Master Thesis

CIPQ

Creating a **C**ommon **I**nterface for **P**rivacy Preserving **Q**ueries in Pervasive Computing

Fall 2007
SW10, project group SW543A
Department of Computer Science
Aalborg University

Faculty of Engineering and Science
Aalborg University

Department of Computer Science

SEMESTER TOPIC:
Database Technologies

TITLE:
CIPQ - Creating a **C**ommon **I**nterface for **P**rivacy Preserving **Q**ueries in Pervasive Computing

PROJECT PERIOD:
SW10,
Fall, 2007

PROJECT DEADLINE:
November 29th, 2007

PROJECT GROUP:
SW543A

GROUP MEMBERS:
Mads Schaarup Andersen,
masa@cs.aau.dk

SUPERVISOR:
Simonas Saltenis
simas@cs.aau.dk

NUMBER OF COPIES: 5

NUMBER OF PAGES: 71

ABSTRACT:

This report documents the development of a common interface for privacy preserving queries in pervasive computing. The interface is based on an analysis of existing solutions, and uses a simple client-server architecture, and is created with flexibility, security, accuracy, and complexity, in mind. With the interface a client has the possibility to specify a minimum cloaked area, and maximum communication allowed on a query to query basis.
Two solutions are developed under the common interface. One method based on a data-independent datastructure, called *the Grid Method*, and one based on a data-dependent datastructure, called *the Quadtree Method*. This is done to explore pros and cons of data-dependency on the server. Furthermore, it is discussed how k-anonymity can be added to the solutions.
To experiment with the two developed solutions, a test is implemented. Tests are carried out on a real world data set, a uniform distribution, and several Zipf distributions. It is found that the *Grid Method* performs best under most circumstances, but in the end it is indicated that this might be due to a simplifying design decision.

Preface

This report is written by Mads Schaarup Andersen (project group SW543A) and documents the efforts done on the SW10 semester fall 2007 at the Department of Computer Science, Aalborg University. It also constitutes as the Master Thesis of the Master in Software Engineering. The project was carried out in the *Database Technologies* research group at the university.

This report documents the development of a common interface for preserving privacy in Location Based Services (LBSs). The aim of this is to provide a flexible way of configuring privacy needs on a query to query basis.

Report Structure

The report is structured in the following way. Chapter 1 is an introduction to the topic of the thesis. Chapter 2 is an analysis of the literature on the topic, ending up in a problem statement. In chapter 3 a common interface is defined, and using this two methods are designed. In chapter 4 tests are carried out and reflected upon, and chapter 5 is the conclusion.

Mads Schaarup Andersen

Contents

1 Introduction **1**

2 LBS Privacy Approaches **3**
2.1 Terminology . . . 3
2.2 Quality Attributes . . . 4
2.3 Architecture . . . 5
2.4 Hiding the Client Position . . . 7
2.4.1 Perturbation . . . 7
2.4.2 Cloaked Area . . . 8
2.5 Datastructures . . . 9
2.6 Main Inspirations . . . 10
2.6.1 SMC Protocol Based Approach . . . 10
2.6.2 The Casper Framework . . . 13
2.7 Problem Statement . . . 17

3 Solution **19**
3.1 Common Interface . . . 19
3.1.1 Simplification . . . 20
3.1.2 Foundation of Methods Developed . . . 21
3.2 Grid Method . . . 21
3.3 Quadtree Method . . . 25
3.4 K-Anonymity . . . 32
3.4.1 Levels of K-Anonymity . . . 33
3.4.2 Common Interface and K-Anonymity . . . 34
3.4.3 Modifications to Grid Method . . . 37
3.4.4 Modifications to Quadtree Method . . . 39

4 Test **41**
4.1 Implementation Details . . . 41
4.1.1 Environment . . . 41

4.1.2 Changes from Design 42
4.1.3 Limitations 45
4.2 Test Configuration 46
4.2.1 Data Sets 46
4.2.2 Client Position 47
4.2.3 Test Configurations 47
4.3 Test Expectations 51
4.4 Test Results 51
4.4.1 Test 1 51
4.4.2 Test 2 55
4.4.3 Test 3 58
4.4.4 Reflection on Results 59

5 Conclusion **65**

Bibliography **68**

Acronyms **69**

A Test Distributions **71**

CHAPTER 1

Introduction

In the recent years, the market for mobile software has exploded. The is mainly due to the fact that a lot of people today own advanced mobile phones and Personal Digital Assistantss (PDAs) which serves as a platform for the software. This development has entailed a whole new marked segment in software.

One of the major advantages in mobile software is the ability to use the user's position to provide a service specific to the area. This type of software is called a LBS. An example is the application of Global Positioning System (GPS) in road network navigation. Here, the user has a device which contains software which uses the position of the GPS satellites to calculate the distance to a desired target.

GPS for road networks is however a rather simple form of LBS as all data resides on the client. More interesting is the aspect of having a simple client – also called a *thin* client – interacting with a server for the same information. This is interesting as it will allow for more dynamic services, as the client does not have to be updated whenever there is a change in the environment. This does however yield the requirement that the client has to inform the server of its location. This approach does however have a downside. If the server is continuously aware of the client's position, a malicious server would be able to extract this information and e.g. use it for extortion. This is from now on referred to as *The Privacy Problem*.

To avoid this *Big Brother* scenario a lot of work has been done in this area. This report will try to examine the known solutions today, and be doing this identifying the advantages and flaws of the different solutions to make up a new solution which will then be tested.

If one imagines a system similar to GPS for road networks, but developed with a thin client, all the information resides on the server. Information about the location

of a city is public domain, and if the client does nothing to hide his location, the client's queries are to be considered public as well [12]. If the client wants to hide his position his query suddenly becomes private. One can also imagine a situation where the data is other clients with hidden positions. Then the data also becomes private. In this project the focus will be on private queries over public data, and to simplify this, static data will also be assumed.

The focus of the solution of this report is to develop a solution for finding the nearest neighbor among a list of points of interest.

CHAPTER 2

LBS Privacy Approaches

The purpose of this chapter is to examine the different solutions to the privacy problem. This is done by explaining the terminology, identifying important software quality attributes, and explaining key aspects to solving the problem. This includes a discussion of architecture, how the client position can be hidden, important datastructures, and main inspirations. This will all serve as a foundation for the *Problem Statement*.

2.1 Terminology

In the previous chapter the purpose was defined as to develop a method which finds the Nearest Neighbours (NN). These neighbors are from now on referred to as *sites*, and so the purpose is to find the nearest site. This is done by a *client* querying a *server*. For this a protocol might need to be utilized. There is furthermore a distinction between *data-independent* and *data-dependent* methods. This refers to whether or not the distribution of sites in the data sets is taken into account. A distinction is also made between the *online* and *offline* phase. The offline phase takes place before the actual querying, and can hence be carried out before interaction between client and server. Online refers to the phase where the two parties interact.

It is not possible for a client not to give away *some* information on his whereabouts. The term *cloaked area* covers all the possible locations of the client. When the server returns a list of possible NNs, this list is called the *candidate list*.

We also define the concept of *spatial privacy*. The meaning of this is that whenever we have a cloaked area in two dimensions, the shape of this region might

reveal a lot about the position of the client in regard to one of the dimensions. The less we reveal about a position in one dimension compared to the other, the better the spatial privacy. Figure 2.1 illustrates shapes with a high degree of spatial privacy and Figure 2.2 illustrates shapes with a low degree of spatial privacy.

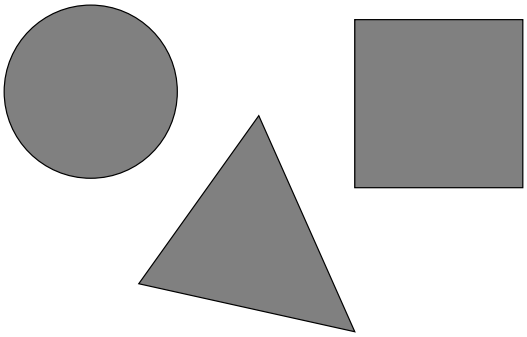


Figure 2.1: Cloaked areas with a high degree of spatial privacy.

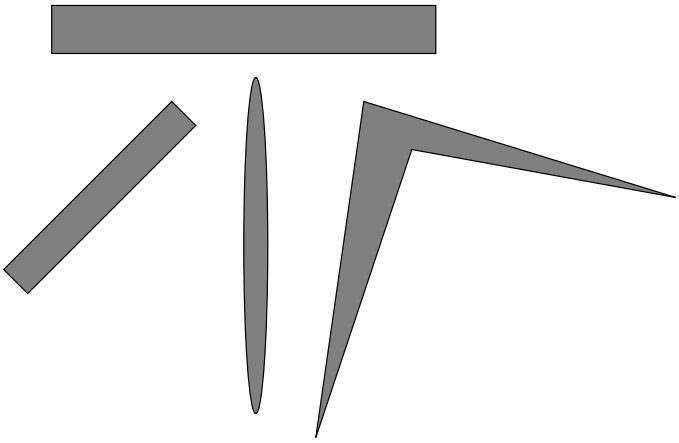


Figure 2.2: Cloaked areas with a low degree of spatial privacy.

2.2 Quality Attributes

To help pinpoint the overall purpose of the solution developed in this project, we present a list and explanation of the software quality attributes which will form the basis of the discussion of the different aspects of the privacy problem. The following quality attributes should have high priority:

Flexibility How well the method is able to change the privacy settings of the client on a query to query basis. The flexibility should be high.

Accuracy Whether the query answer yields the exact answer. The accuracy should be high.

Complexity How much computation is left to the client and how much communication is needed between client and server. Complexity should be low.

Security The privacy of the client. How much information does the client need to give away about his position. Security should be high.

The rest of the chapter will discuss different aspects of privacy in LBSs. This will be in terms of architecture, how to hide the client position, datastructures, and the two main sources of inspiration.

2.3 Architecture

The simplest architecture utilized in LBSs is the standard *client-server* architecture. This consists of a client and a server. In this context the architecture is a pull based approach where the client issues a request to the server, which then issues a reply. This architecture can be seen in Figure 2.3. Formally this architecture has the following steps:

1. Client does preprocessing, and cloaks it's position.
2. Client issues a request (in this context also called a query) to the server.
3. Server processes the query and produces a result in form of a candidate list.
4. Server sends the result to the client.
5. Client processes the answer by doing a NN search.

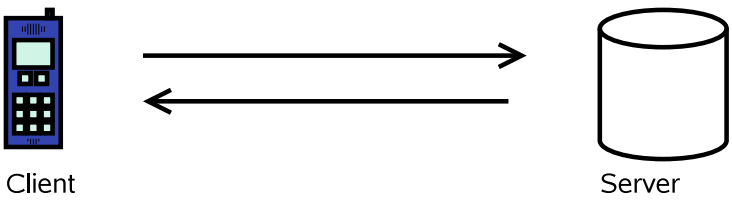


Figure 2.3: Simple client server architecture.

An issue with this architecture is that step 1, in the above steps, requires quite a lot from the client, as the client does not want to send the exact position directly to the server as this would entail no privacy at all.

To cope with this issue a different architecture is proposed. This architecture introduces a third party between client and server. This third party is called an *anonymizer*, and is a trusted party, i.e., a client trusts that the anonymizer will not use the information of the client position for any malicious activity. This modified architecture is depicted in Figure 2.4 and is used in [12, 8]. The steps of a query in this architecture are as follows:

1. Client sends position to trusted anonymizer.
2. Anonymizer does preprocessing to cloak the client position.
3. Anonymizer queries the server.
4. Server processes the query and produces a result in form of a candidate list.
5. Server sends result to anonymizer.
6. Anonymizer does a NN search on result.
7. Anonymizer sends NN to the client.

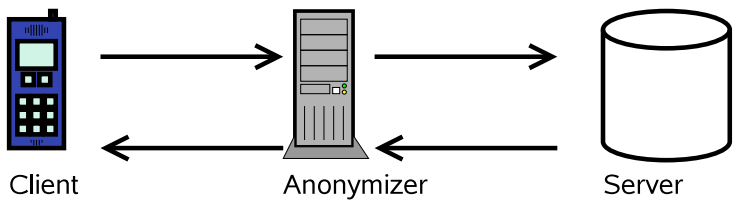


Figure 2.4: The anonymizer architecture.

The advantages of this approach, compared to the simple client-server, is that the communication which is considered costly (the communication the client is involved in) is reduced, as only single sites are transmitted to the client in the anonymizer architecture. Furthermore, most calculations are moved from the client to the anonymizer. The disadvantage is however that a third party is introduced. Even though this is said to be a trusted party, the client position is now made available to a party besides the client itself. This effects the quality attributes of security.

Besides the choice of having an anonymizer or not, the known solutions are based on either having the client send a cloaked area to the server, or having the server make requests on the clients whereabouts. We call this *client based* and *server based* behavior. Most solutions are of the client based kind. Only one known solution is server based, and that is [6]. This will be further explained in Section 2.6.1 on page 10.

2.4 Hiding the Client Position

As mentioned earlier, the client needs to cloak it's position. This can be using one of the following three methods: *perturbation of the position*, *using a cloaked area*, and *using k-anonymity*.

2.4.1 Perturbation

A rather simple solution is the perturbation method. A very simple variant of this method is described in [6]. As the name suggests, this is based on supplying the server with a perturbed, and thereby fake, position defined by a perturbation vector. This gives the advantage that the client can decide how much privacy is needed from request to request. The problem with this solution is that it does not necessarily give the exact answer. This is because the server treats the given position as the exact location of the client, and thereby returns the site closest to the perturbed position. This effects the quality attributes of accuracy. Furthermore, the approach is data-independent, so the client has no way of figuring out what degree of privacy is suitable for a certain position. The inaccuracy problem is illustrated in Figure 2.5 on the next page.

However, more advanced versions of this apparently inaccurate method exists. One of these is *SpaceTwist*. Without going into too much detail, the idea is to overcome the inaccuracy aspect by doing K-Nearest Neighbours (KNN) rather than NN queries. SpaceTwist is constructed so that k is set so that it guarantees that the actual NN of the client position will be included in the result. For further details of this method, see [16].

Despite of the apparent flaw of yielding an inaccurate answer, there are several positive things about perturbation. Each request to the server is treated as a normal request, and thereby does not entail any extra communication overhead. Further-

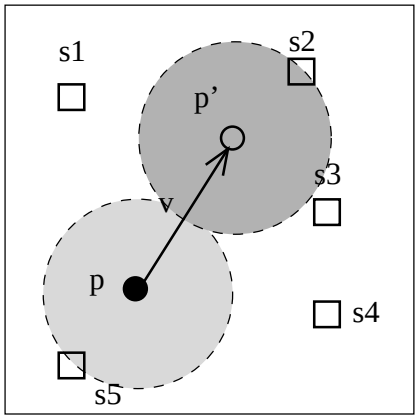


Figure 2.5: An example of the inaccuracy of the perturbation method. The squares s1-s5 represent sites. The filled dot p is the actual client position, p' is the perturbed position, and v is the perturbation vector. The darkest filled circle shows the nearest site to p' and the lighter filled circle reveals the actual nearest site.

more it does not require any form of extra computation on the server. Overall, this solution is good in terms complexity, flexibility, and security, but fails in accuracy. In the more refined methods such as SpaceTwist, the aspect of accuracy is coped with, but this does however also make the solution more complex, and flexibility in SpaceTwist is lower as the attributes cannot be changed from query to query.

2.4.2 Cloaked Area

The final way of cloaking the client position, is to hide the user in a cloaked area. The cloaked area should have the attributes that it should be impossible for an adversary to deduct the actual position from the cloaked area. This means that adding a distance to the point to make out a circle where the actual position is the center of the circle, is not a good choice for the cloaked area.

In the literature different ways of choosing cloaked area, is seen. In [6] the cloaked area is represented as a triangle, in [12] as a rectangle, and in [8] as a polygon.

Outside the scope of this project, in [15] a solution which ensures nothing can be deducted from the cloaked region in a scenario with continuous queries, is presented.

2.5 Datastructures

The last key aspect of the privacy problem, is how the data is structured on the server. The simplest solution would be to have a list of all sites within the space. However, this is not a very efficient solution when the server has to process a large number of queries. Because of this different datastructures are utilized to hold and maintain the data.

As mentioned in the section on terminology (Section 2.1 on page 3), overall there are two different approaches to doing this. A data-dependent and a data-independent, where the first takes site data distribution into account and the second does not.

Data-Independent

The most common data-independent datastructure is a grid. In the simplest form this divides the space into a number of $\lambda \times \lambda$ squares called grid cells. In [6] this simple solution is presented. This solution is based on that the client provides the server with a grid cell rather than an exact position. The server then calculates a result based on the grid cell.

The disadvantage of having a data-independent datastructure is that it requires quite a lot of space, since the same amount of information is stored for empty areas as for areas with high site density.

Data-Dependent

The advantage of the data-dependent datastructures are that they do not store a lot of information about areas with low site density. In common for these are that compared to the simple grid, they do not divide the queryspace into squares of the same size. Instead the space is divided into a number of shapes of different size, the size depending on the site density of the particular area.

In [12] a solution originating in the simple grid is presented. Here a hierarchy of grids are stored in different levels. The solution is based on an anonymizer architecture, and the grid hierarchy is stored at the anonymizer. How the grid hierarchy is structured is explained in detail in Section 2.6.2 on page 13.

Another structure is a Voronoi diagram. This is utilized in [6, 11, 10]. This has the properties that a polygon is created around each site. This polygon has the property that all points inside it are guaranteed to have the site as NN. These polygons are called *Voronoi cells*. This makes for efficient query processing as a query of either a point or an area, will return the sites located in the Voronoi cells which intersect (or contain) the area or point. An example of a Voronoi diagram can be seen in Figure 2.6.

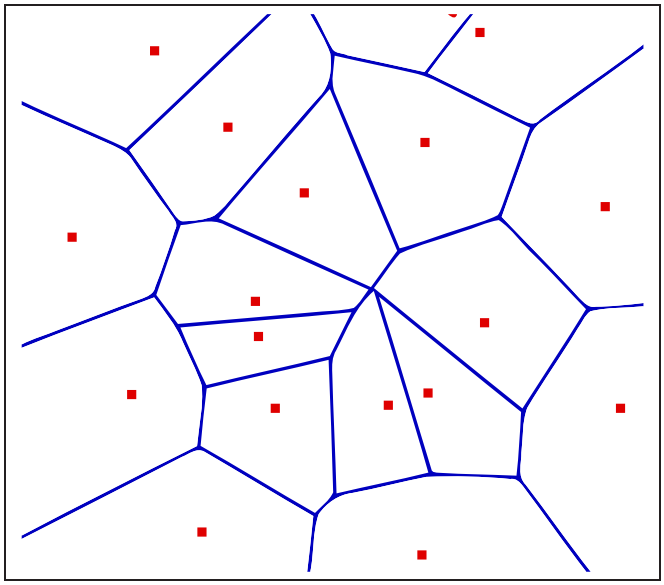


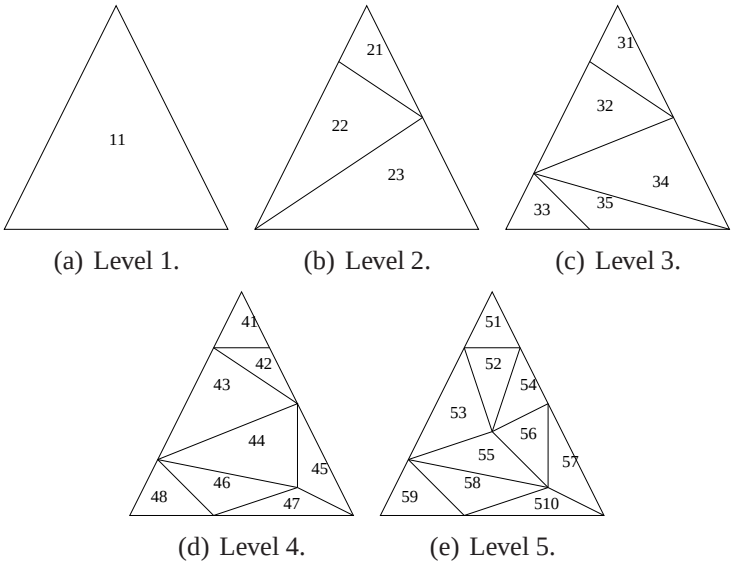
Figure 2.6: Example Voronoi diagram. The small squares are the sites of the queryspace, and the lines make out the polygons surrounding them. The datastructure has the property that all positions inside a polygon have the site in the same polygon as the NN.

2.6 Main Inspirations

In this section the two main sources of inspiration, used for the solution described in the next chapter, will be examined.

2.6.1 SMC Protocol Based Approach

In [6] a solution based on Secure Multi-Party Computational Geometry (SMC) and a Voronoi datastructure is presented. What makes this particular method interesting is the fact that it is a server based solution. The means that the server



(a) Level 1. (b) Level 2. (c) Level 3.

(d) Level 4. (e) Level 5.

Figure 2.7: Different levels in the DAG structure.

asks a series of questions to determine the clients position. As the solution is based on the Voronoi datastructure, this could be done by asking the client if he is in every polygon, one at a time. This would however require the server to ask n questions, where n is the number of sites in the queryspace, questions. To improve this they added a datastructure on top on the Voronoi diagram. This structure is called Directed Acyclic Graph (DAG) and is explained in [9].

To build the DAG on top of the Voronoi diagram, the Voronoi diagram has to be triangulated, and a bounding triangle has to be created. When this is done, the DAG can be built. An example of how a DAG is build, can be seen in Figure 2.7 and 2.8 on the following page. The DAG is constructed bottom up, starting with the finest granularity of triangles (Figure 2.7(e)). Next, vertices with a high number of edges are removed along with the outgoing edges. This is done while making sure that both vertices of an edge are not removed at the same time. Next, the polygons which are not triangles and triangulated again, and then we have a new level (Figure 2.7(d)). This is done until only the bounding triangle remains (Figure 2.7(a)). Then the edges of the DAG are added. This is done by adding an edge from level i to level $i+1$ whenever the triangle of level i intersects a triangle of level $i+1$. The DAG will end up looking like a tree (Figure 2.8 on the following page).

The method of [6] is based on starting from the top level and working the way down the tree. This is done by determining whether the client is in a certain trian-

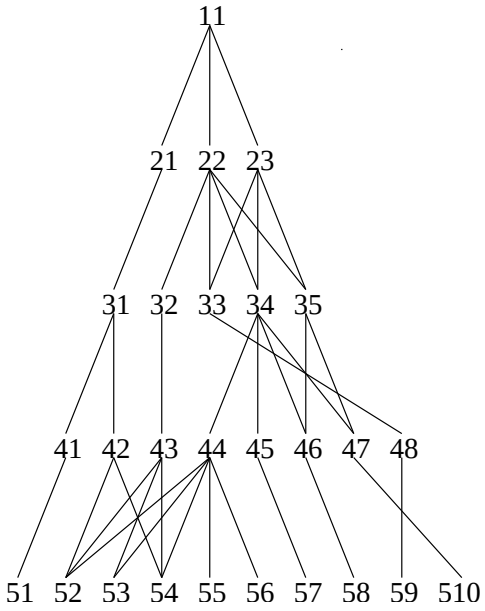


Figure 2.8: The DAG structure.

gle. To answer the question of whether the client is located in a certain triangle, a SMC protocol is utilized. This protocol is described in details in [5]. Doing this has the property that the client will not be able to deduct anything about the triangle, and the server will not be able to deduct anything about the position of the client. This process goes on until a triangle which is totally enclosed in a Voronoi cell is found. The server then returns the site of this Voronoi cell to the client, and the client is then guaranteed that this site is NN.

Discussion

The main problem with the SMC based approach is that there is a lot of communication between client and server. This is due to several reasons. First reason is that the DAG structure has the problem that a triangle in level i can be linked with more than one site on level $i-1$. This means that a client might answer $true$ to being in more than one site on a level in the hierarchy. Furthermore, the SMC protocol needs more than one *request-reply* pair to figure out if the client is in a certain triangle. This is even though there is not really any harm in letting the client know which triangle the server is testing, i.e., this is considered a pointless communication overhead.

In terms of quality attributes this solution provides both high and low security.

High, as provided by the SMC protocol. Low, as the method only comes to a halt when a triangle at the lowest level of the hierarchy is reached. This might be a very small area, hence the degree of spatial privacy can become very low.

As there is no way of providing a minimum area cloaked area for the client, or any other parameters, flexibility is considered low. Complexity of the solution is high with respects to the SMC protocol, but low in terms of the calculation done at the client. The client basically just needs to initiate a query, and answer a series of boolean questions. With regards to the last quality attribute of accuracy, the solution only returns one site, which is guaranteed to be NN.

2.6.2 The Casper Framework

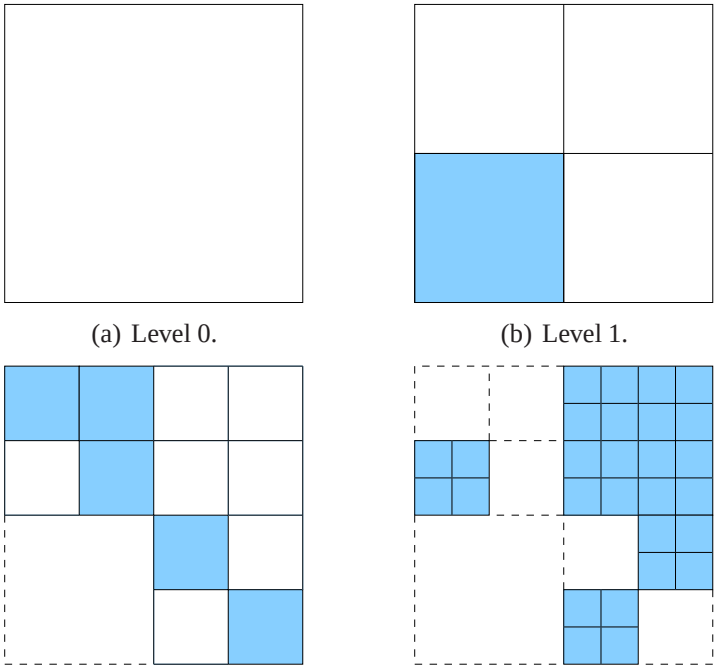
The Casper framework [12] presents a solution to the privacy problem which is based on k-anonymity, a data-dependent datastructure, and the anonymizer architecture. In this framework the anonymizer keeps track of all the users of the system. The datastructure which is utilized is a hierarchy of grids. The top level contains the entire space. The second level is divided into four squares, and on the next level each of these are divided into four squares, and so on. Level i always has four times the amount of square in level $i-1$. Actually squares are only created is it is needed, so if a square only contains one points it is not necessary to divide the square at the next level. This also has the effect that squares can be empty. An example of a grid hierarchy with four level is shown in Figure 2.9 on the next page. The squares of all the levels are indexed in the same hash-table, which is located on the anonymizer.

Another interesting aspect of the framework is the way one can calculate the area surrounding the square in which the client is located, which are candidates to be NNs to all points within the square. The area is called the *extended area*.

Calculating Extended Area

As the solution is based on a grid, a way of finding the smallest area in which NN might be located is presented. This is divided into four steps and these are shown in Figure 2.10 on page 16.

Step 1 The first step is to find the sites nearest to each of the four corners of the client square. These sites are called filter sites, and are named f_1, f_2, f_3,



(a) Level 0. (b) Level 1.

(c) Level 2. (d) Level 3.

Figure 2.9: An example of a four level grid hierarchy in the Casper framework. The grayed out areas contain only one point, and do therefore not need to be further divided in the next level. The dashed out areas, are areas which do not exist for in the following level.

and f_4. These correspond to the corners c_1, c_2, c_3, and c_4. This is shown in Figure 2.10(a) on page 16.

Step 2 In this step we want to find the point m_{ij} on each line e_{ij} (the line between c_i and c_j), which divides the line into two segments $c_i m_{ij}$ and $m_{ij} c_j$. The idea is to do it so that each point in the first segment has f_j as it's nearest filter site, and the second segment has f_j as it's nearest filter site. If the two corners c_i and c_j has the same filter site (i.e., $f_i = f_j$), the point m_{ij} does not exist as all points on line e_{ij} have f_i as the nearest filter site. If the two filter sites are not identical, we find m_{ij} by having a line L_{ij} between the two filter sites f_i and f_j. The line P_{ij} is then plotted. This has the properties that it's intersection divides L_{if} into two equally long line segments, and that it is orthogonal to L_{ij}. P_{ij}'s intersection with e_{ij} is the point m_{ij}. This is shown in Figure 2.10(b) on page 16.

Step 3 In this step we calculate the extended area. We want to find the largest distance of any point on e_{ij} to it's nearest filter site. Only three points need

to be examine. These are c_i, c_j, and m_{ij}. We want to find the largest of these distances, being the length of the segments $c_i t_i$, $c_j f_j$, and $m_{ij} f_i$. In the case where $f_i = f_j$ the length of $m_{ij} f_i$ equals 0. The area is then extended with the maximum of these three distances in direction e_{ij}. This is shown in Figure 2.10(c) on the next page.

Step 4 In this final step, all the sites within the extended area are added to the candidate list, and then these are sent to the client. The client can then perform NN search.

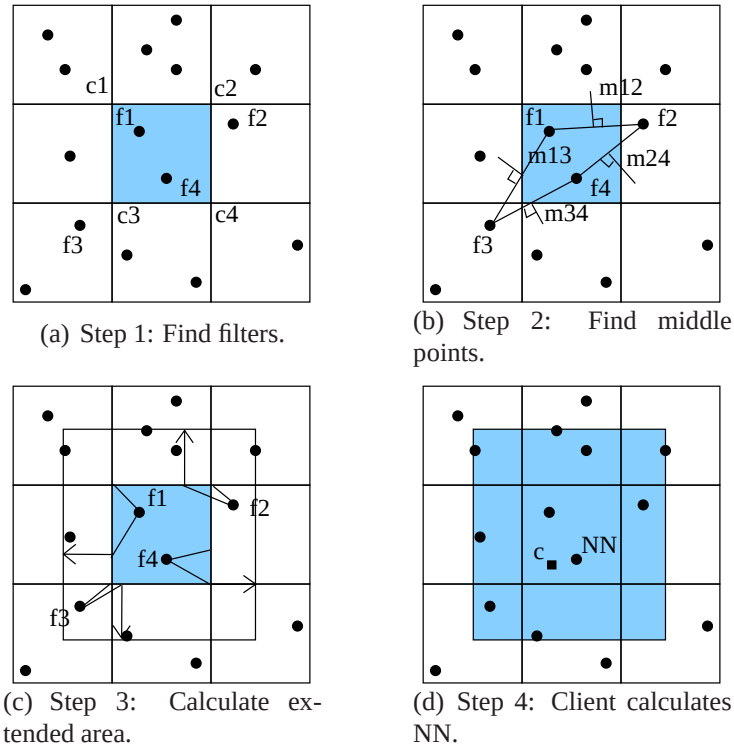
The calculation of the extended area is formalized in Algorithm 2.1.

Algorithm 2.1 $FindExtendedArea(A)$

Require: A : The square in which the client is located.

1: A_{ext} is the extended area and is initially set to A
2: **for all** c_i **in** A **do**
3: $f_i \leftarrow$ the nearest site to c_i
4: **end for**
5: **for all** *edge* $e_{ij} = c_i c_j$ **of** A **do**
6: **if** $f_i == f_j$ **then**
7: $m_{ij} \leftarrow 0$
8: **else**
9: L_{ij} is a line connecting f_i and f_j
10: P_{ij} is a line that splits L_{ij} into two segments of equal length and is orthogonal to L_{ij}
11: m_{ij} is the intersection between P_{ij} and e_{ij}
12: **end if**
13: $d_m \leftarrow Length(m_{ij} f_i) = Length(m_{ij} f_j)$
14: $d_i \leftarrow Length(c_i f_i)$
15: $d_j \leftarrow Length(c_j f_j)$
16: $d_{max} \leftarrow MAX(d_m, d_i, d_j)$
17: Expand A_{ext} by distance d_{max} in $c_i c_j$ direction
18: **end for**
19: **return** A_{ext}

The last interesting thing about the Casper framework, is that they introduce a flexible way in which the client can change the privacy settings.



(a) Step 1: Find filters.

(b) Step 2: Find middle points.

(c) Step 3: Calculate extended area.

(d) Step 4: Client calculates NN.

Figure 2.10: The four steps in calculated the extended area which can contain candidate sites.

Privacy Model

In the framework, the concept of a privacy model is introduced.The settings of this model are stored at the anonymizer. They can however be changed by the client at any time (except in the middle of an ongoing query), and the model therefore enhances the quality attribute of flexibility. In this solution the privacy model lets the client specify a minimum cloaked area. This area is represented as a rectangle. The privacy model is also where k-anonymity is introduced. The user is able to set among how many (k-1) other users he wants to be indistinguishable.

Discussion

The Casper framework has a lot of interesting aspects. In particular in terms of flexibility as the client is able to change the privacy settings at any time. In terms of security the solution is based on an anonymizer, which in the scope of this project is considered an unnecessary risk, thereby yielding lower security.

With the proposed method of calculating the extended area, in which NNs can be located, the accuracy is high. In terms of complexity the solution has a very thin client, but this is mainly due to the fact that the anonymizer architecture is used.

2.7 Problem Statement

In relation to the different aspects of the privacy problem, we will now explain how the four quality attributes should be supported in out solution. They are specified in prioritized order.

Security Privacy of the client. The solution has to provide the server with as little information as possible. Also no parties besides the client and server should be involved, i.e. an anonymizer architecture is out of the question.

Flexibility Defined as being able to adjust privacy settings from query to query. More specific the client needs to be able to control how much communication is allowed between itself and the server. Furthermore the client has to be able to specify a minimum cloaked area.

Complexity The complexity of the solution. Here the client should be kept as thin (i.e. simple) as possible, leaving most of the calculation to the server. The server should furthermore be able to do as much calculation offline as possible.

Accuracy Defined as the accuracy of the query answer. This should be high in the sense that the client is able to obtain the exact answer.

The above can also summed up in the following demands:

- Third parties have to be eliminated.
- The client has to be able to control how much communication is allowed between client and server.
- The client has to be able to specify a minimum cloaked area.
- The client request should give an exact answer.
- The client should always get an answer. This has higher priority than ensuring that communication is kept to a minimum.

- The client should be a thin client, and leave most calculations to the server.
- The server should be able to do as much calculation as possible, offline.
- The solution should explore the differences between data-dependent and data-independent datastructures.

CHAPTER 3

Solution

The following chapter describes the solution developed in this project. The solution is based on the analysis - more specific, the problem statement - of the previous chapter. Two solutions will be developed. One based on a data-independent datastructure and one on a data-dependent datastructure. They both have to adhere to the following common interface.

3.1 Common Interface

As stated in the problem statement in Section 2.7 on page 17, the client has to be able to control the amount of communication units used and the minimum cloaked area. Furthermore, this has to be done *on the fly*, i.e., the client has to have the possibility to change these settings from one query to the next. This is made possible using the following interface of which both methods have to adhere:

```
Input:  m: Maximum communication
        a: Minimum cloaked area
Output: Nearest site

Interface Query(m,a)
```

In this interface *Maximum communication* is defined as how many communication units are allowed to take place. The parameter is an positive integer. In this context communication covers both the size of the data set being transmitted as well as the amount of request-reply pairs. The two are defined as follows:

Request-reply pair Defined as one request from the server or client, plus the resulting reply. Uses 2 communication units.

Candidate list The number of elements being transmitted in the candidate list. Each element uses 1 communication unit.

Minimum cloaked area is defined as the acreage of the cloaked area. The unit of measurement has to be the same as the unit of measurement of the queryspace. This is also a positive integer.

With the two above parameters specified, the overall goal of this interface is to get a candidate list without using more than the specified maximum communication, and without revealing more about the client position than the specified minimum cloaked area. We therefore want a candidate list sent to the client as soon and the above can be satisfied. We call such a state a *satisfactory state*.

The interface itself should be implemented in the client. This is because we do not want the server to know neither the remaining communication units, or the minimum cloaked area. The client therefore has to keep track of the two parameters and never make the server aware of the these.

Sometimes a satisfactory state is never reached. An example is, when the value of maximum communication units is smaller than the smallest candidate list which can possibly be returned. As stated in the problem statement of the previous chapter, the need of an answer is regarded higher than the need to satisfy the maximum communication parameter. We call a state were we are forced to exceed maximum communication a *state of overflow*. Once such a state is reached, the server should immediately return the candidate list.

3.1.1 Simplification

To simplify the project it is chosen to limit the query space to a square, so that width and height are identical. Furthermore it is chosen that the queryspace itself has to be a square.

In addition, the proposed solutions will only be concerned about single queries, and the consequence of continuous queries are out of the scope of this project.

3.1.2 Foundation of Methods Developed

As mentioned, two solutions will be developed. These will be based on a number of request-reply pairs. These will be used to the send the bounds of a square and the number of sites present within the bounds. The clients will then, based on the two parameters of the interface, decide when they are in a satisfied state, and when this state is reached the candidate list will be send to the client.

3.2 Grid Method

The first method developed is based on a data-independent datastructure at the server. Here we choose a very basic grid. In Section 2.5 on page 9 this was explained. We have identified the problem of such a solution as being that instead of giving away a point, one now had to give away a grid cell, which was predefined at the server. We also saw a solution trying to cope with this problem by having different size grids stored at the server. We are however not satisfied with leaving the decision of our minimum cloaked area to be left to the server. Instead we propose a new solution based on the simplest one level grid - being the one with only one size grid cells on the server. This new solution entails some changes both on the server and the client.

As explained, there is not much difference between the server in this solution compared to the one in a basic grid approach. This means that the basic datastructure is not changed at all. On initialization all sites and distributed into the corresponding grid cells based on their position. The size of a grid cell is defined as $\lambda \times \lambda$, meaning that the entire grid will consist of $\frac{queryspace.width}{\lambda} \times \frac{queryspace.width}{\lambda}$ cells. These different concepts are visualized in Figure 3.1. The parameter λ is set on initialization of the server, and is made available to the client. The difference from this method to the basic grid method, lies in the way the client is able to query the server.

Instead of querying a single cell, it should be possible to query a range of cells. When querying a grid cell (or a range of grid cells) it is however not enough to only include the sites contained in the cells themselves. How to extend the area so that all candidate sites to all points inside the gridsquare is explained in detail in Section 2.6.2 on page 13. We here propose a simplification of this. We choose this even though it compromises the quality attributes of accuracy defined in the previous chapter. We do this as the maximum inaccuracy is relatively rare. The

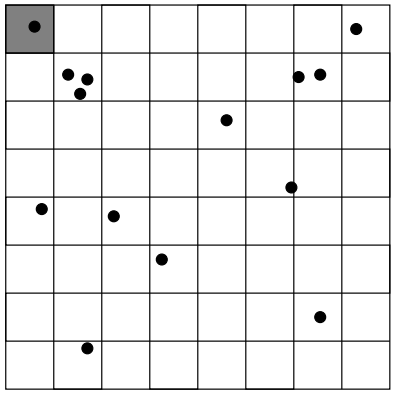
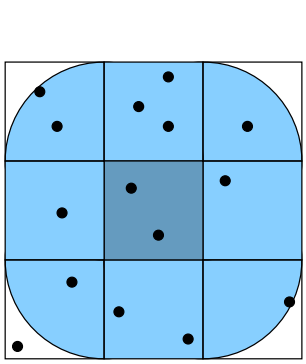
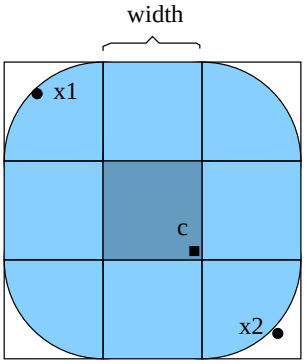


Figure 3.1: A sample grid. The black dots represent sites. The gray square in the top left corner is a grid cell with width λ. The queryspace is the entire grid.



(a) The grayed out area represents the simplified extended area.



(b) Situation showing the worst case scenario of the simplified extended area, in which the highest inaccuracy will occur.

Figure 3.2: This figure illustrates the simplification of the extended area, and the inaccuracy which this can yield.

simplification is shown in Figure 3.2.

The simplification includes all points which are *width* away from the cloaked area. *width* is the width of the square which consitutes the cloaked area. This can be seen in Figure 3.2(a). In Figure 3.2(b) the worst case scenario is illustrated. The client is in the bottom corner of the area provided to the server, and x_1 is just within *width* of the extended area. x_2, which is the actual closest site, is located one unit further away than *width*. This has the result that x_1 is exactly the diagonal of the cloaked area further away than x_2. This entails the an inaccuracy of $\sqrt{width^2 + width^2} = \sqrt{2} \times width$. The cloaked area with width equal to *width*

does not have to be a single grid cell, but can be composed of many grid cells. In fact it does not even have to snap to the grid in the first place. In that case the server will calculate in which squares the bounds are located and do a query on those, and afterward sort out the sites which are inside the bounds.

A major disadvantage of a grid based approach is that the datastructure is data-independent. This means that the amount of information stored does not differ much from an empty area to an area with a high density of sites. In relation to the common interface, this is by intuition not good as client might give away a small cloaked area, even in the cases where a much larger area would yield the same number of candidate sites. This effect can be seen in Figure 3.3. We therefore propose a solution where the client queries areas of decreasing size until a satisfactory state is reached. Essentially, this means that we add an extra layer to the grid method, enabling the queries, rather than the datastructure to be independent.

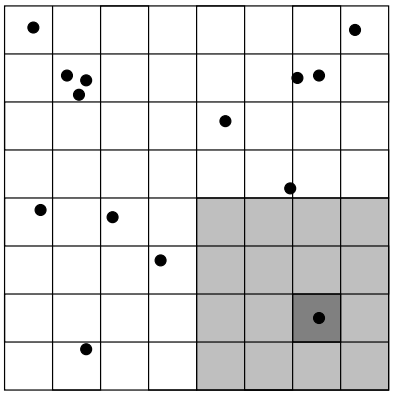


Figure 3.3: Querying the darkest grayed out area will give the same result as querying the brighter grayed out area.

As earlier explained, it is decided that the areas to be queried should all be squares (from now on referred to as querysquares). This is because it yields the highest degree of spatial privacy. The list of querysquares should always contain and start with the entire queryspace. This is the case as the entire queryspace might satisfy the parameters provided to the common interface. It also has to be included because as we require that an answer is always provided, and if we somehow create the squares so that they do not cover all sites of the queryspace, we are not sure to get an answer. Furthermore, we want the smallest, and thereby last, possible querysquare to be determined by the parameter *Minimum cloaked area*. The other querysquares are of size $2^k\lambda \times 2^k\lambda$ where k is a non-negative increasing integer (which satisfies Minimum cloaked area $\leq 2^k\lambda \times 2^k\lambda \leq$ queryspace). The querysquares furthermore has to satisfy the property that querysquare[i] should always totally encapsulate queryspace[i+1]. If this was not the case a malicious

server could obtain more precise information on the location of the client than intended. Formally the server could deduct that the client has to be located in $querysquare[i] \cap querysquare[i-1]$. This effect is shown in Figure 3.4. Further than this restriction, the querysquares should be created randomly. The pseudo code for creating the squares can be found in Algorithm 3.1.

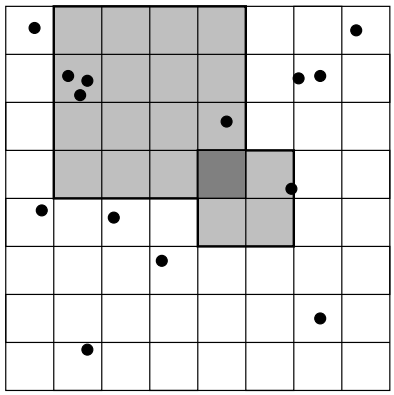


Figure 3.4: The bright grayed out areas represent two different querysquares. The client can then only the located in the intersection between the two represented by the dark gray area.

Algorithm 3.1 CreateSquares()

Require: $Area_{min}$: Minimum cloaked area.

1: $querysquares \leftarrow$ initialize array of squares
2: $cell \leftarrow$ grid cell containing clients position
3: $minimumArea \leftarrow$ create a square of $Area_{min}$ size and enlarge it so that it snaps to the grid. The square should totally encapsulate $cell$
4: $querysquares[0] \leftarrow minimumArea$
5: $j \leftarrow$ set j to the minimum integer which satisfies $2^j\lambda > minimumArea.size$
6: **for** $i = j$ **until** $2^i\lambda \times 2^i\lambda \geq queryspace.size$ **do**
7: $querysquares[i-j+1] \leftarrow$ create $2^i\lambda \times 2^i\lambda$ square encapsulating $querysquares[i-j]randomly$. The square should also snap to the grid
8: **end for**
9: $querysquares[querysquares.size] \leftarrow queryspace.bounds$
10: $querysquares.reverse$
11: **return** $querysquares$

We now have an array of querysquares where querysquare[0] represents the entire queryspace and queryspace[size-1], the minimum cloaked area. These are then sent to the server in increasing order. Each time the server returns the size of the candidate list of that given querysquare. It is then up to the client to test whether or not that size satisfies the parameters *maximum communication* and *minimum*

area. When a satisfactory state is reached, no more querysquares are sent to the server, and instead the client will issue a request for the candidate list of that given querysquare. The satisfactory state is reached whenever the size of the candidate list is less than or equal to what is left of the communication budget - 2. The latter is due to the fact that a request-reply pair is used for the request for the candidate list. The pseudo code for this can be seen in the form of the *GetSites* algorithm in Algorithm 3.2.

Algorithm 3.2 $GetSites(querysquares)$

Require: Com_{max}: maximum amount of communication allowed.
querysquares: array of squares.
$Server.numberOfSites(querysquare)$: a function that makes a call to the Server and returns the number of sites in *querysquare*.
$Server.SitesIn(querysquare)$: a function that makes a call to the Server and return the candidate list of sites in *querysquare*.

```
1: for all querysquare in querysquares do
2:    lastSquare ← querysquare
3:    Com_max ← Com_max − 2
4:    if Server.#sites(querysquare) < Com_max − 2 then
5:       return Server.GetSites(querysquare)
6:    end if
7: end for
8: return Server.GetSites(querysquares.lastElement)
```

In the end, the client does NN search on the candidate list to retrieve the answer to the query. The entire grid method is shown in Algorithm 3.3 and a simplified flow of messages can be seen in the sequence diagram of Figure 3.5.

Algorithm 3.3 Grid method

```
1: querysquares ← Client calls CreateSquares()
2: C_list ← Client calls GetSites(querysquares)
3: Client does nearest neighbor search on C_list
```

3.3 Quadtree Method

Even though we add an extra layer to the grid based approach, making the queries data-dependent, the method is in essence a data-independent solution that relies

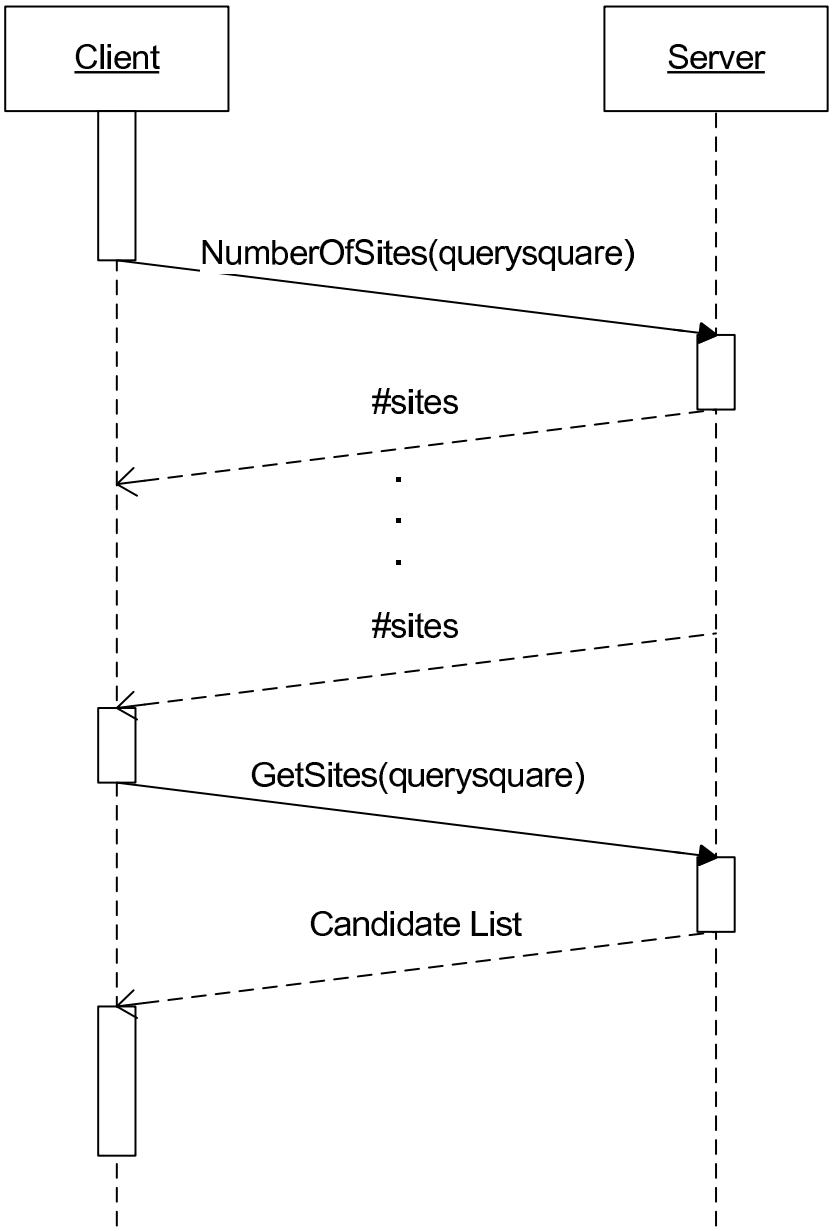


Figure 3.5: Sequence diagram of the grid method.

on a grid datastructure. Because of this we want to develop a method which relies on a server based data-dependent datastructure. In the analysis we found that the datastructure that gives the highest degree of data dependency is a Voronoi based graph structure. However the drawbacks of this datastructure, in terms of extra communication and a low degree of spatial privacy, means that we dismiss this structure. To obtain a high spatial privacy we want a structure which lets us provide the cloaked area as a square as we do in the grid based approach. Furthermore we want a datastructure which has a number of levels, and is indexed in a tree providing the possibility to determine the cloaked area in which the client is located by a higher and higher accuracy by traversing the tree.

What we want is therefore a structure which divides the queryspace into squares, but does so in relation to the density of the sites in different areas. In Section 2.6.1 on page 10 we saw such a structure. This was however not arranged in a tree structure. One datastructure which has the same attributes and is arranged in a tree is the quadtree, which has a lot of similarities with the structure used in [12]. For a detailed look into quadtrees see [7]. To give an idea of how a quadtree works see Figure 3.6 with a grid view of a quadtree and Figure 3.7 on the next page which shows the tree representation of a quarter of the grid view. More specifically the bottom-right quarter. A quadtree is a recursive datastructure where a quadtree can have none or exactly four subtrees (some of which may be empty). The chosen version of the quadtree, a quadtree either has no children and contain maximum one site, or a quadtree has four children and no sites.

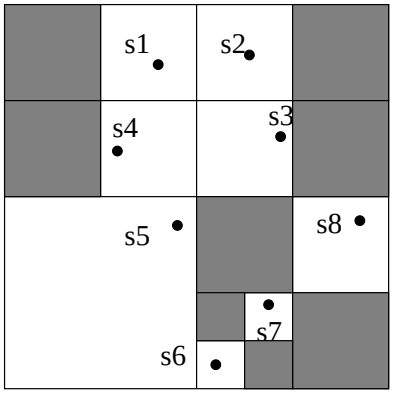


Figure 3.6: An example quadtree of eight sites number from s1 to s8. Shaded areas do not contain any sites, and therefore no information need to be stored about the subtree.

A quadtree has some similarities with a grid, as it divides the queryspace into a number of squares. The difference lies in the fact that the squares are only as small as it is required in the specific region. The quadtree is used to store the sites at the server, and as long as the set of sites remains the same this only has to be

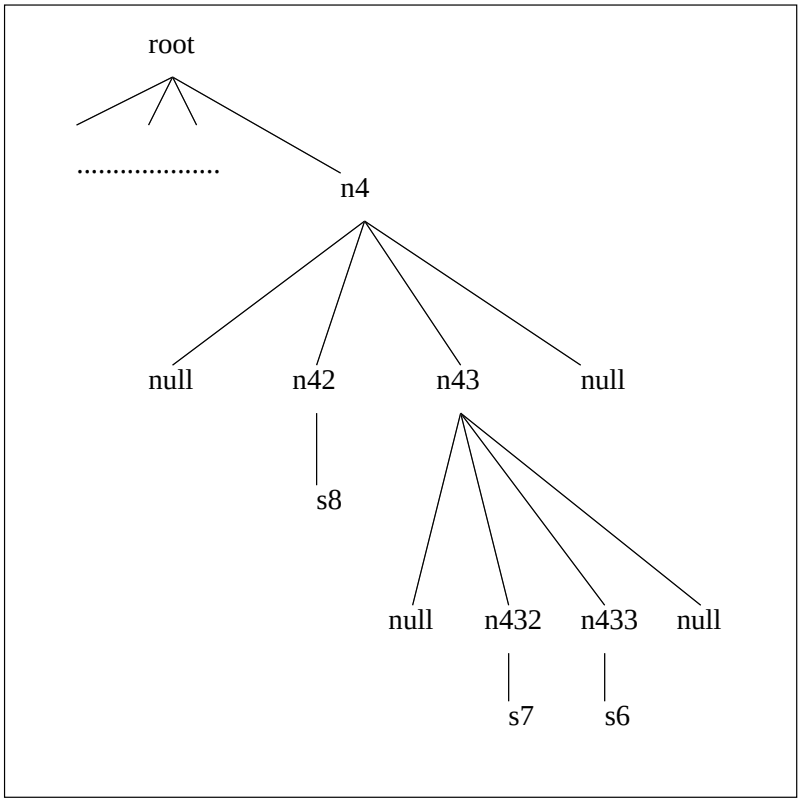


Figure 3.7: Subtree of the forth node of the root of the quadtree in Figure 3.6 on the preceding page.

calculated once. Efficient methods of adding and removing sites do exist [7], but for this project we assume static site data. Searching a quadtree is done by issuing a *range query*. Pseudo code for a range query can be seen in Algorithm 3.4 on the next page.

Even though we dismissed the idea of using the Voronoi based graph structure, and dismissed using a SMC protocol the idea of this method is still based on the overall idea of [6]. This is in the sense that the server sends a series of boolean requests to the client. This is done by starting at the top level of the datastructure and traversing the tree until a satisfactory state is reached.

The server side of this method starts by creating a quadtree of the site data. How this is done can be seen in Algorithm 3.5 on the facing page. As in the grid method, it is however not enough to be able to return the sites of the square in which the client is located. We also have to include surrounding areas which can contain NNs for the bounds of the square. We do this is the same way as in the

Algorithm 3.4 RangeQuery(range)

Require: *quadtree.bounds.intersects*(*range*): a function that takes a range as input and outputs a boolean representing whether of not the bounds of the quadtree intersects with range.
quadtree.numberOfSites: number of sites in the quadtree.
quadtree.subtrees: subtrees of the quadtree.

```
1:  result ← initialize result set of points in the range
2:  if quadtree.bounds.intersects(range) then
3:      if quadtree.numberOfSites == 1 then
4:          result.add(quadtree.site)
5:      else if quadtree.numberOfSites > 1 then
6:          for all subtree in quadtree.substrees do
7:              result.add(subtree.rangeQuery(range))
8:          end for
9:      end if
10: end if
11: return result
```

grid method (for details see Section 3.2 on page 21).

Algorithm 3.5 *BuildQuadtree*($Points_{list}, bounds, parent$)

```
1:  subtrees ← null
2:  this.Points_list ← Points_list
3:  this.bounds ← bounds
4:  this.parent ← parent
5:  if Points_list is non-Empty then
6:      Divide bounds into four equally big squares and name them b1, b2, b3, b4
7:      Divide Points_list into four new list called l1, l2, l3, l4 according to in which
        of b1 − b4 they are located
8:      BuildQuadtree(l1, b1, this)
9:      BuildQuadtree(l2, b2, this)
10:     BuildQuadtree(l3, b3, this)
11:     BuildQuadtree(l4, b4, this)
12: end if
```

For each square sent to the client, all possible NN have to be included. This is done with a *rangeQuery* on the extended area.

Opposed to the grid method the client does not need to do any preprocessing. The client simply initiates a query by sending a query request to the server. The server

then starts traversing the quadtree from the top. It does this by asking the client whether he is in the bounds of the quadtree.

Earlier in this section we mentioned that the server issues boolean requests. This is not exactly true as this would not enable the client to efficiently control the two parameters of the common interface. When queried the client can be in the following situations:

1. Client position is not in the quadtree.
2. Client position is in the quadtree, quadtree contains sites, but a satisfactory state is not reached.
3. Client position is in the quadtree, quadtree contains sites, and a satisfactory state is reached.
4. Minimum cloaked area is larger than quadtree.

To be able to communicate this back to the server an enum is created which can have the following values (the parenthesized number indicates which of the above situations this refers to) and should yield the specified action on the server:

NO (1) Proceed by querying siblings.

YES (2) Proceed by querying children.

RESULT (3,4) Proceed by requesting the candidate list of the quadtree from the server.

The pseudo code for the server's querying can be seen in Algorithm 3.6 on page 32. The pseudo code for whenever the server queries the client can be seen in Algorithm 3.7 on page 33.

The entire method is written in pseudo code in Algorithm 3.8 on page 33, and the simplified flow of request-replies can be seen in the sequence diagram of Figure 3.8 on the next page.

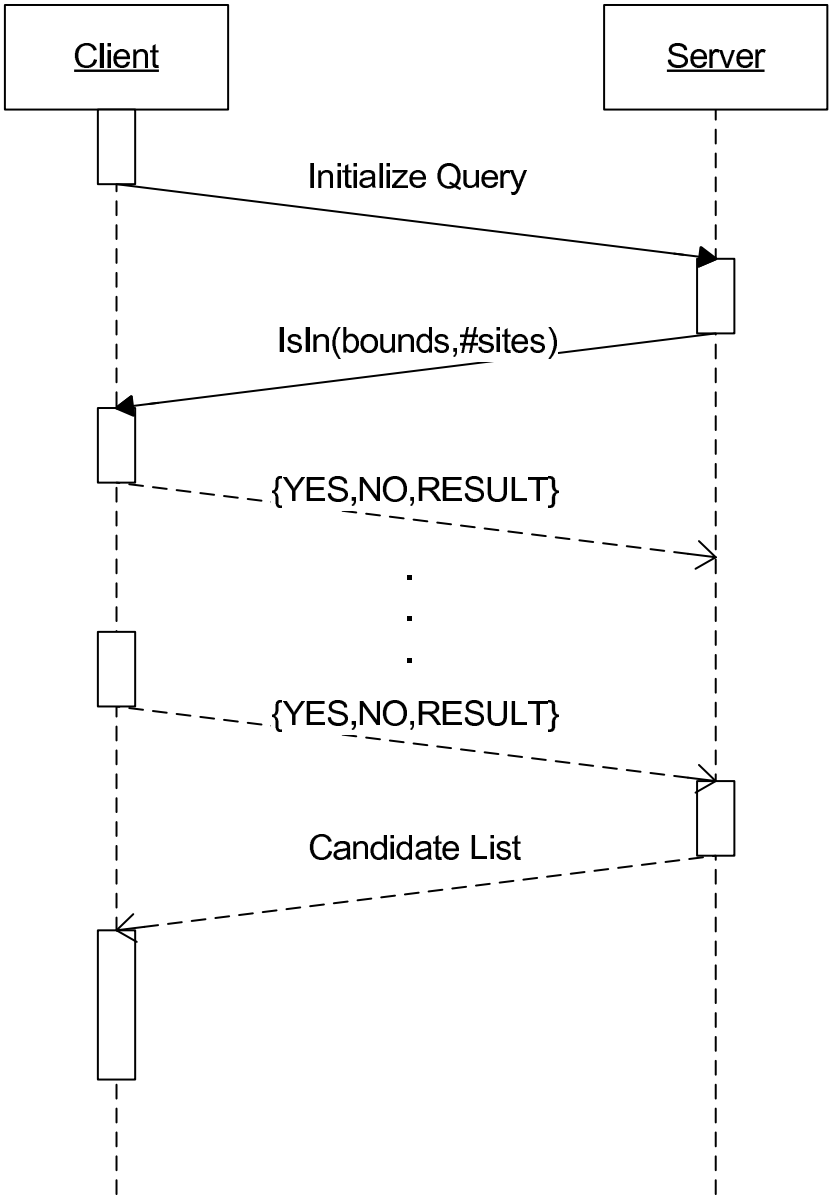


Figure 3.8: Sequence diagram of the quadtree method.

Algorithm 3.6 $FindRegion(quadtree)$

Require: *quadtree.subtree*: Returns the subtrees of *quadtree*.
quadtree.#sites: Returns the number of sites in the subtree.
quadtree.parent: Returns the quadtree which is the parent of *quadtree*.
quadtree.sites: Return the sites in *quadtree*.
$ReplyEnum = YES, NO, RESULT$: An enum which represent client answers.
quadtree.bounds: returns the bounds of the quadtree.
$Client.IsIn()$: A call to the client which return a *ReplyEnum*.

```
1: if quadtree.subtrees == null then
2:    if quadtree.#sites > 0 then
3:       return quadtree.sites
4:    else
5:       return quadtree.parent.sites
6:    end if
7: end if
8: for all subtree in quadtree do
9:    ClientReply ← Client.IsIn(subtree.bounds, subtree.#sites)
10:   if ClientReply == YES then
11:      FindRegion(subtree)
12:      break {Client can only be in one subtree}
13:   else if ClientReply == RESULT then
14:      return subtree.sites
15:   end if
16: end for
```

3.4 K-Anonymity

The two solutions presented offer some user defined degree of privacy. The two methods have that in common that the cloaked area they provide is a square. This is not necessary a bad thing as it gives a high degree of spatial privacy, but having more than one of these square areas as the cloaked area will provide additional privacy. We want to do this by introducing k-anonymity in the methods. Earlier this was defined as having k points with the client position being one of them, and making this position indistinguishable among the k points from the server's point of view. Usually when talking about k-anonymity, the k-1 other positions are other users (e.g. in the Casper framework [12]), which are then made indistinguishable by an anonymizer. In this project we do not use an anonymizer, so these k-1 points have to be created by the client itself. We do this by creating k-1 random points. This does however present a new problem, which will be discussed in the

Algorithm 3.7 $IsIn(bounds, \#sites)$

Require: $Area_{min}$: Minimum cloaked area.
Com_{max}: Maximum communication.
$ReplyEnum = YES, NO, RESULT$: An enum which represent client answers.
$ClientLocation$: Location of the client.
$bounds.contains(location)$: A function that takes a $location$ a return if it is within $bounds$.

```
1: Com_max ← Com_max − 2
2: if bounds.contains(ClientLocation) then
3:    if #sites ≤ Com_max − 2 then
4:       return RESULT
5:    else
6:       return YES
7:    end if
8: else
9:    return NO
10: end if
```

Algorithm 3.8 Quadtree method

```
1: quadtree ← Server calls BuildQuadtree(Points_list, bounds, null)
2: Client initializes the query by sending a request to the server
3: C_list ← Server calls FindRegion(quadtree)
4: Server sends C_list to client
5: Client does nearest neighbor search on C_list
```

following section.

3.4.1 Levels of K-Anonymity

Imagine that a client is stationary and makes a large number of queries over a short amount of time. If the k-1 positions are created randomly for each request it might become apparent for an adversary that one of the positions stay the same. The lower k is, the more this will show. Due to this we propose the following levels of k-anonymity:

Level 1 The k-1 positions are created at random within the query space on each query.

Level 2 The k-1 are created at the initialization of the client, and the client positions actual movement is mimiced in each of the fake positions.

Level 3 The k-1 are created at the initialization of the client. The fake positions move according to a random pattern of movement. This pattern might be a person or a cars movement in terms of velocity and acceleration (depending on the domain of application). To make this possible the client also has to keep track of the time between each issued query.

Level 4 As in level 3, but with the addition of map topology. This might be a 2D map including information on how hard (if not impossible) an area is to cross. In addition this map could also include 3D information such as height.

Adding a k-anonymity layer on top of the developed methods would obviously not just be having a cloaked region plus k-1 points, and the adversary would know that the client is not in any of the points. The client will have to keep the information of all k points and treat them each as the real location in each contact with the server. This entails a requirement for some modifications to the two solutions presented earlier in this chapter.

As stated earlier, in this report we are only concerned with single queries and a Level 1 k-anonymity is therefore sufficient, and this will therefore be examined further. If we were to introduce Level 2 and 3 k-anonymity, this would only entail changes at the client as it at most needs to know human movement patterns. Having Level 4 k-anonymity would require some additional communication with the server as map topology has to be communicated from the server to the client.

Before we take a look at how we can add k-anonymity to the solutions, we have to figure our how we interpret the common interface in this context.

3.4.2 Common Interface and K-Anonymity

In the common interface we defined the two variables *Maximum communication* and *Minimum cloaked area*. In order to make sense of adding k-anonymity we have to figure out exactly how these are interpreted in this context. There are basically two ways of doing this, either dividing or multiplying by k. These two interpretations will now be discussed.

Dividing by K

The first possibility is to divide the parameters by k, so that each of the k-positions gets an equal share in both communication and minimum area. This means that minimum cloaked has to be smaller than or equal to the sum of all k cloaked areas, $Area_{min} \leq \sum_k Area_k$ and that maximum communication allowed is the communication used by all k positions, $Com_{max} \leq \sum_k Com_k$. Strictly speaking dividing by k is the solution that adheres to the common interface the most, as results from this method would be directly comparable to the two methods without k-anonymity. However having a large k might entail a scheme that is more equal to informing the server of k points, than that of k areas. This is illustrated in the Figure 3.10 where k is high. This is in contrast to what would happen with a low k, illustrated in Figure 3.9.

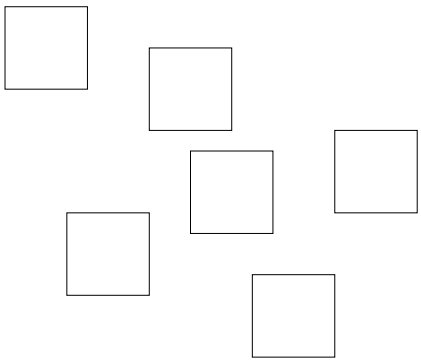


Figure 3.9: Cloaked areas in the divide by k scheme, with a low k.

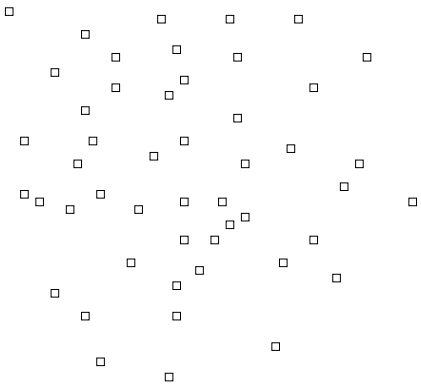


Figure 3.10: Cloaked areas in the divide by k scheme, with a high k.

When doing so one might end up in a situation where all the cloaked areas line up in some way. An example of this can be seen in Figure 3.11 on the following page

where all the cloaked areas are one a line. This will yield a low degree of spatial privacy. However, if k is high enough and the pseudo random generator is good enough this should rarely be a problem.

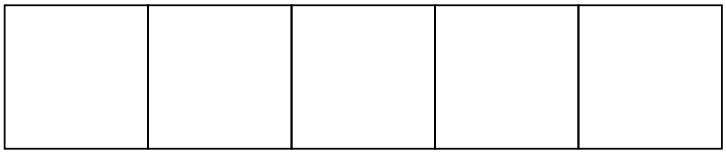


Figure 3.11: Situation where the k cloaked areas are on a line. k = 5.

Multiplying by K

The second possibility is to interpret the parameters as what is allowed to be used by each of the k positions. This will eliminate the problem we have in the *divide by k* scenario with having very small areas. However this approach will also make it hard to compare the k-anonymity implementations of the common interface with other implementations.

As explained there are drawbacks of both multiplying and dividing by k when interpreting the parameters of the common interface to k-anonymity. However it does make the most sense to use the divide by k approach as this makes the implementation more comparable to other implementations. This especially goes for the maximum communication parameter, as it is our main that k-anonymity solutions are not allowed more communication than other methods. Regarding the minimum cloaked area, this can more freely be interpreted by the different clients, so that the possibility of having a lot of very small areas for large ks can be avoided. One should however also keep in mind that having a large cloaked area (connected or not) yield a higher possibility to reach a state of overflow. In the further examination we presume that maximum communication is always divided by k.

Another thing worth examining is how the k queries should be carried out. Should they be separate with a query for each of the k positions or should they be bundled.

Separate Queries

This is simplest of the two ways of querying. Here no modification to the methods have to be provided. The client simply issues a separate query for each of the k points. This will be more costly in terms of communication, as this requires that k is strictly divided into k pieces all of size $kCom_{max} = \frac{1}{k}Com_{max}$. For a large value of k this might result is each of the k queries reaching a state of overflow, all returning all sites within the queryspace. The lower the value of k, the lower the possibility of reaching this state becomes. Still, there can be a lot of wasted communication in querying the same cloaked areas. This is the case in the two methods described earlier in this chapter. They both start out requesting the number of sites in the entire queryspace. In k separate queries this request would therefore be issued k times. However, there is also an advantage of having separate queries. The server has no idea that it is indeed the same client that issues all the requests. This is opposed to bundled queries.

Bundled Queries

In bundled queries we bundle the information of all k positions in one query to the server. At the very least this avoids spending k communication on initialization and the final delivery of the candidate list. This does also have the disadvantage that the server knows that a certain client is in one of the k positions. In relation to the two methods described earlier in this chapter, one would also be able to avoid querying the same cloaked area more than once.

Once again there are pros and cons of each method. If one uses separate queries only small changes has to be made, and they all have to be made on the client. Here the client would have to issue the k queries and then search for the NN among the k result sets. Due to this, what is further examined in this chapter is how to modify the two proposed solutions to include k-anonymity with bundled queries.

3.4.3 Modifications to Grid Method

As explained, we want to add k-anonymity to the grid method using bundling. As the areas queried in the grid method are chosen by the client itself, rather than the server, this is where we want to do the modification. Instead of having an

single square we want a list of k squares. Further more it is decided to divide the minimum cloaked area, so that the sum of all the squares which constitute cloaked areas in a query is greater than or equal to the minimum cloaked area.

In the grid method without k-anonymity, we create an array of querysquares, where the first and biggest one is the bounds of the queryspace and the smallest one is the minimum area (actually the minimum area snapped to the grid). These are then sent to the server one by one in decreasing order. We here propose to convert this in a rather naive way by creating a list of k querysquares for each place in the original querysquares array. Each querysquare in the list of k squares should be created exactly as they are in the grid method without k-anonymity, with one exception.

The first thing to do is to create the list which should be last one in the array. In this step we do not want to send identical or overlapping querysquares. To ensure this, for each position in the array, we create a square of the first of the k positions. We then loop through the remaining points to see if any of these are contained in the same square. For all the positions where that is the case, we do not create an additional square. We however still have to make sure that the sum of the squares is never smaller than minimum cloaked area. If this is the case in a list the squares are expanded in some arbitrary way while still holding the shape of a square. Still, this has to be done without making overlapping squares.

The next thing to do is to create the other list of squares in the array. This is done identical to the method without k-anonymity. We here do allow squares to overlap, and we therefore will end up all lists being the same size, the last one actually having a list of identical squares. In this solution we leave it up to the server to figure out which squares are identical. We do realize that this could be optimized, but as the main scope is not k-anonymity, we will only make a comment on how this could probably be done. Just as in the first list we created one can imagine that the same could be done to see if a small square is contained in a larger square. To further optimize this, one could imagine that it is possible to find an algorithm that would create n big squares encapsulating m smaller squares, where $n \leq m$.

Other than the creating of the square we need to modify the server so that it takes an array of querysquare lists rather than an array of querysquares. As this can be done by doing the same as the server does in the former described method by looping through the querysquares in each request, this is considered trivial, and will not be examined further.

3.4.4 Modifications to Quadtree Method

Opposed to the grid method the quadtree method is based on that the server has control over which areas are queried at a certain time. This means that we cannot modify the squares which are queried, as the server only knows the replies it gets from the *IsIn* queries. As k-anonymity is not the main scope of this project and in realization that it would require a lot of work to get the quadtree method efficiently with bundled queries, we propose that adding k-anonymity to the quadtree should be done using single queries for each of the k-positions. This should be done with the parameters minimum cloaked area and maximum communication both set to $\frac{1}{k}$ of the original value. The client should then discard all the answers but the one concerning the true client position, and should do NN on the candidate list retrieved from that query.

CHAPTER 4

Test

This chapter describes the testing of the proposed solution. We want to test how the two solutions perform in terms of how much communication is used with different settings of the parameters of the common interface. The main purpose is to see how data-dependency effects the test output. The chapter will furthermore include implementation details of the solution, a test setup, a discussion of expectations to the tests, test results, and in the end a discussion of the results.

In terms of what the results of these experiments can be compared to, the answer is none. What we have proposed in this report is a solution which is based on a common interface, which is not seen before with the requirements stated in the problem statement in Section 2.7 on page 17. The most similar thing was seen in [12], but as it was based on a different architecture, the tests are not comparable. We therefore only compare the test results internally.

4.1 Implementation Details

This section will discuss the implementation details of the test. This will include a description of the environment, the limitations of this specific implementation, and how the implementation differs from the design proposed in the solution chapter.

4.1.1 Environment

This will explain the details of the hardware and software environment in which the tests were performed.

Hardware

The tests were carried out on an IBM R50 ThinkPad, type 1829-7RG:

- Pentium M 1.5 GHz
- 512 MB RAM
- Windows XP Professional

For a full specification see [4].

Software

The implementation was done using the *Eclipse* Integrated Development Environment (IDE) version 3.3 (for details see [3]). The test implementation was compiled and run using *Java SE 6* (see [1]). Furthermore a 3rd party package was used to create the Zipf distribution (explained in Section 4.2.1 on page 46). This package is called *Colt* (see [2]), and contains a lot of functionality for scientific and technical computing in Java.

4.1.2 Changes from Design

To implement the solutions presented, some changes and decisions had to be made. This was done to make the implementation more practical and realistic.

Quadtree

The main problem with the quadtree solution proposed, was that it includes a lot of recursion. The datastructure is in itself recursive, and the algorithms used are recursive as well. In an implementation this presents a problem as a lot of recursive calls might lead to a stack overflow. In the quadtree we call the *BuildQuadtree* (Algorithm 3.5 on page 29) recursively until each site is put in it's own node. If the distance between the points is close to zero, the number of recursive calls on the stack is close to infinity (if the points are identical, i.e., the distance between them

is zero, we have another problem which we will deal with later). In an implementation we do however have a discrete space, and therefore a minimum possible distance between two points. This can however still lead to a stack overflow.

Instead of pushing the recursive calls on the stack, we propose a solution where a queue is utilized in place of the stack. This is however only necessary when we need to examine more than one subtree of the quadtree. In algorithms such as *FindRegion* (Algorithm 3.6 on page 32), only one subtree needs to be be examined. The same approach can however still be used to avoid recursion.

In Listing 4.1, the test implementation source code of *RangeQuery* (Algorithm 3.4 on page 29) is shown. This is implemented iterative, i.e., without recursion, opposite to the pseudo code version.

```
1  public Vector<MultiplePoint> rangeQuery(Rectangle range) {
2    LinkedList<Quadtree> queue = new LinkedList<Quadtree>();
3    Vector<MultiplePoint> containedPoints = new Vector<MultiplePoint>();
4
5    queue.add(this);
6    while(!queue.isEmpty()) {
7      Quadtree tree = queue.poll();
8      if(tree.getSubtrees() != null) {
9        for(Quadtree subtree: tree.getSubtrees()) {
10         if(range.intersects(subtree.getBounds()) {
11           queue.add(subtree);
12         }
13       }
14     }
15     else {
16       if(tree.getPoints().size() > 0) {
17         for(MultiplePoint mp: tree.getPoints()) {
18           if(range.contains(mp.getPoint()) {
19             containedPoints.add(mp);
20           }
21         }
22       }
23     }
24   }
25   return containedPoints;
26 }
```

Listing 4.1: RangeQuery implemented in Java without recursion.

In line 2 the queue is initialized as a `LinkedList` of `Quadtree`s. This collection is used as it has methods for both queue and stack operations. Queue operations are `add(object)` and `poll(object)`, which adds an object (of class `Quadtree`) to the end of the list and removes and returns the first object, respectively. Stack operations are `push(value)` and `pop(value)`, which puts a value in beginning of the list and removes and returns the first object, respectively. This means that the behavior of the list can easily be changed from a queue to a stack. In other word, utilizing a queue yields a breath-first approach, and utilizing a stack yields

a depth-first approach. As recursion uses the program stack, utilizing the stack behavior would do the exact same thing as a recursive call does. Instead we use the queue behavior. In line 5 the top level of the quadtree is added. In line 6 a while loop starts running until the queue is empty. We then, in line 7, poll an element from the queue, and if the quadtree of the current loop has subtrees, we add the subtrees which intersect the parameter `range` to the queue (lines 8-14). If the quadtree does not have any subtrees, we return the points of the quadtree which are contained in `range` (lines 15-23). Finally in line 25, we return the result.

As explained in the above the solution, we are also presented with a problem in case two or more points are identical. In real life this might be the case if sites represent stores. Depending on the measurement unit of the data set, two stores might be located on the same coordinate. This could e.g. be a mall. We therefore have to take make sure that the quadtree can be constructed even with two identical sites. If we use the pseudo code version from Algorithm 3.5 on page 29, we will end up in an infinite loop, where the quadtree is looking for a quadtree in which the two points are located in two different subtrees. To avoid this we add some additional information to a point. Besides having a point in a two dimensional space (represented by two coordinates), we add the number of occurrences to the point. When constructing the tree, the algorithm will regard this as a single point. However, we need to remember to count the all occurrences of the point, when returning a candidate list to the client. If this is not done we will end up in with a case where the total number of sites in the data set will be less than the original data set. In essence, this means that a single quadtree now can hold an infinite number of points, but now this is not a problem. One might encounter a problem when adding meta-data to the sites, e.g. *store name*, *opening hours*, etc. However, this is outside the scope of this project, so we will disregard this. In the test implementation, the point with number of occurrences is represented as the `MultiplePoint` class used in Listing 4.1 on the preceding page.

Simplification

We want to make the implementation rather simple, so we will get fewer points of error. To do this, we choose to make the experiments with the simplified extended area (explained in Section 3.2 on page 21).

Data Sets and Queryspace

Later, in Section 4.2.1 on the following page, we will describe the different data sets and how they will be used . We based our choice of number of sites and the size of the queryspace, on the real life data set which is approximately 120,000 points, within a space of $1,000,000 \times 1,000,000$ possible locations.

4.1.3 Limitations

As the main aim of these tests were to examine the communication used, and not performance in terms of seconds, no time is spent on optimizing the internals of the different methods. Furthermore, nothing is done to make the datastructures in each of the solutions persistent. I.e. they should only be kept in memory. In addition we do not want to use time on optimizing things such as space consumption and operations on lists, etc.

In regards to the use of Java, we want to run the Java Virtual Machine with standard parameters. After implementing the tests, the choices mentioned above led to a number of limitations.

Quadtree

Earlier in this chapter we mentioned that the we might get a very high number of recursive calls on the stack. We also presented a solution to how this can be fixed. Even with our limitation of the queryspace and the data sets, this will still lead to a situation where at least 120,000 nodes of the quadtree have to be created, as only one point can be stored in each. Furthermore, 120,000 only represents the leaf levels of the entire tree, so actually we store more than 120,000 quadtree. This will often lead to a heap overflow. To avoid this we allow up to 100 points in each leaf-node. In addition we want to treat identical points as one.

Grid

As we have a space of $1,000,000 \times 1,000,000$ possible points, the maximum number of grid cells would be 10^{12}. As each cell holds a list of points, and some

additional information, having 10^{12} grid cells also quickly leads to a heap overflow. Using experiments, an appropriate size of the grid cells was found to be $1,000 \times 1,000$ points, leaving us with $1,000 \times 1,000$ grid cells.

K-Anonymity

As mentioned in the solution chapter, k-anonymity is not the main goal of this project, so we chose not to experiment with this.

4.2 Test Configuration

This section describes the details of the data sets used in the tests. Furthermore, the specific tests are specified.

4.2.1 Data Sets

We want to test the solutions with three different kinds of data sets of approximately the same number of sites:

Real World Data set A data set of postal addresses in the three metropolitan areas of New York, Philadelphia and Boston in the North East of the U.S. [13]. Contains 123,593 sites. Depicted in Figure 4.1 on the next page.

Uniform Distribution Randomly created uniform distribution with 120,000 sites. Depicted in Figure 4.2 on page 48.

Zipf Distribution(ρ) Randomly created Zipf distribution [14], with the skew parameter ρ. Also with 120,000 points. The distribution will be created with

$$\rho = 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 1.7, 1.8, 1.9, 2.0$$

The distribution is often used in the testing spatial data set algorithms. This is due to the fact that it is a distribution with a variable degree of skew. Zipf distributions with skew factor 1.1 and 1.7 can be seen in Figure 4.3 on page 48 and 4.4 on page 49, respectively. The reason that there does not seem to be a lot of points in Figure 4.4, is because most of them are

concentrated around $(0,0)$. The remaining Zipf distributions can be seen in Appendix A on page 71.

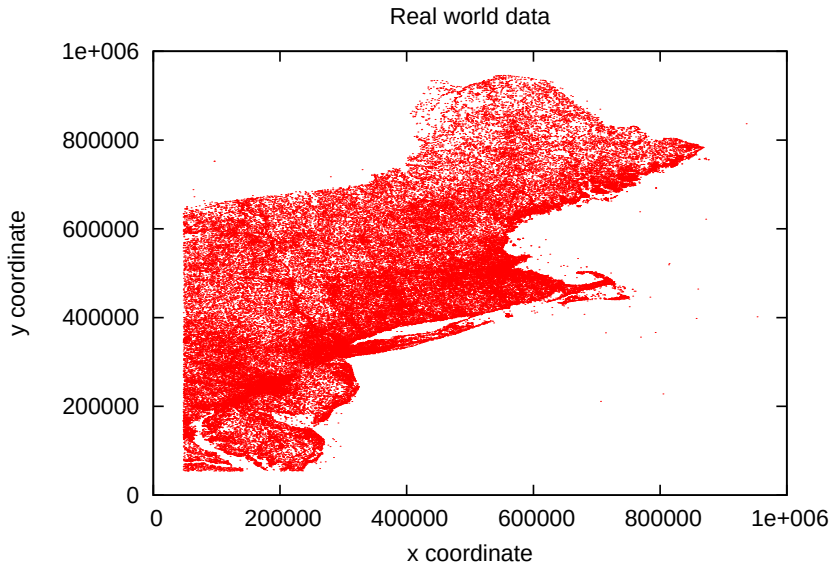


Figure 4.1: Real world data set containing 123,593 postal addresses represented as points.

4.2.2 Client Position

In order to make useful results we want to have more than one client point position. For each test we use the same 100 evenly distributed client query points. The result is then the average of the test results from querying the 100 points. The client positions can be seen in Figure 4.5 on page 49.

4.2.3 Test Configurations

The overall goal is to test how different things affect the communication used. This should be done testing the different variables. In the end this should result in a graph for each test.

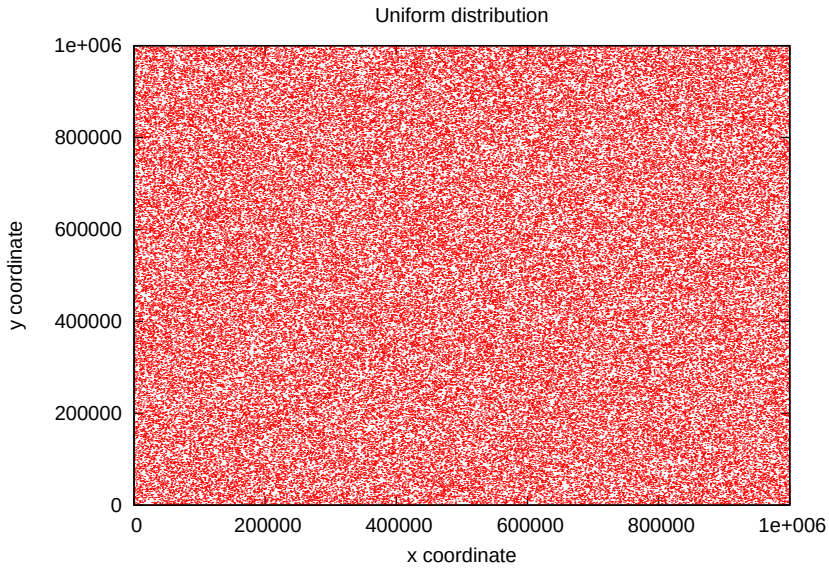


Figure 4.2: Uniform distribution containing 120,000 points.

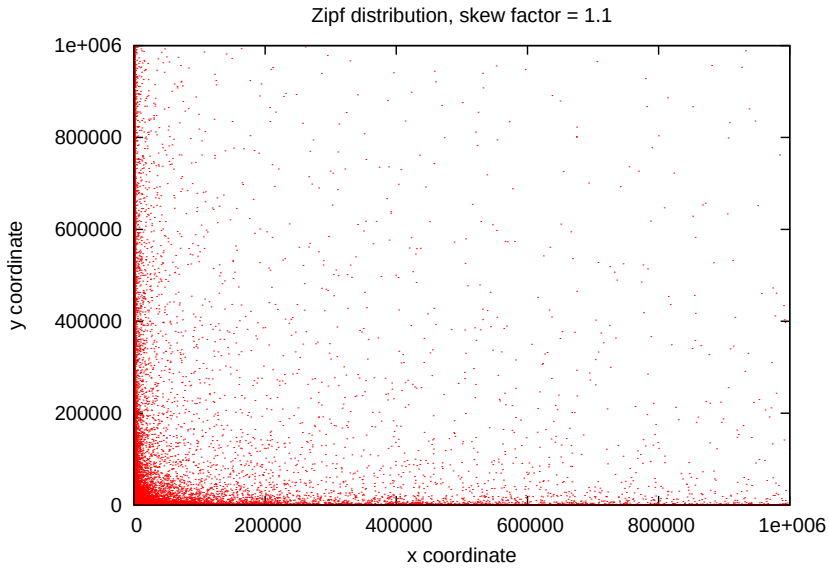


Figure 4.3: Zipf distribution with skew factor 1.1, containing 120,000 points.

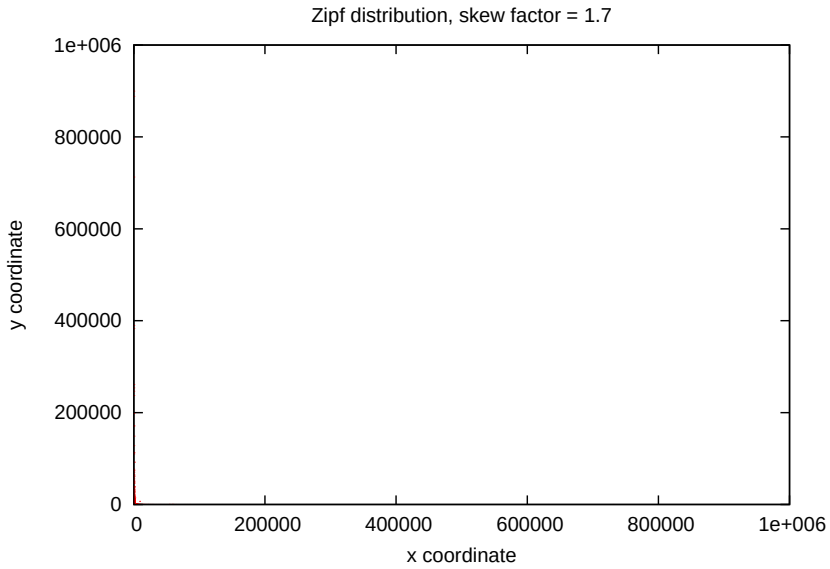


Figure 4.4: Zipf distribution with skew factor 1.7, containing 120,000 points.

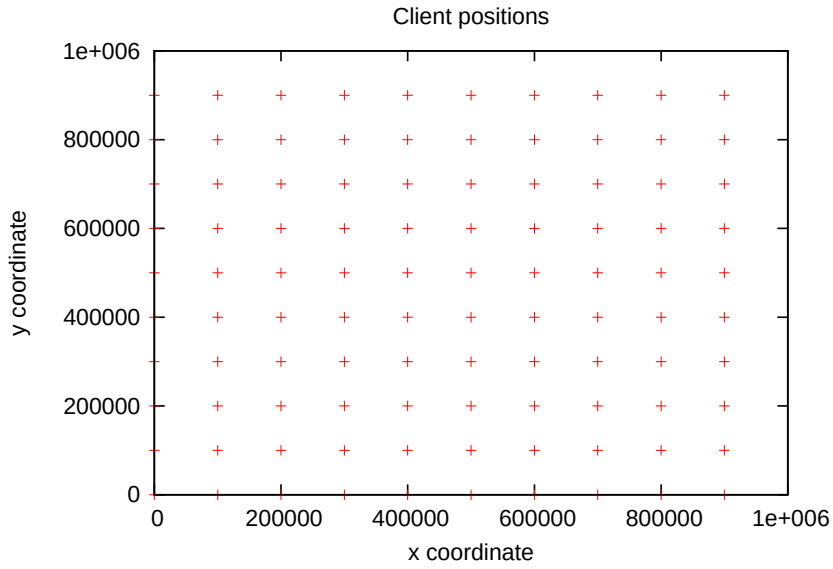


Figure 4.5: The 100 client positions of which the average resulted will be used.

Test 1

This test will determine the relation between the maximum communication and the communication used. It will do this with minimum cloaked area set to two different values, being: 1 and $100,000^2$.

The test is carried out on all data sets, each resulting in a graph. We will do two skew factors of the Zipf distribution, these are 1.1 and 1.7.

x: Maximum communication

y: Communication units used

Test 2

This test will determine the relation between the parameter minimum cloaked area, and the amount of communication used. The maximum communication is set to two different values, being: $30,000$ and $5,000$.

The test is carried out in the same way as Test 1. The parameters are defined as:

x: Minimum cloaked area

y: Communication units used

Test 3

The purpose of this test is to examine how different values of the skew factor ρ in the Zipf distribution affects the communication used. The values of maximum communication and minimum cloaked area are set to two different values for each of the solutions. These two sets are as follows:

(maximum communication, minimum area) = $(40,000\ ,\ 1)$

(maximum communication, minimum area) = $(5,000\ ,\ 200,000^2)$

These four result sets has to be depicted in the same graph. The parameters are:

x: ρ in $Zipf(\rho)$

y: Communication units used

4.3 Test Expectations

Our expectations of the tests are based on an idea that using a data-dependent structure should perform better in distributions with a high skew. I.e., perform better when the distribution of points is very uneven throughout the queryspace. It is apparent from the description of the two methods that the quadtree requires more communication to get to the square (bounds of a quadtree) in which the client is located, compared to the grid method which always yields a square in which this property is satisfied. We do however expect this difference to be more than evened out with the benefit of constructing the datastructure based on the density of points in a certain area. In relation to the different data sets, we would expect the algorithms to perform as follows:

Uniform distribution We expect the algorithms to perform very similar, with a minor advantage to the grid method over the quadtree method based on the lower communication between client and server.

Zipf Distributions The higher the skew of the distribution, the better we would expect the quadtree method to perform compared to the grid method.

Real World Data Set As the density of sites vary in different areas of the queryspace, we will once again expect the quadtree method to outperform the grid method.

4.4 Test Results

The following section contains the test results.

4.4.1 Test 1

The following contains the test results from test 1. q denotes the quadtree method, and g the grid method. The sub scripted numbers represent the two different values

the minimum cloaked area parameter have been set to. In this case it is defined as follows:

1. Minimum cloaked area = 1
2. Minimum cloaked area = $100,000^2$

An additional line has been added. This is the straight line. This represents the maximum communication allowed. This mean that when ever a line is over this line, the method has overflown.

Real World Data

The graph for the real world data set can be seen in Figure 4.6. The graph shows

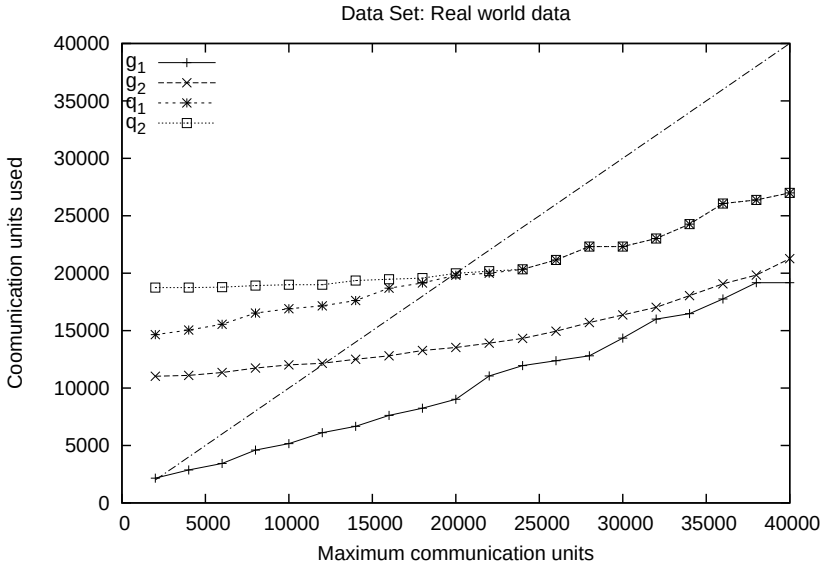


Figure 4.6: Test 1 results with real world data set.

that the grid method performs better than the quadtree method with both values for minimum cloaked area. g_2 even performs better than q_1. g_1 is the only method that always reaches a satisfactory state, i.e., a state without overflow (from our lowest test value of 2,000). g_2 reaches satisfactory states with values over approximately 12,000, and both q_1 and q_2 return satisfactory states with values over 20,000.

Uniform Distribution

The graph for the uniform distribution data set can be seen in Figure 4.7. As

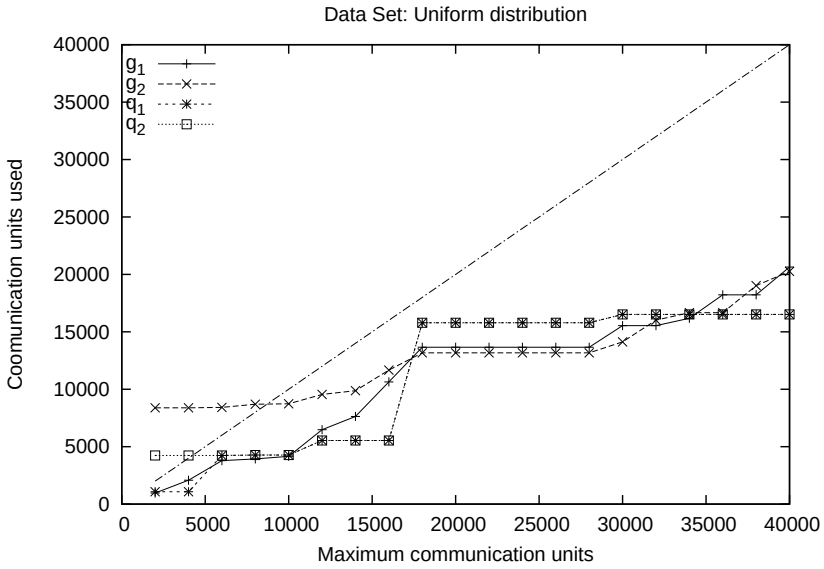


Figure 4.7: Test 1 results with uniform distribution data set.

the graph shows the grid method and quadtree method perform nearly identical. Opposed to the real world data set, the quadtree actually has a small advantage. Neither g_1 or q_1 overflows with the tested values. g_2 stops overflowing around 9,000, and q_2 around 4,000.

Zipf Distribution - 1.1

Figure 4.8 on the next page shows the Zipf distribution with skew factor 1.1. As shown on the graph, the grid method performs remarkably better than the quadtree method. g_1 never overflows and g_2 stops around 7,000, whereas q_1 stops around the maximum test value, and q_2 even later.

Zipf Distribution - 1.7

Figure 4.9 on the following page shows the Zipf distribution with skew factor 1.7. In this figure we see that the amount of communication used is approximately the

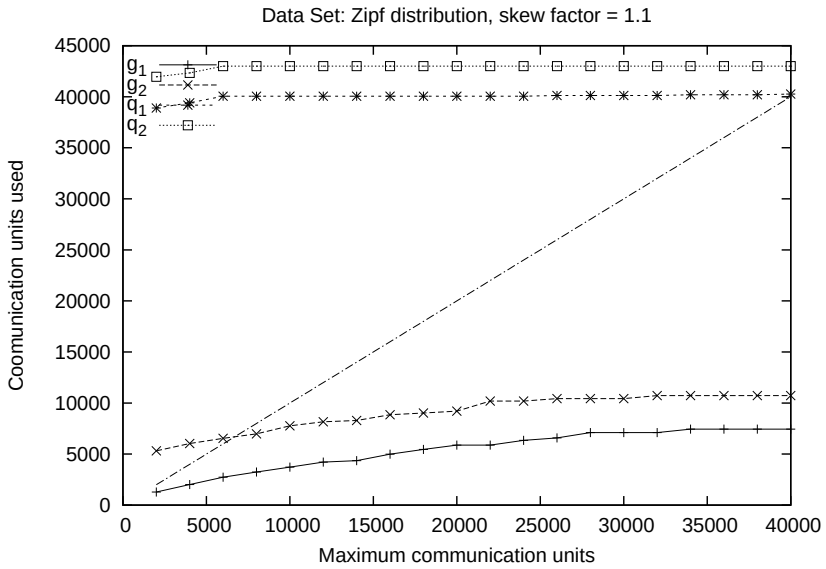


Figure 4.8: Test 1 results with Zipf distribution with skew factor 1.1 data set.

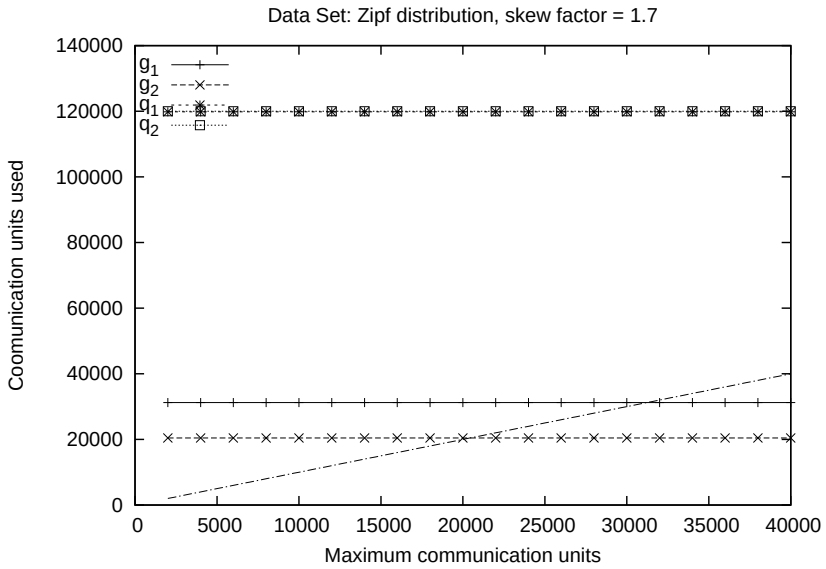


Figure 4.9: Test 1 results with Zipf distribution with skew factor 1.7 data set.

same no matter that we specify as maximum communication. This is due to the fact that all the sites are concentrated around $(0,0)$. As in the Zipf distribution with skew factor 1.1, the grid method clearly outperforms the quadtree method. g_1 stops overflowing at 21,000, and g_2 at 31,000. Both q_1 and q_2 return something close to the entire result set no matter the value of maximum communication.

4.4.2 Test 2

The following contains the test results from test 2. As in test 1, q denotes the quadtree method, and g the grid method. The sub scripted numbers represent the two different values the maximum communication parameter have been set to. In this case it is defined as follows:

1. Maximum communication = 30000
2. Maximum communication = 5000

Two additional lines have been added. These are the horizontal lines. These represent the two values of max communication. In some of the graphs the highest of these lines ($y = 30000$) is actually the top of the diagram. When g_1 and q_1 are over the highest of these lines they overflow, and g_2 and q_2 when they are over the lowest horizontal line.

Furthermore we choose to depict the graphs with a x-axis of $(\text{minimum area})^2$ as this will then represent the width of the cloaked area square.

Real World Data

The graph for the real world data set can be seen in Figure 4.10 on the next page. The figure shows once again that the grid method performs better than the quadtree method. Neither g_1 or q_1 overflow with the tested values. They will however do this eventually as the value of minimum cloaked area gets close to the entire queryspace. This is due to the fact that eventually you will reach an area where all the points are included, and as each of these require one communication unit, it will overflow. g_2 starts overflowing when the value reaches $51,000^2$ and q_2 will never be in a satisfactory state with maximum communication set to 5,000.

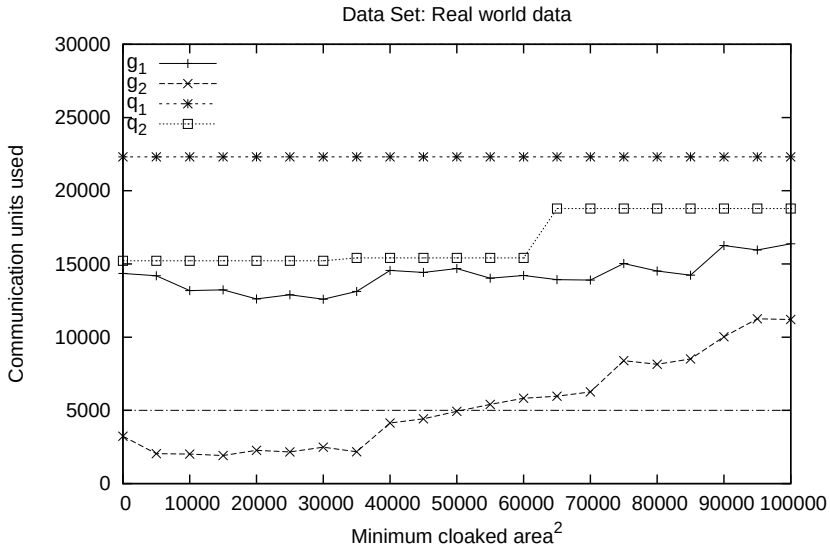


Figure 4.10: Test 2 results with real world data set.

Uniform Distribution

Figure 4.11 on the facing page shows the uniform distribution data set. As in test 1, the figure shows that when querying a uniform distribution, the quadtree method has an advantage over the grid method. g_1 is a little lower than q_1, but neither of them overflow within the tested values. They will however do this eventually due to the reasons mentioned in the former section. q_2 always reaches a satisfactory state with the tested value. g_2 starts overflowing when the minimum cloaked area is greater than $76,000^2$.

Zipf Distribution - 1.1

Figure 4.12 on the next page shows the Zipf distribution data set with a skew factor of 1.1. In this figure, it is clear that the grid method performs much better than the quadtree method. g_1 never overflows with the tested values, and g_2 starts overflowing around $75,000^2$. The difference in maximum communication does not affect the quadtree method much, and both q_1 and q_2 overflow in all situations.

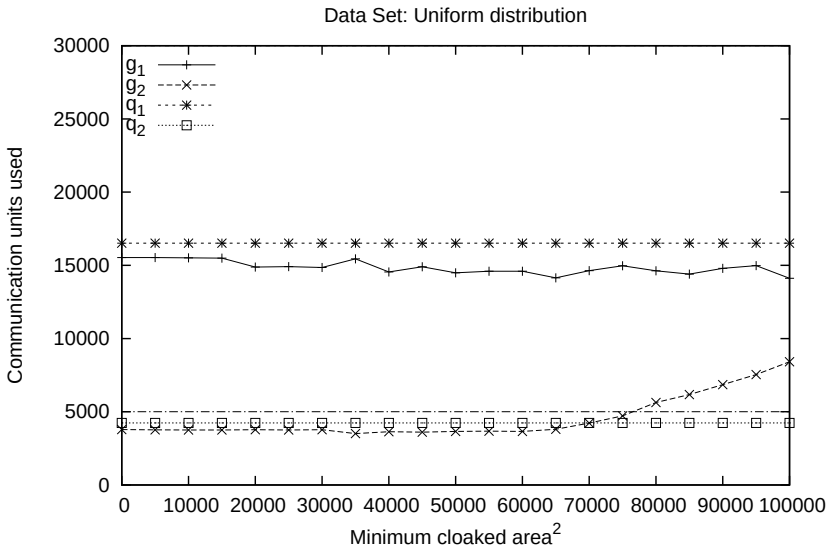


Figure 4.11: Test 2 results with uniform distribution data set.

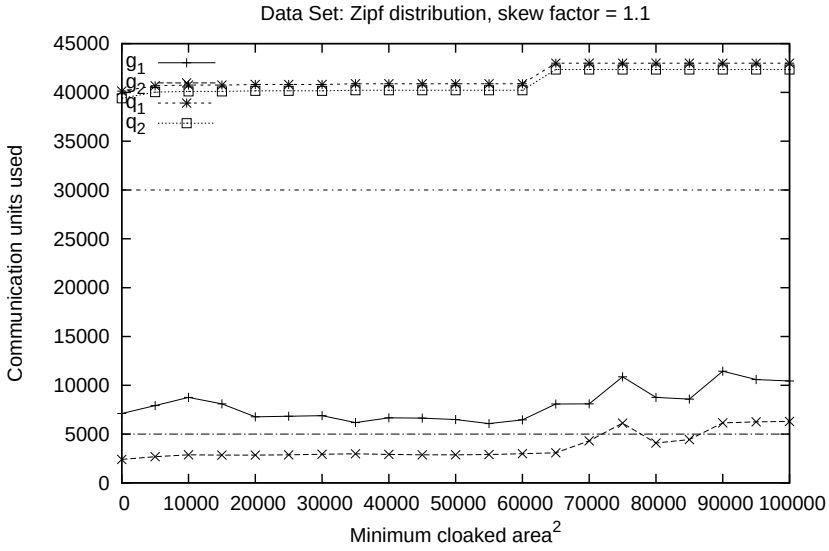


Figure 4.12: Test 2 results with Zipf distribution with skew factor 1.1 data set.

Zipf Distribution - 1.7

Figure 4.13 shows the Zipf distribution data set with a skew factor of 1.7. In this

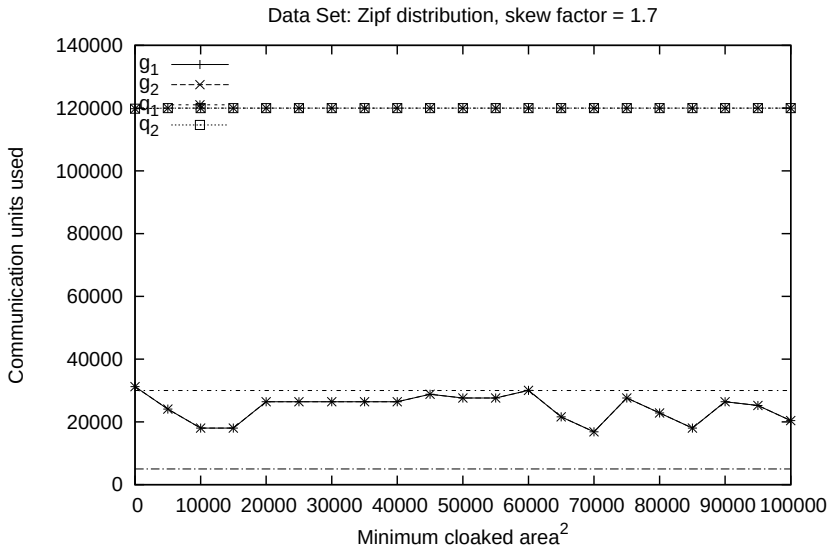


Figure 4.13: Test 2 results with Zipf distribution with skew factor 1.7 data set.

figure, we see that the two different values of maximum communication does not effect how the two different solutions perform. As in test 1, this data set makes the quadtree method return all the sites of the queryspace. $g1$ and $g2$ both spends around 30,000 communication units with the tested values of minimum cloaked area. This means that g_1 reaches a satisfactory state, and g_2 overflows.

4.4.3 Test 3

The following contains the test results from test 3. As in test 1 and 2, q denotes the quadtree method, and g the grid method. The sub scripted numbers represent the two different sets of values of the minimum cloaked area parameter and maximum communication have been set to. In this case it is defined as follows:

1. Maximum communication = 30000 and minimum cloaked area = 1
2. Maximum communication = 5000 and minimum cloaked area = 200000^2

Two additional lines have been added. These are the horizontal lines. These represent the two values of max communication. When g_1 and q_1 are over the highest of these lines they overflow, and g_2 and q_2 when they are over the lowest horizontal line.

Figure 4.14 shows the results of test 3. As can be seen in the figure, the values of

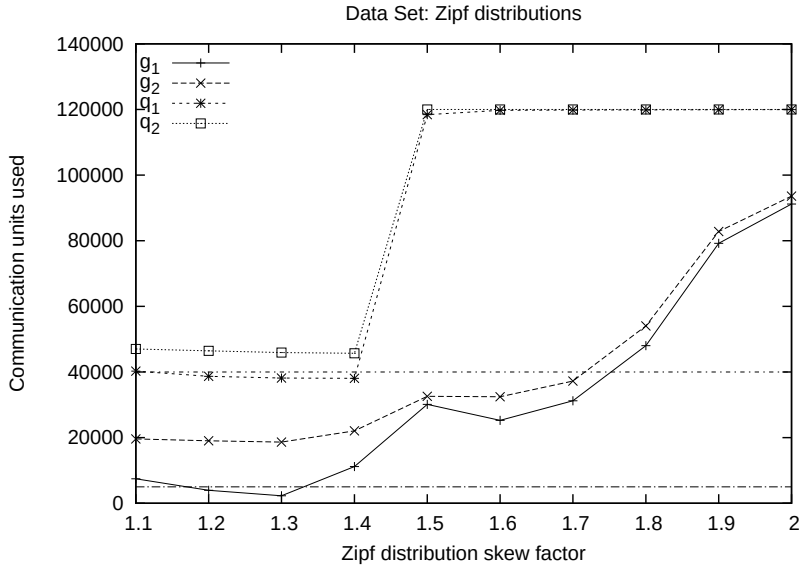


Figure 4.14: Test 3 results.

case 2 will never reach a satisfactory state, neither with the quadtree method or the grid method. g_1 start overflowing when the value of the skew factor exceeds 1.7. q_1 overflows when the skew factor exceeds 1.4. Once again this means that the grid method performs better than the quadtree method. This can also be seen on the different lines in the graph. The communication units used by the grid method climbs slowly, but as soon as the skew factor reaches 1.5, the quadtree method is close to returning all the sites of the queryspace.

4.4.4 Reflection on Results

As the result show the quadtree method which we expected to perform better than the grid method actually performs worse on every data set besides the uniform distribution.

One could expect that the simplification we made in Section 3.2 on page 21, might

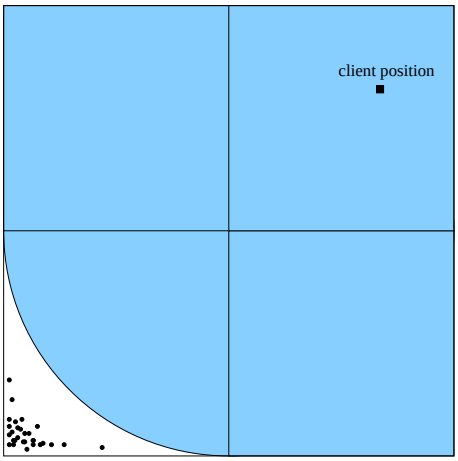


Figure 4.15: The extended area using the simplification.

be responsible.

On Calculating Extended Area

We want to examine how different ways of calculating the extended area has in relation to the candidate list. We want to examine this is relation to the Zipf distributions, as these gave the worst performance of the quadtree method. We want to do this, as we want a method which perform good no matter how sites are distributed.

The Simplified Method As mentioned above, the current implementation is done using a simplification. This simplified method has the advantage that it can be calculated without taking the location of sites into account. This does however have the disadvantage that we might not always get the exact answer to a query. In relation to the quadtree method and the Zipf distribution, we might often find ourselves in a situation where all the sites of the queryspace need to be returned. Such a situation is illustrated in Figure 4.15. Due to the Zipf distribution all sites are concentrated around $(0,0)$. Using the simplified method, this will entail that when the client is located in the top right square, none of the points will be in the extended area, and we therefore need to return the candidates from one level higher in the quadtree, yielding the entire set of sites within the query space.

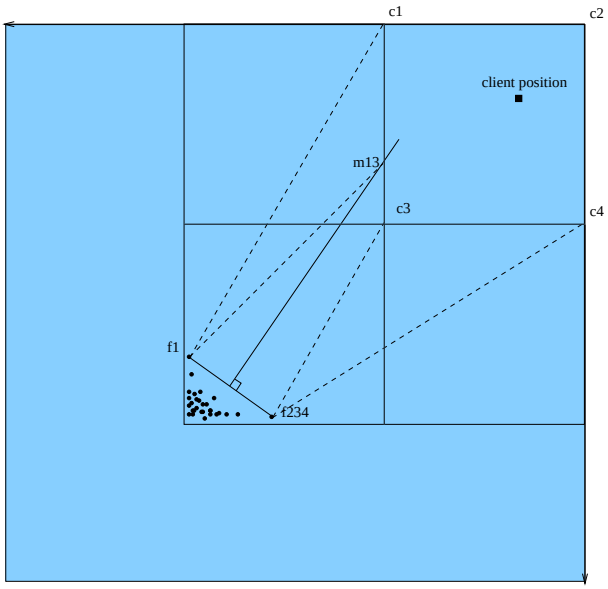


Figure 4.16: The extended area using the method from [12]. We here identify the two filter sites, and using these calculate how much the area should be expanded in direction of e_{34} and e_{13}

The Casper Method In [12] a solution to calculating the extended area is proposed. This is also presented in Section 2.6.1 on page 10. Compared to the simplified method, the Casper method takes the position of sites into account. This should entail an optimized extended area. In Figure 4.16 it is illustrated how the method performs in Zipf distribution also used to illustrate the simplified method. As we see, the way the area is extended will also entail that all sites of the queryspace will need to be returned.

In this method they claim that the method is inclusive and minimal. While being inclusive, we found a problem in the way they reason about their solution being minimal. We therefore propose an extension to the Casper method.

The Extended Casper Method The problem lies in the case where the two corner on a line have the same filter site, i.e., $f_1 = f_2$ for e_{12}. They claim that the area should be extended with the maximum distance from the filter site to one of the two corner, i.e. $MAX(Length(fc_1), Length(fc_2))$. This is however not the case. We only need to expand the area to the filter site. The situation is illustrated in Figure 4.17 on the following page. In the figure the bold line is the line to which the Casper method suggests that the area should be extended. It is however only necessary to extend the area to the dashed line on which the filter site is. By

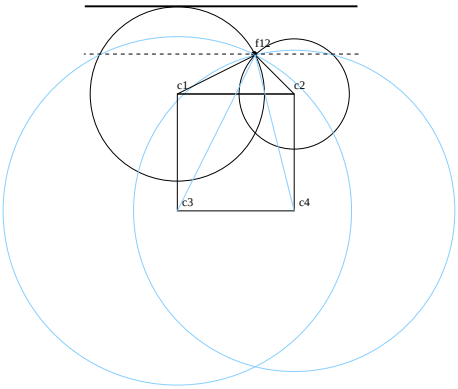


Figure 4.17: This figure indicates how the method mentioned in [12] has a problem. They propose that the entire area under the bold tangent should be included as extended area, whereas only the area beneath the dashed horizontal line needs to be included.

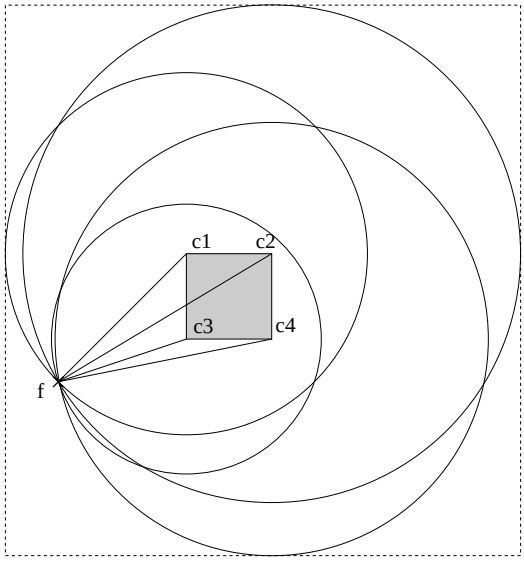


Figure 4.18: The case when all corner (c_1-c_4) have the same filter site, i.e., only one filter site exists. This is f on the figure. The grayed out area is the square in which the client is located. The dashed line represents the area the method in [12] considers minimal.

adding circles from the corners c_3 and c_4, we see that they never go outside the dashed line without being enclosed in one of the circle of c_1 or c_2. This mean if a site was to be located in the areas which exceed the dashed line, one of the corners c_1 and c_2 would have another filter site.

This extension to the Casper method also leads to situation in which all the corners have the same filter site. In this situation only the filter site would have to be returned. This is illustrated in Figure 4.18. All corners have f as their filter site.

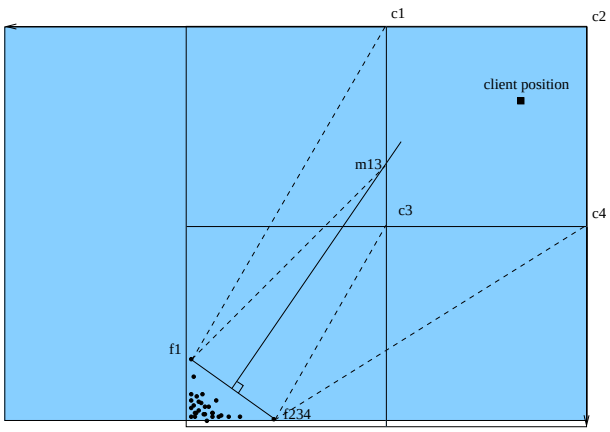


Figure 4.19: The extended area using the extended casper method.

The dashed outer region is the region that the method in [12] would give as a result. However, all possible locations in the original area will always be closer to f than any other site. This is true as no circle with center O, where O is a point within the cloaked area, with a radius the length of Of will go outside the corner circles depicted in the figure. Because of this only the site f have to be returned as the candidate list whenever all corners have the same filter site.

Figure 4.19 shows how the extension affects the extended area in the Zipf distribution from before. Here we see that it is no longer necessary to return all the sites of the queryspace.

Which Solution Should be Used?

From the test results, it is clear that the way the solutions are implemented at the time of testing, the grid method is to prefer. Even though the quadtree method has a slight advantage in a uniform distribution, but this is probably just a coincidence. The grid method performs much better in all other situations, so it is always to prefer.

How the parameters of minimum cloaked area, and maximum communication should be set is quite difficult to answer, as one effects the other. It is however clear that increasing the minimum area does not have a very big impact on the candidate list (remember, minimum area in the graphs are actually the square of the minimum area). It should however be increased slightly. Setting the maximum communication to around 20,000 should in most cases be able to provide a non-

overflowing answer. The only time this is a problem is in Zipf distributions with a high skew factor. This is however mainly due to the fact that a lot of points will have identical coordinates.

Imagine the border case with a skew factor so high that all 120,000 points are identical and located at $(0,0)$. No matter what, both solutions would always return all site data, as they would all be the nearest point. In the Zipf distribution with skew factor 2.0 used in this implementation, we have $(0,0)$ represented 44,460 times. Depending on the application domain in which the solutions could be deployed, the data sets might be constructed so that points could never be identical.

CHAPTER 5

Conclusion

In this report a common interface for flexible privacy preserving in LBSs has been documented. This was done under the topic of *Privacy in Pervasive Computing*.

Emphasis was put on examining the different aspects of already known solutions to the problem. This was done through an examination of related work. Through this examination it was chosen to focus on a previously unexplored area within the limits of the topic. This was done by focusing on the quality attributes of *flexibility*, *security*, *accuracy*, and *complexity*, and ended up in a problem statement in the form of a list of formal requirements to the solution.

Next, a large amount of work was put on developing a common interface which would enforce the problem statement. Here it was chosen that a client should be able to specify his privacy settings in form of *maximum communication* and *minimum cloaked area*. In the further development of the solutions it was decided to give the user privacy in the form of a cloaked area.

Two solutions were developed. This was mainly done to focus on the difference between having a data-dependent and a data-independent datastructure at the server. Furthermore the solutions are examples of both client based and server based approaches. The methods developed was one based on a grid, and one based on a quadtree. In addition it was explained how adding an extra layer in form of k-anonymity could be beneficial.

To reflect on how the two solutions developed would perform against each other, a testing implementation was done. It was chosen only to implement the two methods without k-anonymity. The tests were done with the purpose of testing how much communication the different solutions would with different data sets. The data sets utilized was a real world data sets, a uniformly generated data set,

and several skew data sets using a Zipf distribution. This was done in a theoretical setting with the communication between client and server simulated. The results of the tests were at first glance surprising, but was later explained with a limiting design decision on how to create the extended area. This affected the software quality attribute of *accuracy*. The problem was examined and a solution to how this could be changed, was presented. In the current implementation, it was found that the grid method would always be preferable.

Overall it is found that the common interface developed contributes with a secure and flexible method for privacy preserving queries in pervasive computing. This was due to the fact that the method has no apperent insecurities, and that the privacy settings can be changed from query to query. Furthermore, even though the grid method performed better, it is concluded that implementing the optimization for calculating the extended area, might change this fact.

Bibliography

[1] Java se 6. http://java.sun.com/javase/.

[2] The colt project, November 2007. http://dsd.lbl.gov/ hoschek/colt/.

[3] Eclipse project, November 2007. http://www.eclipse.org/.

[4] Ibm r50 type 1829-7rg, November 2007. http://www-307.ibm.com/pc/support/site.wss/quickPath.do?quickPathEntry=18297RG.

[5] M. J. Atallah and W. Du. Secure multi-party computational geometry. In *WADS '01: Proceedings of the 7th International Workshop on Algorithms and Data Structures*, pages 165–179, London, UK, 2001. Springer-Verlag.

[6] M. J. Atallah and K. B. Frikken. Privacy-preserving location-dependent query processing. In *ICPS '04: Proceedings of the The IEEE/ACS International Conference on Pervasive Services (ICPS'04)*, pages 9–17, Washington, DC, USA, 2004. IEEE Computer Society.

[7] R. A. Finkel and J. L. Bentley. Quad trees: A data structure for retrieval on composite keys. *Acta Inf.*, 4:1–9, 1974.

[8] B. Gedik and L. Liu. Location privacy in mobile systems: A personalized anonymization model. *icdcs*, 00:620–629, 2005.

[9] D. G. Kirkpatrick. Optimal search in planar subdivisions. *SIAM J. Comput.*, 12(1):28–35, 1983.

[10] M. Kolahdouzan and C. Shahabi. Voronoi-based k nearest neighbor search for spatial network databases.

[11] M. R. Kolahdouzan and C. Shahabi. Continuous k-nearest neighbor queries in spatial network databases. In *STDBM*, pages 33–40, 2004.

[12] M. F. Mokbel, C.-Y. Chow, and W. G. Aref. The new casper: query processing for location services without compromising privacy. In *VLDB '06: Proceedings of the 32nd international conference on Very large data bases*, pages 763–774. VLDB Endowment, 2006.

[13] RTreeportal.org. North east dataset, November 2007. Web-address: http://www.rtreeportal.org/datasets/spatial/US/NE.zip.

[14] E. W. Weisstein, November 2007. http://mathworld.wolfram.com/Zipf-Distribution.html.

[15] J. Xu, J. Du, X. Tang, and H. Hu. Privacy-preserving location-based queries in mobile environments. Technical report, Hong Kong Baptist University, 2006.

[16] M. L. Yiu, C. S. Jensen, X. Huang, and H. Lu. Spacetwist: Managing the trade-offs among location privacy, query performance, and query accuracy in mobile services. In *24th IEEE International Conference on Data Engineering (ICDE)*, April 2008 (to appear).

Acronyms

DAG Directed Acyclic Graph

PDA Personal Digital Assistants

LBS Location Based Service

SMC Secure Multi-Party Computational Geometry

GPS Global Positioning System

KNN K-Nearest Neighbours

NN Nearest Neighbours

IDE Integrated Development Environment

APPENDIX A

Test Distributions

The section contains the remaining Zipf distributes used to test the solutions. These are depicted in Figure A.1, A.2 on the following page, A.3 on the next page, A.4 on page 73, A.5 on page 73, A.6 on page 74, A.7 on page 74, and A.8 on page 75.

It is chosen to depict the entire queryspace in all figures, as this enables them to be compared. This might have the result that some of the distributions seem empty, but this only means that the points are all close to $(0, 0)$.

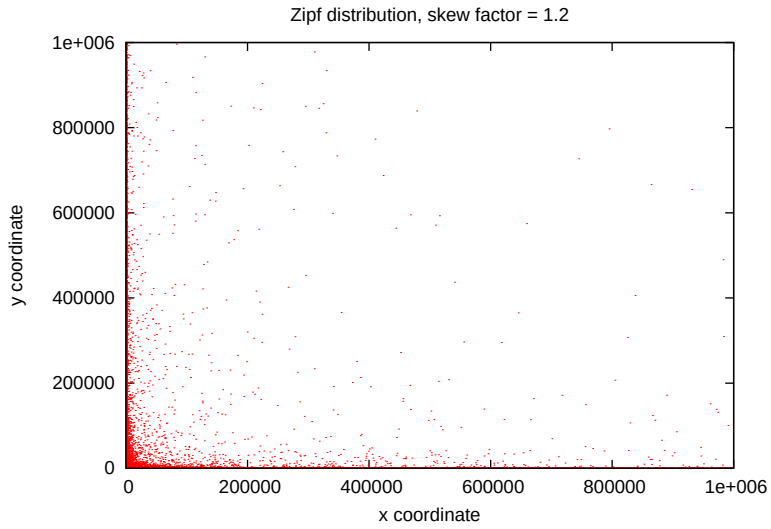


Figure A.1: Zipf distribtion with skew factor 1.2

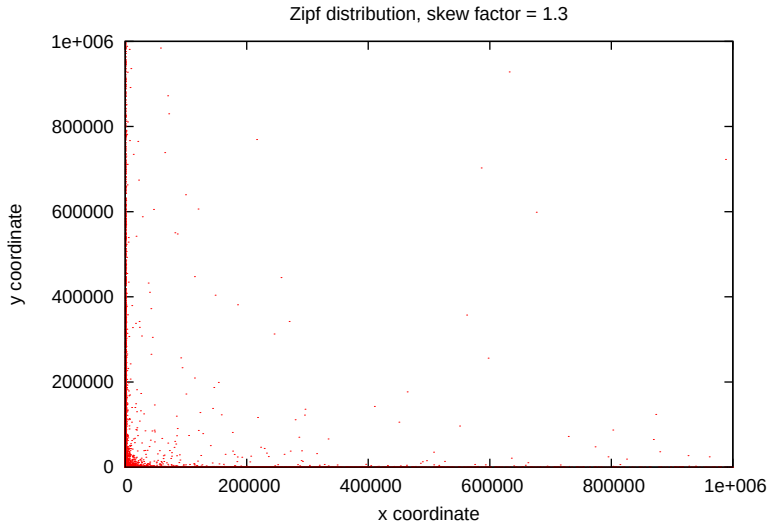


Figure A.2: Zipf distribtion with skew factor 1.3

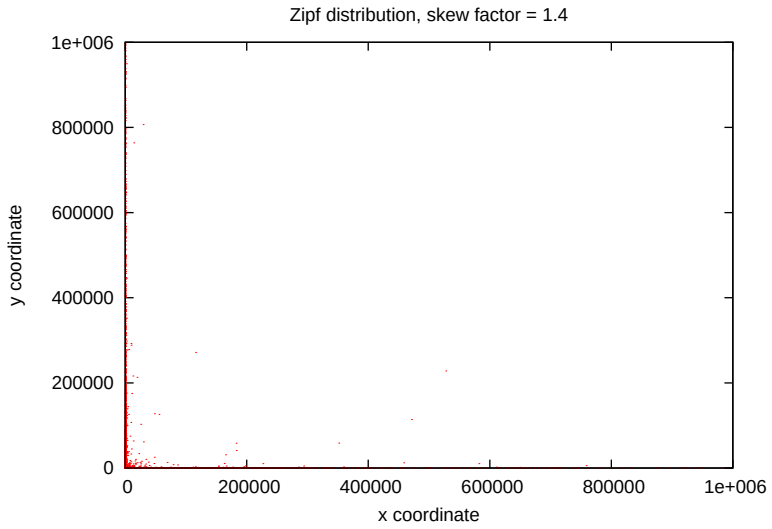


Figure A.3: Zipf distribtion with skew factor 1.4

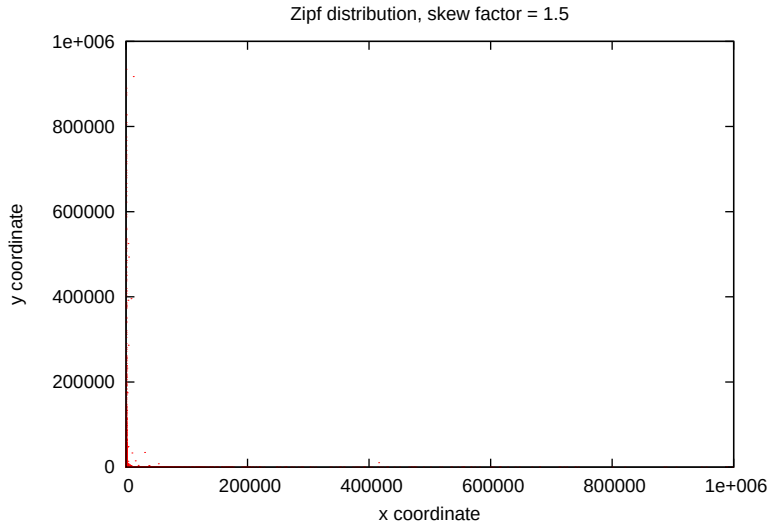


Figure A.4: Zipf distribtion with skew factor 1.5

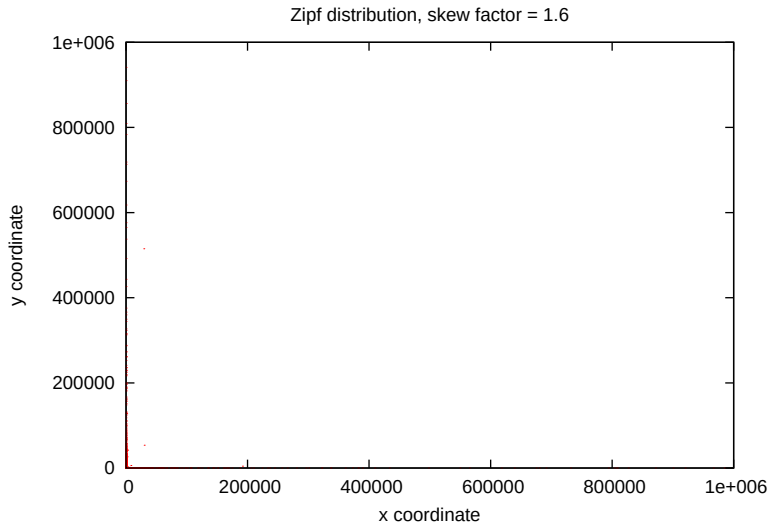


Figure A.5: Zipf distribtion with skew factor 1.6

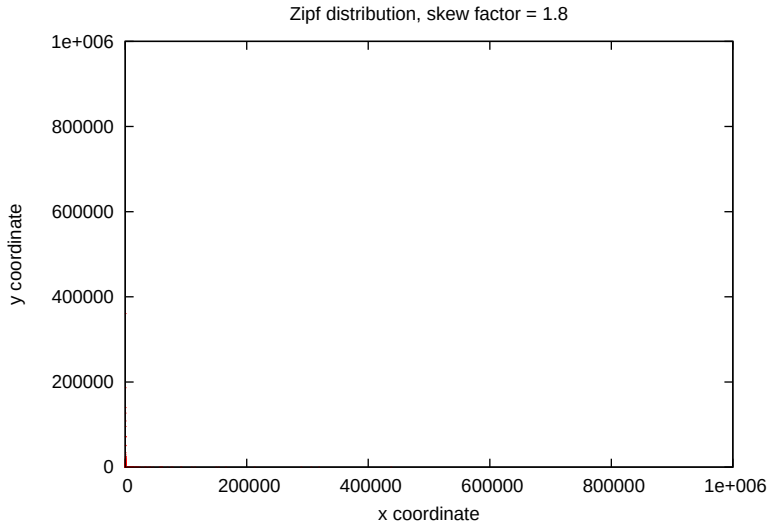


Figure A.6: Zipf distribtion with skew factor 1.8

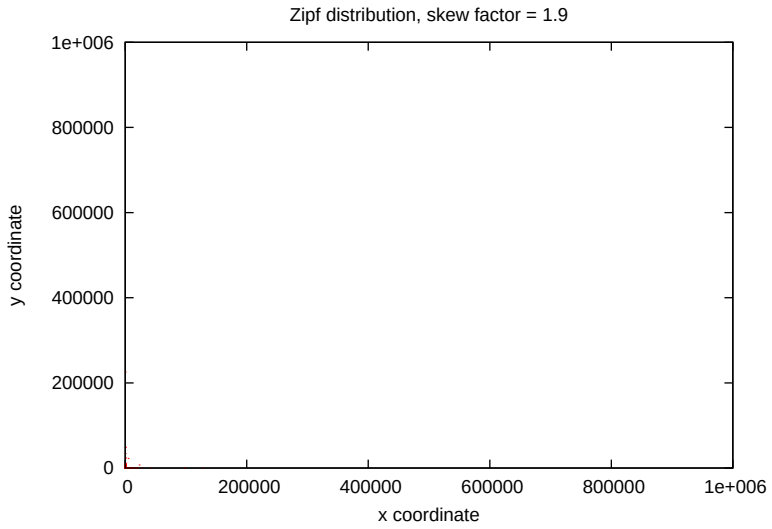


Figure A.7: Zipf distribtion with skew factor 1.9

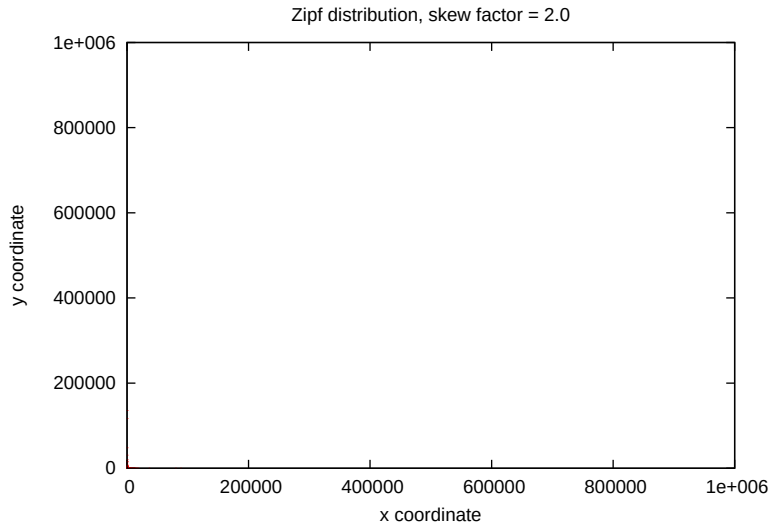


Figure A.8: Zipf distribtion with skew factor 2.0