved in real-time. The results showed not only that optimal approaches are not efficient to meet the time requisite of the application, but also that the problem required a priori information about the number of point clusters in the region.

In a second phase, the definition of representation was changed to find the solution to the *geometric disk cover* problem instead. This new approach manages to compute the cardinality of the final subset with no information about the region. To solve this problem, we implemented two versions of an approximation algorithm, both of which calculated subsets very efficiently, with a guarantee of approximation to the optimal value. We then performed some tests to determine the most efficient solution. Additionally, we used heuristic approaches to further speed up the algorithm and reduce its memory footprint, at the cost of losing the guarantee of approximation.

Throughout this thesis, we found that interpreting the representation problem as *geometric disk cover* problem, and solve it with an approximation algorithm is an efficient way to be applied in the context of a real-time web application. We also found viable solutions for solving larger instances of the problem.

5.1 Future Work

The next step for the web application developer is to integrate the algorithms described and implemented during this thesis. Since the algorithms are already implemented taking into account their place in the architecture of the final product, the inputs and outputs are already conformed to the specifications given. Full integration requires that the communication layer, as well as the proper protocol request and response parsers are implemented.

Although the results were satisfactory, it may still be possible to develop more efficient algorithms. The bottleneck for our final approaches lies in the proximity graph building stage, particularly performing the various range search queries to establish neighbouring points. This means that implementing more efficient range search structures is a good strategy to find better solutions. Furthermore, since the evaluation of the results is entirely subjective, new solutions might require different interpretations on the concept of representation, such as the notion of uniformity or other similar metrics.

Appendix A

Theorems and Proofs

A.1 Median of Medians

Efficiently constructing a balance k-d tree depends on an efficient method to pick the point that divides the hyper-rectangle in two. One way to reasonably quickly find a value close to the median is to find the median of a sample. To ensure the quality of this sample, is to gather the medians of smaller subsets which can be quickly calculated. This algorithm is an example of a *selection algorithm* and is known as the *median of medians* algorithm [7].

The median of medians algorithms works as follows. Any starting array S consisting of n arbitrary values is split into $n/5$ sub-arrays, each containing at most 5 elements (the last array might have less, depending on whether n is divisible by 5 or not). For each of the sub-arrays, the median can be calculated in constant time, since for 5 values it can be done in at most 6 comparisons, which for the whole array S takes $6n/5$ comparisons. After finding all the sub-arrays' medians and gathering them in a new array F, the algorithm then is called recursively for F until only one value M remains. M is then used to partition the input into two sub-groups: elements smaller than M and elements larger than M. The two subgroups are then concatenated in increasing order and with M in between them, and the algorithm is recursively called again for the group that contains the $n/2$th point of the newly concatenated list. Whenever the list has less than a given number of elements, the median is calculated via brute-force, to avoid infinite recursion. This value is returned by the initial recursive call of the function.

As stated above, this algorithm only returns a value close to the real median. Despite this, it can proven that for any array S, the value M is always between the 30th and the 70th percentiles. At each recursive stage, the values in F larger than M are discarded. This means that out of the $n/5$ values for any given vector, $n/10$ is larger by definition, since M is picked as the median. For each value in F larger than M, there are also two other values that are larger than M, since each value in F was chosen as a median out of 5 different values. This means that the number of values greater than M is at most $3n/10$. Similarly, by a symmetric proof, there are also $3n/10$ values in S smaller than M. This also means that the second recursive call can at worst have $7n/10$ elements, which is a constant fraction of the input. This property is essential in proving the linear complexity of the algorithm.

Analysing the time complexity $T()$ of this algorithm requires analysing separately both recursive calls of the algorithm. The first recursive call occurs in a list of size $n/5$, and takes $T(n/5)$ time. The second recursive call occurs in a list with $7n/10$ elements, which takes $T(7/n)$. Finding the median for a group of 5 elements requires a constant number of comparisons. These comparisons can be arranged in such way that only 6 are necessary for a group of 5 elements. This means that the algorithm has a constant factor of 6/5 for calculating a median on its smallest division. $T(n)$ is then given by:

$$T(n) \leq 6n/5 + T(n/5) + T(7n/10) \tag{A.1}$$

If $T(n)$ has, in fact, linear time complexity, then there is a constant c such that:

$$\begin{aligned} T(n) &\leq 6n/5 + cn/5 + 7cn/10 \\ &\leq n(12/5 + 9c/10) \end{aligned} \tag{A.2}$$

If $T(n)$ is to be at most cn, so that the induction proof is valid, then is must be true that:

$$\begin{aligned} n(6/5 + 9c/10) &\leq cn \\ 6/5 + 9c/10 &\leq c \\ 6/5 &\leq c/10 \\ 12 &\leq c \end{aligned} \tag{A.3}$$

This proves that $T(n) \leq 12n$, or any larger constants than 12 multiplied by n comparisons.

A.2 Set Cover Approximation Algorithm

Theorem A.1. *The greedy algorithm for the Set Cover can find a collection with at most $m \log_e n$ sets, where m is the optimal number, and n is the number of elements covered by all sets.*

Proof. Let the universe U contain n points, which can be covered by at least m sets. The first set picked by the algorithm has size at least n/m. The number of elements of U left to cover n_1

is

$$n_1 \leq n - n/m = n(1 - 1/m) \tag{A.4}$$

The remaining sets must contain at least $n_1/(m-1)$ elements, otherwise the optimal solution would have to contain more than m sets. By iteratively calling the same process, the number of sets at stage i is given by

$$\begin{aligned} n_{i+1} &\leq n_i(1 - 1/m) \\ n_{i+1} &\leq n(1 - 1/m)^{i+1} \end{aligned} \tag{A.5}$$

If it takes k stages for the greedy algorithm to cover U, then $n_k \leq n(1-1/m)^1$ needs to be less than 1.

$$\begin{aligned} n(1 - 1/m)^k &< 1 \\ n(1 - 1/m)^{m\frac{k}{m}} &< 1 \\ (1 - 1/m)^{m\frac{k}{m}} &< 1/n \\ e^{-\frac{k}{m}} &< 1/n \ldots (1-x)^{\frac{1}{x}} \approx 1/e \\ k/m &> \log_e n \\ k &< m \log_e n \end{aligned} \tag{A.6}$$

This means that the size of the collection of sets picked by the greedy algorithm is bound above by $m \log_e n$, which gives the greedy algorithm a $\mathcal{O}(\log_e n)$ approximation to the optimal solution.

□

A.3 Circle Packing Upper Bound

The circle packing problem's goal is to find the best disposition of non-intersecting circles inside a square, given the ratio of the circle diameter to the width of the square [16]. In our approach to the geometric disk cover, the radius of the disks is given as a fraction d of the largest dimension of the rectangular region. This means that the case of a rectangular window is bound from above by the solution of its containing square. The circle packing algorithm also expects non-intersecting circles contained by a square. To find the circle packing solution for our problem, we cannot use or values of distance d for our value of width w, since the circles in our problem have different properties. The properties must be accounted for to reach the equivalent values for the radius r' in relation to the width w' for the circle packing problem.

Since each covering disk around a centroid can intersect other disks in our problem, but cannot contain any other centroids. The centres of the disks cannot be closer than the radius d. Therefore, the circle packing has to be calculated with circles with radius r' half the radius r of the geometric disk cover disks, to ensure the centres are at the right distance.

Furthermore, the circles in the circle packing usually have to be fully contained in the square. In our case, however, they can be partially outside, providing the centre is inside. This means that the side of the square to the corresponding circle packing instance w' is equal to the side of the original region w plus two times the circle packing radius r'.

This means that to find the ratio r'/w' in relation to d, where $r' = r/2$ and $w' = w + 2r'$.

$$\frac{r'}{w'} = \frac{r/2}{w+2r'} = \frac{r/2}{w+r} = \frac{r}{2(w+r)} = \frac{dw}{2(w+dw)} = \frac{d}{2+2d} \tag{A.7}$$

given by $r = \frac{d}{2+d}$. This means that the corresponding radius to the values of $d = \{0.1, 0.15, 0.2\}$ is given by $r \approx \{0.4545, 0.6522, 0.0833\}$, for which the best values found as of the writing of this thesis are $k = \{128, 59, 36\}$, respectively, as demonstrated in [3, 22, 25]. Figure A.1 shows the best distributions found so far.

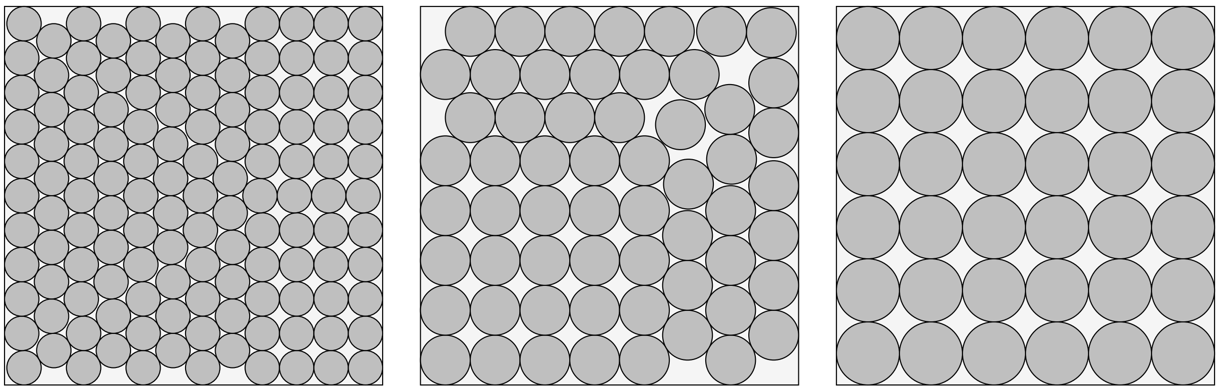


Figure A.1. Most efficient distributions for circle packing for $d = 0.1$, $d = 0.15$,$d = 0.2$.

With the exception of the latter, these values are yet to be proven to be optimal. However, the optimal values are not expected to be very different, as the remaining area is very limited. The

References

[1] Greece TSP dataset, University of Waterloo, Ontario, Canada., . URL `http://www.math.uwaterloo.ca/tsp/world/grpoints.html`.

[2] Sweden TSP dataset, University of Waterloo, Ontario, Canada., . URL `http://www.math.uwaterloo.ca/tsp/world/swpoints.html`.

[3] B. Addis, M. Locatelli, and F. Schoen. Packing circles in a square: New putative optima obtained via global optimization. *Optimization Online*, 154, 2005.

[4] N. Amenta, S. Choi, and G. Rote. Incremental constructions con BRIO. In *Proceedings of the Nineteenth Annual Symposium on Computational Geometry*, pages 211–219. ACM Press, 2003.

[5] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.

[6] M. Blum, R. W. Floyd, V. Pratt, R. L. Rivest, and R. E. Tarjan. Time bounds for selection. *Journal of computer and system sciences*, 7(4):448–461, 1973.

[7] M. Blum, R. W. Floyd, V. Pratt, R. L. Rivest, and R. E. Tarjan. Time bounds for selection. *Journal of computer and system sciences*, 7(4):448–461, 1973.

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

[9] A. Bowyer. Computing dirichlet tessellations. *The Computer Journal*, 24(2):162–166, 1981.

[10] H. Cambazard, D. Mehta, B. O'Sullivan, and L. 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.

[11] A. Caprara, P. Toth, and M. Fischetti. Algorithms for the set covering problem. *Annals of Operations Research*, 98(1-4):353–371, 2000.

[12] G. Cornuéjols, G. L. Nemhauser, and L. A. Wolsey. The uncapacitated facility location problem. Technical report, DTIC Document, 1983.

[13] M. S. Daskin. *Network and discrete location: models, algorithms, and applications.* John Wiley & Sons, 2011.

[14] M. De Berg, M. Van Kreveld, M. Overmars, and O. C. Schwarzkopf. *Computational Geometry.* Springer, 2000.

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

[16] M. Goldberg. The packing of equal circles in a square. *Mathematics Magazine*, pages 24–30, 1970.

[17] L. Hu, W.-S. Ku, S. Bakiras, and C. 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 Press, 2010.

[18] J. W. Jaromczyk and G. T. Toussaint. Relative neighborhood graphs and their relatives. *Proceedings of the IEEE*, 80(9):1502–1517, 1992.

[19] D.-T. Lee and C. Wong. Worst-case analysis for region and partial region searches in multidimensional binary search trees and balanced quad trees. *Acta Informatica*, 9(1): 23–29, 1977.

[20] N. Megiddo and A. Tamir. New results on the complexity of p-centre problems. *SIAM Journal on Computing*, 12(4):751–758, 1983.

[21] A. Okabe, B. Boots, K. Sugihara, and S. N. Chiu. *Spatial Tessellations: Concepts and Applications of Voronoi Diagrams*, volume 501. John Wiley & Sons, 2009.

[22] R. Peikert. Dichteste packungen von gleichen kreisen in einem quadrat. *Elem. Math*, 49(1): 16–26, 1994.

[23] H. Sagan. *Space-Filling Curves*, volume 18. Springer-Verlag New York, 1994.

[24] J. R. Shewchuk. *Lecture Notes on Delaunay Mesh Generation.* Department of Electrical Engineering and Computer Science, University of California at Berkeley, 1999.

[25] P. G. Szabó, M. C. Markót, T. Csendes, E. Specht, L. G. Casado, and I. García. *New approaches to circle packing in a square: with program codes*, volume 6. Springer Science & Business Media, 2007.

[26] J. Valença. Visualisation and Analysis of Geographic Information: Algorithms and Data Structures. *AGILE 2015 - 18th International Conference on Geographic Information Systems* , 2015.

[27] D. Vaz. Subset selection algorithms in multiobjective optimization. MSc in informatics engineering, University of Coimbra, Portugal, 2013.

[28] V. V. Vazirani. *Approximation algorithms.* Springer Science & Business Media, 2013.