M.Sc. Engg. Thesis

Obstructed Group Nearest Neighbor Queries in Spatial Databases

by

Nusrat Sultana

Submitted to

Department of Computer Science and Engineering
in partial fulfilment of the requirements for the degree of
Master of Science in Computer Science and Engineering

Department of Computer Science and Engineering
Bangladesh University of Engineering and Technology (BUET)
Dhaka 1000

July 2014
Dedicated to my loving parents

Author’s Contact

Nusrat Sultana
Software Engineer
Email: nusratst.cse@gmail.com
The thesis titled “Obstructed Group Nearest Neighbor Queries in Spatial Databases”, submitted by Nusrat Sultana, Roll No. 0411052002P, Session April 2011, to the Department of Computer Science and Engineering, Bangladesh University of Engineering and Technology, has been accepted as satisfactory in partial fulfillment of the requirements for the degree of Master of Science in Computer Science and Engineering and approved as to its style and contents. Examination held on July 1, 2014. The thesis is found satisfactory and has been accepted by the Board of Examiners on September 6, 2014.

Board of Examiners
1. Dr. Tanzima Hashem Chairman
   Assistant Professor (Supervisor)
   Department of CSE, BUET, Dhaka 1000.

2. Dr. Mahfuzul Islam Member
   Head and Professor (Ex-Officio)
   Department of CSE, BUET, Dhaka 1000.

3. Dr. Abu Sayed Md. Latiful Hoque Member
   Professor (Ex-Officio)
   Department of CSE, BUET, Dhaka 1000.

4. Dr. Md. Shohrab Hossain Member
   Assistant Professor
   Department of CSE, BUET, Dhaka 1000.

5. Dr. Mohammad Nurul Huda Member
   Professor (External)
   Department of Computer Science and Engineering
   United International University, Dhanmondi, Dhaka-1207.
Candidate’s Declaration

This is hereby declared that the work titled “Obstructed Group Nearest Neighbor Queries in Spatial Databases” is the outcome of research carried out by me under the supervision of Dr. Dr. Tanzima Hashem, in the Department of Computer Science and Engineering, Bangladesh University of Engineering and Technology, Dhaka 1000. It is also declared that this thesis or any part of it has not been submitted elsewhere for the award of any degree or diploma.

_________________________________

Nusrat Sultana
Candidate
Acknowledgment

I express my heart-felt gratitude to my supervisor, Dr. Tanzima Hashem for her constant supervision of this work. She helped me a lot in every aspect of this work and guided me with proper directions whenever I sought one. Her patient hearing of my ideas, critical analysis of my observations and detecting flaws (and amending thereby) in my thinking and writing have made this thesis a success. I would also want to thank the members of my thesis committee for their valuable suggestions. I thank Dr. Md. Shohrab Hossain, Dr. Abu Sayed Md. Latiful Hoque and specially the external member Dr. Mohammad Nurul Huda.

In this regard, I remain ever grateful to my beloved parents, who always exists as sources of inspiration behind every success of mine I have ever made.
Abstract

A group nearest neighbor (GNN) query is an important class of information and enquiry services. Researchers have focused on developing efficient algorithms to evaluate variant of GNN queries in recent years. In this thesis, we introduce *obstructed group nearest neighbor* (OGNN) queries that enable a group of pedestrians located at different places to meet at a data point such as a restaurant that minimizes their aggregate travel distance in presence of obstacles such as buildings and lakes. We propose the first comprehensive approach to process an OGNN query. The obstructed distance between two point locations is defined as the length of the shortest path that connects them without crossing any obstacles. The aggregate obstructed distance is measured in terms of the total or the maximum travel distance of the group members in an obstructed space. We develop two techniques, *Single Point Aggregate Obstructed Distance* (SPAOD) computation and *Multi Point Aggregate Obstructed Distance* (MPAOD), for computing aggregate obstructed distance between a data point and a set of query points, which is an essential part in evaluating GNNs in an obstructed space. SPAOD computation does not retrieve an obstacle multiple times but may retrieve additional obstacles, which are not required for computing the aggregate obstructed distance. On the other hand, MPAOD may retrieve the same obstacle multiple times but does not retrieve any obstacle, which is not required for computing the aggregate obstructed distance. We propose two methods: *Group Based Query Method* (GBQM) and *Centroid Based Query Method* (CBQM) to evaluate OGNN queries. The key idea of GBQM and CBQM is to incrementally retrieve data points until the actual obstructed GNN has been found. GBQM retrieves the Euclidean group nearest neighbors with respect to the locations of group members, whereas CBQM retrieves the Euclidean nearest neighbors with respect to the centroid of the locations of group members. We validate the efficacy and efficiency of our solutions with an extensive experiments using both real and synthetic datasets and present a comparative analysis among our proposed algorithms in terms of query processing overhead.
2.6.1 Rtree Indexing Structure .................................................. 16

3 Obstructed Distance Computation ........................................... 18
  3.1 Visibility Graph ............................................................. 18
    3.1.1 Incremental Visibility Graph Computation Strategy .......... 19
  3.2 Obstructed Distance between two Points .............................. 20
  3.3 Aggregate Obstructed Distance from Multiple Points to a Single Point .... 23
    3.3.1 Single Point Based Aggregate Obstructed Distance (SPAOD) Computation .. 23
      3.3.1.1 Correctness of Algorithm CompAggObsDist (SPAOD) .......... 30
      3.3.1.2 Time Complexity of Algorithm CompAggObsDist (SPAOD) .... 31
    3.3.2 Multi Point Based Aggregate Obstructed Distance (MPAOD) Computation .. 31
      3.3.2.1 Correctness of Algorithm CompAggObsDist (MPAOD) .......... 35
      3.3.2.2 Time Complexity of Algorithm CompAggObsDist (MPAOD) ... 36

4 Obstructed Group Nearest Neighbor Queries ................................ 37
  4.1 Centroid Based Query Method (CBQM) for SUM ....................... 37
    4.1.1 Correctness of Algorithm CBQM_SUM ................................ 42
    4.1.2 Time Complexity of Algorithm CBQM_SUM .......................... 43
  4.2 Centroid Based Query Method (CBQM) for MAX ....................... 43
    4.2.1 Correctness of Algorithm CBQM_MAX ................................ 47
    4.2.2 Time Complexity of Algorithm CBQM_MAX .......................... 47
  4.3 Group Based Query Method (GBQM) ...................................... 48
    4.3.1 Correctness of Algorithm GBQM ...................................... 52
    4.3.2 Time Complexity of Algorithm GBQM ............................... 52

5 Experiments ............................................................................. 54
  5.1 Comparison of OGNN and Obstructed Distance Computation Algorithms .... 55
    5.1.1 Effect of Group Size ..................................................... 56
      5.1.1.1 Aggregate function sum ............................................ 56
      5.1.1.2 Aggregate function max ............................................ 58
    5.1.2 Effect of k ............................................................... 59
      5.1.2.1 Aggregate function sum ............................................ 59
5.1.2.2 Aggregate function MAX ........................................ 61
5.1.3 Effect of Query Area .............................................. 62
  5.1.3.1 Aggregate function SUM ...................................... 63
  5.1.3.2 Aggregate function MAX ...................................... 65
5.1.4 Effect of Cardinality Ratio ...................................... 66
  5.1.4.1 Aggregate function SUM ...................................... 66
  5.1.4.2 Aggregate function MAX ...................................... 67
5.2 Comparative Analysis ................................................ 69

6 Conclusion ................................................................. 70

References ................................................................. 72
List of Figures

1.1 An OGNN query example .................................................. 2
1.2 Euclidean and obstructed distance between two points ................. 4
1.3 $f = \text{sum}$ ................................................................. 5
1.4 $f = \text{max}$ ................................................................. 5

2.1 Point position of R-Tree ..................................................... 15
2.2 Tree structure ................................................................. 15
2.3 An R-Tree Example ......................................................... 15
2.4 A best-first search example in Rtree ...................................... 16

3.1 Visibility graph and obstructed distance ................................ 19
3.2 Obstructed distance $\text{dist}_O(p,q)$ calculation between $p$ and $q$ .......... 21
3.3 Obstructed distance $\text{dist}_O(p,q)$ calculation between $p$ and $q$ .......... 22
3.4 Example of SPAOD .......................................................... 29
3.5 Example of MPAOD ......................................................... 34

4.1 Example of aggregate function $\text{Sum}$ for CBQM ......................... 41
4.2 Example of aggregate function $\text{max}$ for CBQM ......................... 46
4.3 Example of GBQM .......................................................... 51

5.1 Effect of group size on $\text{sum}$ (Germany dataset) ....................... 56
5.2 Effect of group size on $\text{sum}$ ($U$ dataset) ............................... 57
5.3 Effect of group size on $\text{sum}$ ($Z$ dataset) ............................... 57
5.4 Effect of group size on $\text{max}$ (Germany dataset) ....................... 58
5.5 Effect of group size on $\text{max}$ ($U$ dataset) ............................... 59
5.6 Effect of group size on \( \text{MAX} \) (\( Z \) dataset) .................................................. 59
5.7 Effect of \( k \) on \( \text{SUM} \) (Germany dataset) .................................................. 60
5.8 Effect of \( k \) on \( \text{SUM} \) (\( U \) dataset) .................................................. 61
5.9 Effect of \( k \) on \( \text{SUM} \) (\( Z \) dataset) .................................................. 61
5.10 Effect of \( k \) on \( \text{MAX} \) (Germany dataset) .................................................. 62
5.11 Effect of \( k \) on \( \text{MAX} \) (\( U \) dataset) .................................................. 62
5.12 Effect of \( k \) on \( \text{MAX} \) (\( Z \) dataset) .................................................. 63
5.13 Effect of query area on \( \text{SUM} \) (Germany dataset) .................................................. 63
5.14 Effect of query area on \( \text{SUM} \) (\( U \) dataset) .................................................. 64
5.15 Effect of query area on \( \text{SUM} \) (\( Z \) dataset) .................................................. 64
5.16 Effect of query area on \( \text{MAX} \) (Germany dataset) .................................................. 65
5.17 Effect of query area on \( \text{MAX} \) (\( U \) dataset) .................................................. 65
5.18 Effect of query area on \( \text{MAX} \) (\( Z \) dataset) .................................................. 66
5.19 Effect of cardinality on \( \text{SUM} \) (\( U \) dataset) .................................................. 67
5.20 Effect of cardinality ratio on \( \text{SUM} \) (\( Z \) dataset) .................................................. 67
5.21 Effect of cardinality on \( \text{MAX} \) (\( U \) dataset) .................................................. 68
5.22 Effect of cardinality ratio on \( \text{MAX} \) (\( Z \) dataset) .................................................. 68
List of Tables

2.1 BFS in Rtree ................................. 17
3.1 Notation used and their meanings ................................. 24
3.2 SPAOD Example .................................. 29
4.1 Notation used and their meanings ................................. 38
4.2 $CBQM_{SUM}$ Example ................................. 41
4.3 $CBQM_{MAX}$ Example ................................. 45
4.4 GBQM for $SUM$ Example ................................. 51
5.1 Experimental Setup .................................. 55
# List of Algorithms

1. `CompAggObsDist(Q, p, RT_O, LVG)` ............................................. 26
2. `CompAggObsDist(Q, p, RT_O, LVG)` ............................................. 32
3. `CBQM_SUM (Q, p, k, RT_O, RT_P)` ............................................. 40
4. `CBQM_MAX (Q, p, k, RT_O, RT_P)` ............................................. 45
5. `GBQM (Q, p, k, RT_O, RT_P)` .................................................. 50
Chapter 1

Introduction

The proliferation of location aware mobile devices such as GPS enabled smart phones are attracting an increasing number of users to access information and enquiry services. An important class of information and enquiry service is a group nearest neighbor (GNN) query that enables a group of users to meet a data point such as a restaurant, a shopping mall or a movie theater that minimizes the aggregate travel distance. For example, with a GNN query, a group of friends located at different places of a city center can know the location of a restaurant that minimizes their total travel distance. The aggregate distance could also be measured in terms of the maximum travel distance of group members, which enables the group to meet at a data point within the shortest possible time.

Existing research have developed algorithms for processing GNN queries in the Euclidean space [1–6] and road networks [7, 8]. However, these algorithms have not considered pedestrians and their travel path which may include blocking obstacles. In a park, there can be obstacles like trees, lakes or playgrounds; no pedestrian can walk through a lake without crossing it. Similarly, in a city center there might be buildings or roads for vehicles which obstruct a pedestrian’s walking path. However, GNN queries in the Euclidean space or road networks ignore obstacles and measure the distance between two points using Euclidean or road distance, respectively. In this thesis, we introduce Obstructed Group Nearest Neighbor (OGNN) queries and propose the first comprehensive approach to find group nearest neighbors in presence of obstacles.
CHAPTER 1. INTRODUCTION

1.1 Research Problem

There exist algorithms [1–8] to evaluate GNN queries in spatial databases and road networks, which are not applicable for pedestrians. In reality, a pedestrian’s walking path is an obstructed space, where the pedestrian’s movement is restricted by obstacles. The nearest restaurant for a group of friends located at different location of a city center in terms of Euclidean distance might not be the nearest according to the obstructed aggregate travel distance for an OGNN query due to blocking buildings.

Figure 1.1: An OGNN query example

Consider a scenario in Figure 1.1a. Here we have to find the obstructed group nearest neighbor of 3 points \( q_1, q_2 \) and \( q_3 \) and the obstacles \( o_1, o_2, o_3, o_4 \) and \( o_5 \) are shown as shaded area. Though the Euclidean GNN query returns the point \( p_1 \) as the answer (Figure 1.1b), if we consider obstacles then we will find that \( p_2 \) is the actual obstructed group nearest neighbor (Figure 1.1c) as \( q_2 \) and \( q_3 \) has to travel a lot to reach the point \( p_1 \) as there is an obstacle in between.

Query processing in a spatial network has similarities with processing queries in an obstructed space, since in both cases movement is restricted (to the underlying network or by the obstacles). Edges in a spatial network denote the permitted paths for movement, whereas obstacles in an obstructed space represent areas where movement is prohibited. This fact differentiates query processing for the two cases.

Though there are algorithms [9–13] for evaluating other types of queries such as nearest neighbor
queries, moving nearest neighbor queries, reverse nearest neighbor queries in the obstructed space, group nearest neighbor queries in an obstructed space has not yet been studied in the literature. We develop the first algorithms to efficiently compute aggregate obstructed distance and evaluate OGNN queries.

In the following sections we formally discuss obstructed space, aggregate obstructed distance and obstructed group nearest neighbor queries and discuss about basic ideas that we are going to use throughout the thesis.

1.1.1 Obstructed Space

Obstructed space consists of obstacles like buildings, lakes, roads for vehicles, trees etc. including the point of interests like restaurants, movie theaters etc. and the query points i.e., the group of people introducing the query. Obstructed space is different from Euclidean space and network space. Compared to Euclidean space which does not consider obstacles that block the direct path between two points, the obstructed distance in obstructed space counts obstacle and finds the shortest obstacle avoiding path between those two points. Besides, network space considers an underlying fixed network structure and the objects are bound to move across the structure which is also different from obstructed space.

1.1.2 Aggregate Obstructed Distance

Obstacle path problem is a popular topic in Robotics, Computational Geometry, GIS, Game Planning etc. Obstructed path is the shortest path among two points \( p \) and \( q \) in the presence of a set of obstacles \( O \). The special property to be mentioned here is, the obstacles are non-overlapping 2D polygons and the shortest path among the two points do not cross the interior of any obstacle. The shortest path distance connecting two objects without crossing any obstacle in an obstructed space (space consisting of obstacles/polygon) is called the obstructed distance \( \text{dist}_O(p, q) \), between a data point \( p \) and a query point \( q \).

In Figure 1.2 the obstructed distance \( \text{dist}_O(p, q) \), between the data point \( p \) and the query point \( q \) is shown. The shaded polygons are obstacles in this obstructed space. The Euclidean distance
CHAPTER 1. INTRODUCTION

Figure 1.2: Euclidean and obstructed distance between two points

dist_E(p,q), between the data point p and the query point q is also shown by a dashed line.

The aggregate obstructed distance between a data point p and a set of query points Q is defined as dist_{AO}(p,Q). Aggregate obstructed distance is computed with respect to a monotonically increasing function f. We can say,

$$dist_{AO}(p,Q) = f(dist_O(p,q_1), dist_O(p,q_2), ......., dist_O(p,q_n))$$

where dist_O(p,q_i) is the obstructed distance between data point p and query point q_i.

The function f can be sum (minimizing the total travel distance) or max (minimizing the maximum travel distance). We store all our obstacles and data points in a spatial database using an indexing method. We use an Rtree based indexing method for our algorithms, though any kind of indexing method is applicable for our algorithms.

1.1.3 k Obstructed Group Nearest Neighbor (kOGNN) Queries

Given a set of locations Q = \{q_1, q_2, \ldots, q_n\} for n users, a set of data points P = \{p_1, p_2, \ldots, p_m\}, and a set of obstacles O = \{o_1, o_2, \ldots, o_h\}, a k obstructed group nearest neighbor query (OGNN) returns k points from P, which have the k smallest values for an aggregate function f. The aggregate function can be max or sum. If dist_O(p,q_i) is the obstructed distance between p and i^{th} query point q_i, then,
1. **An aggregate function for sum**: An aggregate function \( \text{SUM} \) minimizes the *total of obstructed distances* of the users.

   if \( f(p, Q) = \text{SUM} \), then,
   \[
   \text{dist}_{AO}(p, Q) = \sum_{i=1}^{n} \text{dist}_{O}(p, q_i),
   \]
   where \( q_i \) is the \( i^{th} \) point

2. **An aggregate function for max**: An aggregate function \( \text{MAX} \) minimizes the *maximum obstructed distance*.

   if \( f(p, Q) = \text{MAX} \), then,
   \[
   \text{dist}_{AO}(p, Q) = \max_{1 \leq i \leq n} \text{dist}_{O}(p, q_i),
   \]
   where \( q_i \) is the \( i^{th} \) point

Figure 1.3 shows an example, for aggregate function \( \text{SUM} \) where, the OGNN query returns point \( p_1 \), that minimizes the total obstructed distance of the users. \( p_1 \)'s total obstructed distance is less than the total obstructed distances of all the other query points. The aggregate obstructed distance for function \( \text{SUM} = \sum_{i=1}^{3} \text{dist}_{O}(p_1, q_i) = 35 \leq \text{dist}_{AO}(p, Q) \forall p \in P \), where \( P \) is a set of data points in the obstructed space.

Figure 1.4 shows an example for aggregate function \( \text{MAX} \) where, the OGNN query returns point \( p_2 \), that minimizes the maximum obstructed distance. \( p_2 \)'s maximum obstructed distance is less than the maximum obstructed distances of all the other query points. The aggregate obstructed distance for function \( \text{MAX} = \max_{i=1}^{3} \text{dist}_{O}(p_2, q_i) = 8 \leq \text{dist}_{AO}(p, Q) \forall p \in P \),
where $P$ is a set of data points in the obstructed space.

### 1.2 Solution Overview

As there can be a large number of obstacles in a pedestrian’s walking path in real world, finding the actual OGNN from a large dataset in real time is a major challenge. The less the number of data points that an OGNN algorithm requires to consider for identifying the actual obstructed OGNN, the more efficient the algorithm is as for each retrieved data point, an OGNN algorithm computes the aggregate obstructed distance of the data point from the locations of pedestrians.

We develop variant of pruning techniques to make the candidate answer set smaller and thereby reduce the query processing overhead. In addition, we present efficient algorithms to compute obstructed aggregate distance, which is a key component to efficiently evaluate the OGNN. Considering all obstacles for computing distance every time is not a feasible solution. The base idea of our aggregate obstructed distance computation technique is to consider only those obstacles, which are required for distance computation and to avoid retrieving the same obstacles multiple times. In Section 1.2.1 and Section 1.2.2, we give a short overview of our proposed solutions for aggregate obstructed distance computation and OGNN query evaluation.

#### 1.2.1 Aggregate Obstructed Distance Computation

Algorithms [9, 10, 12] already exist for finding the obstructed distance between two points, but computing the aggregate obstructed distance from a single data point ($p$) to multiple query points ($Q$) is not discussed in the literatures. A straightforward approach can be to use existing algorithms to find our aggregate obstructed distance by considering the data point ($p$) and one query point ($q \in Q$) at a time from the set of query points ($Q$), then applying aggregate operation in the results. This technique involves high IO access. Same obstacle may be considered multiple times for different query points in the group ($Q$), which is not desired.

We give two algorithms for computing aggregate obstructed distance between a data point $p$ and multiple query points $Q$ (i.e., the locations of the group members). One algorithm is based on the centroid of the query points which we call Single Point Aggregate Obstructed Distance (SPAOD) computation
and another algorithm is based on the data points which we call \textit{Multi Point Aggregate Obstructed Distance} (MPAOD) computation. The key difference between the two algorithms is in the way we retrieve the obstacles from the database.

For single point based approach, we incrementally retrieve the obstacles from the centroid of the query points and no obstacle is retrieved more than once. After retrieving obstacles having distance equal to a threshold with respect to the centroid of the query points we compute the aggregate obstructed distance between a data point \( p \) and a set of query points \( Q \). When computing aggregate obstructed distance for the subsequent data points we just increase the threshold and retrieve more obstacles for that increased threshold only. This algorithm also reuse the already computed obstructed shortest path distance when computing obstructed shortest path distance for subsequent data points.

Another kind of aggregate obstructed distance computation algorithm is multiple point based approach. The intuition behind this algorithm is, when computing aggregate obstructed distance between a data point \( p \) and a set of query points \( Q \), we retrieve obstacles having distance equal to a threshold with respect to the data point \( p \). For the subsequent data points we retrieve a new set of obstacles. The new set of obstacles may include already retrieved obstacles, which invokes same obstacle retrieval multiple times. This is the main difference between the two algorithms SPAOD and MPAOD. Other than that, MPAOD checks the intersection of the retrieved obstacles with the latest shortest path which also make the two algorithms unique.

\subsection{1.2.2 \textit{kOGNN} Queries}

In this thesis, we propose two algorithms for finding \textit{OGNN} of a set of query points \( Q \). Each algorithms has its own way to determine the \textit{OGNN} depending on the aggregate function \texttt{sum} or \texttt{max}. The first approach \textit{GBQM} (Group Based Query Method), is based on the Euclidean GNN of the set of query points \( Q \). We incrementally retrieve Euclidean GNNs of the set of query points \( Q \) and compute its aggregate obstructed distance, until the best \( k \) obstructed group nearest neighbor have been found.

For the second approach \textit{CBQM} (Centroid Based Query Method) our intuition is to find the Euclidean nearest neighbor with respect to the centroid of the query points. In this technique, we incrementally retrieve Euclidean nearest neighbors with respect to the centroid of the query points until the best \( k \)
OGNN have been found.

1.3 Contributions

We summarize the contributions of this thesis as follows:

- We introduce $k$ obstructed group nearest neighbor queries ($k$OGNN) in obstructed space.
- We propose efficient algorithms to evaluate $k$ obstructed group nearest neighbors ($k$OGNN) for a set of query points $Q$. One Algorithm $CBQM$ (Centroid Based Query Method) is based on the Euclidean nearest neighbor with respect to the centroid of the query points and another Algorithm $GBQM$ (Group Based Query Method), is based on the Euclidean GNN of the set of query points.
- We develop two techniques to compute aggregate obstructed distance between a data point ($p$) and a set of query points ($Q$). In one Algorithm Single Point Aggregate Obstructed Distance (SPAOD) computation we incrementally retrieve the obstacles from the centroid of the query points and no obstacle is retrieved more than once. Whereas, another kind of aggregate obstructed distance computation algorithm is Multiple Point Aggregate Obstructed Distance (MPAOD) computation. The intuition behind this algorithm is, when computing aggregate obstructed distance between a data point $p$ and a set of query points $Q$, we retrieve obstacles having distance equal to a threshold with respect to the data point $p$.
- We conduct extensive experimental analysis using both real and synthetic datasets and show that our proposed algorithms can determine OGNN with reduced time and space overhead.

1.4 Outline

The remaining part of the thesis is organized as follows:

In Chapter 2, we study existing research related to different of queries queries in spatial database along with some spatial indexing technique.

In Chapter 3, we explain the algorithms for computing aggregate obstructed distance.

In Chapter 4, we describe our two algorithms to evaluate $k$ obstructed group nearest neighbor
(kOGNN) queries.

In Chapter 5, we implement our algorithms and show some experimental results using both real and synthetic datasets.

In Chapter 6, summary and outcome of the thesis with possible future directions are described.
Chapter 2

Related Work

In this chapter, we study existing research related to GNN queries and obstructed nearest neighbor queries in spatial databases. In Sections 2.1 and 2.2 we analyze some existing approaches related to GNN queries in the Euclidean space and road networks, respectively. In Sections 2.3 and 2.4 we study existing obstacle path problems in computational Geometry and in data clustering literature, respectively. Existing queries in obstructed space is studied in Section 2.5. Section 2.6 presents indexing techniques for storing and accessing data from the database.

2.1 Group Nearest Neighbor Queries in Euclidean Space

Papadias et al. [1] proposed techniques for finding Euclidean GNN. They proposed three algorithms Multiple Query Method (MQM), Single Point Method (SPM) and Minimum Bounding Method (MBM). MQM is a threshold based algorithm, it computes the GNN incrementally until the threshold crosses the best distance found so far. MQM traverses the same node through different queries but SPM finds GNN by a single traversal. SPM at first finds the centroid of query points which has minimum aggregate distance from all of them. It applies a depth-first or best-first search from centroid and uses some heuristic based on triangular inequality to prune intermediate nodes and determines the real GNN. Minimum Bounding Method (MBM) uses the same strategy as SPM but instead of centroid it finds a minimum bounding rectangle $M$ to prune the search space in a single query.

In [5], Papadias et al. proposed algorithm for Euclidean GNN which supports 3 types of aggregate function $\text{sum}$, $\text{max}$ and $\text{min}$. $\text{sum}$ Euclidean GNN query reports the location that minimizes the sum
of distances to all point of interest. \textsc{max} outputs the points that minimizes the maximum distance or leads to earliest possible pickup time. However, \textsc{min} outputs the points that minimizes the minimum distance. They also discussed the problem in terms of weighted query and incremental reporting of results.

In [14], Sato and Narita proposed algorithms for computing approximate k-GNN over remote spatial databases which also supports \textsc{sum} and \textsc{max} aggregate functions. This paper relies on a \textit{Representative Query Point} (RQP) to be used as a key of a k-Nearest Neighbor (k-NN) query for searching spatial data.

Two \textit{Ellipse} based pruning method was proposed in [15] for GNN queries which take into account the distribution of query points. The methods employ an ellipse to approximate the extent of multiple query points, and then derive a distance or minimum bounding rectangle (MBR) using that ellipse to prune intermediate nodes. They considered an ellipse with two foci, which is the trajectory of all the query points in the set and all points outside the ellipse are farther than the points inside the ellipse.

### 2.2 Group Nearest Neighbor Queries in Road Networks

Group nearest neighbor queries are also studied in road networks. In [7], Yiu \textit{et al.} proposed algorithm which utilizes Euclidean distance bounds, network materialization and spatial access methods when finding GNN in spatial network. This paper presents three approaches for GNN queries in road networks. First algorithm is applicable when the Euclidean distance is the lower bound of the network distance between two network nodes. The rest two algorithms is based on the idea of multiple sources. In [8], Sun \textit{et al.} proposed algorithms for finding \textit{merged group nearest neighbor (MANN)} where, a merged set consists of a given query point set and others points needed from a candidate set in road network. Since, our obstructed group nearest neighbor query does not have any underlying fixed network structure so we can not use road network based GNN queries in our algorithm.
2.3 Obstacle Path Problems in Computational Geometry

Path problems in obstructed space are studied in Computational Geometry [16] where the problem consists of main memory and shortest path problems considering obstacles. Most solutions are based on visibility graph and visibility polygon [17–22]. Lozano et al. [23] proved that the shortest path between two source and destination point lies in the visibility graph and can be computed by any conventional shortest path algorithms [24, 25].

The literatures in Computational Geometry rely on preprocessing and does not prune any obstacle. Since, spatial database applications may require update of spatial data, preprocessing will not give exact result set. On the other hand, considering all the obstacles for each query point is costly. For very large databases, real time query processing is almost impossible for the expensive query processing overhead.

2.4 Obstacle Path Problems in Data Clustering Literature

The data clustering literature uses precomputed and materialized visibility graph [26, 27]. A large number of clustering algorithms exist in obstructed space [28, 28–31]. Clustering divides a set a of points into smaller homogeneous group/clusters. Due to large spatial datasets pre computation of visibility graph is not suitable in our problem. We construct the visibility graph incrementally, which is described in Chapter 3.

2.5 Nearest Neighbor Queries in Obstructed Space

An obstructed nearest neighbor query was first extensively studied by [9], they propose four efficient algorithms for range search, obstructed nearest neighbor, e-distance join and closest pair query in obstructed space. The range search returns those data points from a data point set which are within a predefined obstructed distance in presence of obstacle. An obstructed nearest neighbor search returns the k data points which have smallest obstructed distances from a query point q. The closest pair
query returns the closest data points between two data points set, which have smallest obstructed distance.

In [10], Xia et al. also proposed efficient algorithms for finding obstructed nearest neighbor queries in an incremental way and filters out a large number of obstacles. The obstructed distance is computed incrementally using a visibility graph with only the core obstacles. They also propose efficient algorithms for finding the candidate data points pruning out a lot of unrelated data points.

In [11], Gu et al. construct a grid-partition index combined with the obstructed voronoi diagram. They used a precomputed voronoi diagram with obstacles and introduced a new concept obstructed bisector. They also claim that the nearest neighbor searching in this type of grid-partition based indexing is independent of of the query object \( q \). This paper presents three types of pruning heuristics pruning by no obstacles, pruning by cell border min and pruning by cell border max when computing the obstructed distance. However, the key difference between grid partition index and Rtree based index is, Rtree based index can efficiently handle dynamically updated data point whereas grid partition can only handle static objects as the voronoi diagram is precomputed.

Continuous obstructed nearest neighbor queries are discussed in [32, 33], which finds nearest neighbors of all the points of a specified query line, assuming the query line does not intersect any obstacle. This type of query is named as Continuous Obstructed Nearest Neighbour (CONN) query. They perform only a single query over the query line segment and process the relevant data points and obstacles via the concept of control points and quadratic-based split point computation approach. They propose two different approach to continuous obstructed \( k \) nearest neighbor and trajectory obstructed \( k \) nearest neighbor (TO\( k \)NN) to compute the kNNs for each point along with an arbitrary trajectory.

Another similar type of query is moving \( k \) nearest neighbor query. In [13], Li and Gu discussed this type query, which does not have any specified trajectory of the query object. In [34], moving k-NN algorithms are discussed which are based on a safe-region concept called the V*-Diagram. Since, our proposed obstructed group nearest neighbor queries consider only static objects it is different from moving \( k \) nearest neighbor queries.
In obstructed space, another class of queries are visible \( k \) nearest neighbor queries, which considers \( k \) nearest data point which are not blocked by any obstacles. In [35] and [36], continuous visible \( k \) nearest neighbor queries are discussed and algorithms are proposed for CVNN, which are based on Rtree based indexing.

In [37] and [38] Gao et al. proposed reverse visible \( k \) nearest neighbor queries, which efficiently reduces the preprocessing time and prune the search space through half-plane property and visibility check.

Recently, all visible \( k \) nearest neighbor queries are proposed in [39], which retrieves visible \( k \) nearest neighbors for each point in a query set \( Q \).

Xu et al. [2] proposed a new type of query, Group Visible Nearest Neighbor Query (GVNN), which prunes both data set and obstacle set by defining the invisible region of MBR of query set. This is the first research which relates obstacles with GNN queries. The paper presents two algorithm Multiple Traversing Obstacles (MTO) algorithm and Traversing Obstacles Once (TOO) algorithm to efficiently solve GVNN problem. GVNN queries has some difference with our OGNN queries. GVNN queries prune those data points which cannot be seen by all the query points. Consider a set of query points \( Q \) for which we are going to answer GVNN query, if there is a data point \( p_1 \) which is blocked by an obstacle \( o_1 \), the GVNN query rejects \( p_1 \) though it is visible from all the other query points in \( Q \).

On the other hand, our OGNN query considers aggregate obstructed distance between the candidate data point \( p_1 \) and query point set \( Q \) and does not prune invisible data points.

Another similar type of query in an obstructed space is obstructed reverse nearest neighbor query, given a dataset \( P \) and a query point \( q \), an ORNN query returns all the points in \( P \), that have \( q \) as their nearest neighbor. In [12], Gao and Yang proposed the first approach for answering ORNN which follows a filter-refinement framework and requires no preprocessing and enables effective pruning heuristics. They also introduced a novel boundary region concept. This paper also presents an idea to compute the obstructed distance between two points \( p \) and \( q \) incrementally using previous shortest path calculation.

Maximum visibility query is a special type of query which finds some locations from a set of locations that maximize the visibility of a query object. In [40], Masud et al. proposed three approaches that incrementally considers the relevant obstacles to determine the visibility of the target object.
2.6 Indexing Techniques for Space Partitioning

There are a lot of space partitioning data structures such as, Kd-trees, Quadtrees, Octrees, Rtrees [41–45] etc. Most of the studies in the spatial database involves Rtrees and its variants [46–48]. An Rtree is a depth-balanced tree where each node corresponds to a minimum bounding rectangle. Each leaf node consists of an array of leaf entries and each non-leaf node contains an array of node entries. An example of Rtree is showed in Figure 2.3 which indexes a set of query points.

R-tree has some benefits over other space partitioning data structures. For example, kd-trees [41, 42] partition the whole space into regions whereas R-trees only partition the subset of space containing the points of interest. Kd-trees represent a disjoint partition (points belong to only one region) whereas the regions in an R-tree may overlap. Rtrees are disk-oriented whereas kd-trees are memory oriented. The most important benefit of R-tree is that, it can store rectangles and polygons but kd-trees can only store point vectors. So, R-tress are much more suitable in an obstructed space than kd-tress.

On the other hand, quadtrees [43, 44] are most often used to partition a two dimensional space by recursively subdividing it into four quadrants or regions whereas R-tree is a balanced search tree and it organizes the data in pages, and is designed for storage on disk. Quadtrees work only on two dimensional space but R-trees also work in multi dimensional space. Rtrees requires less storage than quadtree.

Octrees [45] are the three-dimensional analog of quadtrees. For nearest neighbor queries, R-tree indexes are faster that octress and quadtrees.
CHAPTER 2. RELATED WORK

16

a) Data points in space

R$_4$

b) Corresponding Rtree

Figure 2.4: A best-first search example in Rtree

For the aforementioned benefits we have used Rtrees as the indexing structure for both our obstacles and data points, though the algorithms is applicable for any indexing technique.

2.6.1 Rtree Indexing Structure

R-tree performs the traversing of the tree in a branch-and-bound manner. For nearest neighbor search algorithms, two types of technique exists, the depth-first and the best-first.

First of all, the depth-first algorithm starts traversing from the root of the tree and visits the node whose minimum distance (MinDist) from the query object is smallest. In this way, when it reaches a leaf node, it finds the candidate nearest neighbor. After that, it traverse back and only visits the nodes whose minimum distance is smaller than the distance of the candidate nearest neighbor.

The best-first nearest neighbor search is known as the incremental nearest neighbor search. It maintains a priority queue for the visited nodes. The queue stores entries in an increasing order of their minimum distance from the query object. The entry with the smallest minimum distance is visited first. The best-first nearest neighbor search gives the advantage of retrieving the successive nearest neighbors in an incremental manner without re-computing the query from the scratch. We have used the best-first nearest neighbor search approach in this thesis to retrieve entries from RTree.

Figure 2.4 shows an example of best first search (BFS) in Rtree. Figure 2.4a shows the data points
Table 2.1: BFS in Rtree

<table>
<thead>
<tr>
<th>Visit Root</th>
<th>$R_1$</th>
<th>$R_2$</th>
<th>$R_3$</th>
<th>$R_4$</th>
<th>$R_5$</th>
<th>$R_6$</th>
<th>$R_7$</th>
<th>$R_9$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Follow $R_1$</td>
<td>$R_2$</td>
<td>$R_4$</td>
<td>$R_5$</td>
<td>$R_3$</td>
<td>$R_6$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Follow $R_2$</td>
<td>$R_8$</td>
<td>$R_4$</td>
<td>$R_5$</td>
<td>$R_3$</td>
<td>$R_6$</td>
<td>$R_7$</td>
<td>$R_9$</td>
<td></td>
</tr>
<tr>
<td>Follow $R_3$</td>
<td>$R_4$</td>
<td>$R_5$</td>
<td>$R_3$</td>
<td>$R_6$</td>
<td>$R_7$</td>
<td>$R_9$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Report $h$</td>
<td>$R_4$</td>
<td>$R_5$</td>
<td>$R_3$</td>
<td>$R_6$</td>
<td>$i$</td>
<td>$R_7$</td>
<td>$g$</td>
<td></td>
</tr>
</tbody>
</table>

$a, b, c, d, e, f, g, h, i$ in this space and Figure 2.4b shows the corresponding Rtree. The steps to retrieve the nearest neighbor of query point $q$ is shown in Table 2.1, where the first column shows the action in Rtree and the rest of the columns show the entries in the priority queue. The first nearest neighbor $h$ of query point $q$ is reported at the 5th iteration.

The aforementioned literatures find the nearest point for a given query point $q$ in an obstructed space. But, none of them considered a group nearest point for a set of query points $Q$. In this thesis we propose the novel approach to find the OGNN which includes relevant obstacles and data points in a branch and bound manner pruning out unrelated obstacles and data points.
Chapter 3

Obstructed Distance Computation

In this chapter, we propose our algorithms for aggregate obstructed distance computation. In Section 3.1, we describe Visibility Graph, which is used throughout our algorithms. In Section 3.2, we describe existing obstructed distance computation strategies between two points. In Section 3.3 we propose our algorithms for computing aggregate obstructed distance.

3.1 Visibility Graph

We use visibility graph to find the visibility among different points in presence of obstacles in an obstructed space. Visibility graph consists of all the data points and obstacle vertices along with the query points. Each node in the visibility graph represents a point location. There is an edge between two nodes if and only if the two nodes are mutually visible i.e. there is no obstacle edge obstructing the visibility between them. If the line segment connecting two locations does not pass through any obstacle, then we conclude that the two points are visible to each other and we draw an edge between them in the visibility graph [49].

An example of visibility graph is shown in Figure 3.1, where there are two obstacles (o₁, o₂) and two points (p, q). All the visible vertices are joined by an edge and the shortest obstructed path between p and q are showed by a dark solid line.

An important property of visibility graph is, in an obstructed space the edges of the shortest path are the edges of the visibility graph [50].
CHAPTER 3. OBSTRUCTED DISTANCE COMPUTATION

There are four major approaches [51–53] for constructing the visibility graph. The comparison between them are studied in [54]. According to the comparison, the naive approach has a running time of \(O(n^3)\). Lee et al. [51] developed the first nontrivial solution to the visibility problem running in \(O(n^2 \log n)\) time [51]. Welzl et al. gave an algorithm that is based on rotation trees and runs on \(O(n^2)\) time.

The idea of Ghosh and Mount’s solution [53] is a planar scan left to right proceeding by a variant of the Mehlhorn triangulation. It runs in \(O(|e| + n \log n)\) time. The \(n \log n\) is the time for triangulation and \(|e|\) is the size of the visibility graph.

The last two are mostly of theoretical interest as they include complex data structure handling. In this thesis, we follow the algorithm proposed by Lee et al. to construct the visibility graph.

In our current problem, due to the huge size of spatial datasets, it’s not feasible to keep the whole visibility graph in the main memory. Thus, we construct the visibility graph incrementally. We add only those obstacles and data points in the graph which are relevant to the query. We efficiently update the graph when new data points or obstacles arrive. We remove data points or obstacles which are not necessary for our query and keep the graph small.

3.1.1 Incremental Visibility Graph Computation Strategy

To update an existing visibility graph paper, the authors in [9] introduced the following strategies which we also use for the incremental visibility graph construction:
Adding a new obstacle:
To add a new obstacle \( o \) in the visibility graph \( G \), we add all the vertices \( V' \) of \( o \) to the graph and create new edges with all the visible vertices from \( V' \). If there are any existing edges that crosses the interior of \( o \), we remove them from the graph.

Adding a new data point:
To add a new data point \( p' \) in the visibility graph \( G \), we add \( p' \) in the graph and find all the visible vertices \( V' \) of \( p' \) and create new edges between them.

Deleting an existing data point:
To remove a data point \( p \) from the visibility graph \( G \), we just simply remove the point and its incident edges from the graph.

3.2 Obstructed Distance between two Points

In the literature, algorithms [9, 10, 12] already exist for computing obstructed distance between two points \( p \) and \( q \). In this section, we describe those algorithms in short and find the limitations of the existing algorithms.

The Algorithm for computing the obstructed distance between two points \( p \) and \( q \) was first studied in [9]. Without loss of generality, we explain the algorithm in short with Figure 3.2.

At first, the algorithm computes the Euclidean distance \( \text{dist}_E(p, q) \) between the two points \( p \) and \( q \). Then the algorithm retrieves the obstacles \( o_1, o_2 \) that are within the range \( \text{dist}_E(p, q) \) and constructs initial visibility graph with \( o_1, o_2, p \) and \( q \). Using the visibility graph and Dijkstra shortest path algorithm, the algorithm then computes an initial obstructed distance as \( \text{dist}_{O_1}(p, q) \) (Figure 3.2).

However, this initial distance \( \text{dist}_{O_1}(p, q) \), might not be the actual obstructed distance as there might be other obstacles \( o_3, o_4, o_5 \) which effects the actual obstructed distance between \( p \) and \( q \). Thus, the algorithm executes a circular range query to retrieve obstacles \( o_3, o_4, o_5 \) centering \( q \) and with radius \( \text{dist}_{O_1}(p, q) \).

The second obstructed distance \( \text{dist}_{O_2}(p, q) \) (Figure 3.2) is calculated using \( o_1, o_2, o_3, o_4, o_5, p \) and \( q \) by expanding the visibility graph to include new obstacles. In this step the visibility graph from previous
Figure 3.2: Obstructed distance \( \text{dist}_O(p, q) \) calculation between \( p \) and \( q \)

stage is updated using the new obstacles. In this way the algorithm keeps repeating range query for any new obstacles which might affect the obstructed distance between \( p \) and \( q \). The Algorithm terminates when the latest iteration failed to discover any new obstacle or the obstructed distance is same in between two subsequent iterations.

In [10] Xia and Hsu proposed an algorithm for obstructed distance computation which for the first time introduced the idea of retrieving obstacles incrementally which filters out a large number of obstacles. By retrieving obstacles incrementally, the algorithm builds a small visibility graph that is relevant to the query. Initially, the visibility graph contains only the two points \( p \) and \( q \) and the core obstacles \( o_1 \) (obstacles that intersect with the straight line between this two points) (Figure 3.3a). At the next step, the Algorithm computes a candidate shortest path \( \text{dist}_O(p, q) \) from the visibility graph and checks whether \( \text{dist}_O(p, q) \) intersects any obstacles that are not included in the current visibility graph. The algorithm recomputes the obstructed distance between \( p \) and \( q \), until no more obstacles intersect the shortest path (Figure 3.3b).

In [12] Gao and Yang proposed an algorithm for obstructed distance computation which also expands a local visibility graph incrementally and maintains a threshold. The paper proves that if the obstructed distance \( \text{dist}_O(p, q) \) between a data point \( p \) and a query point \( q \) is computed by considering
CHAPTER 3. OBSTRUCTED DISTANCE COMPUTATION

22

Figure 3.3: Obstructed distance $\text{dist}_O(p,q)$ calculation between $p$ and $q$

all the obstacles in the region bounded by a threshold $\lambda$ from the query point $q$, and $\text{dist}_O(p,q) \leq \lambda$ and then $\text{dist}_O(p,q)$ is the real obstructed distance between the data point $p$ and the query point $q$. This algorithm introduces an idea of reusing the already computed obstructed distance computations between the query point $q$ and other obstacle vertices, which reduces no of obstructed distance computation.

These obstructed distance calculation strategy do not describe how to compute obstructed distance from a single data point to multiple query points. For our $k$OGNN query we need to calculate aggregate obstructed distance from a single data point $p$ to multiple query points $Q$. By applying the already existing algorithm recursively between $p$ and each of the query points $q \in Q$ incurs high query processing overhead. Thus, we propose two new algorithms in the following sections to overcome this problem of computing obstructed distance from a single data point to multiple query points.
3.3 Aggregate Obstructed Distance from Multiple Points to a Single Point

In this section, we propose algorithms to calculate aggregate obstructed distance between a data point \( p \) and a set of query points \( Q \). We present two algorithms for processing aggregate obstructed distance between a data point \( p \) and a set of query points \( Q \): Single Point Aggregate Obstructed Distance (SPAOD) computation and Multi Point Aggregate Obstructed Distance (MPAOD) computation.

The key difference between these two algorithms is, SPAOD does not retrieve the same obstacle multiple times like MPAOD, but SPAOD may retrieve some extra obstacles. SPAOD also reuses the already computed obstructed shortest path distance when computing obstructed shortest path distance for subsequent data points. On the other hand, though MPAOD does not reuse already computed obstructed shortest path distances, it filters out a lot of obstacles which does not intersect with the current shortest path between a data point \( p \) and multiple query points \( Q \). Thus, MPAOD keeps the visibility graph small. A small visibility graph make the obstructed distance computation faster. MPAOD works better in a distribution where the data points are located far apart from one another and the chance of same obstacle retrieval multiple times is less. Table 3.1 summarizes the notations used in the rest of this section.

For both the algorithms the aggregate obstructed distance \( \text{dist}_{AO}(p, Q) \) between \( p \) and \( Q \) is calculated by using an aggregate function over individual obstructed distances.

If \( f(p, Q) = \text{SUM} \), then,

\[
\text{dist}_{AO}(p, Q) = \sum_{i=1}^{n} \text{dist}_O(p, q_i), \text{ where } q_i \text{ is the } i^{th} \text{ point}
\]

If \( f(p, Q) = \text{MAX} \), then,

\[
\text{dist}_{AO}(p, Q) = \max_{1 \leq i \leq n} \text{dist}_O(p, q_i), \text{ where } q_i \text{ is the } i^{th} \text{ point}
\]

3.3.1 Single Point Based Aggregate Obstructed Distance (SPAOD) Computation

Algorithm 1 demonstrates the complete algorithm for finding the aggregate obstructed distance between a data point \( p \) and multiple query points \( Q \). The key idea of this algorithm is to reuse the already computed obstructed distances between any query point \( q \) and any vertex \( v \), where a vertex is
### Table 3.1: Notation used and their meanings

<table>
<thead>
<tr>
<th>Notation</th>
<th>Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p$</td>
<td>A data point</td>
</tr>
<tr>
<td>$Q$</td>
<td>A set of query points $Q = {q_1, q_2, \ldots, q_n}$, where $q_i$ represents $i^{th}$ query point</td>
</tr>
<tr>
<td>$RT_O$</td>
<td>An obstacle RTree</td>
</tr>
<tr>
<td>$\text{dist}_O(p, q)$</td>
<td>The obstructed distance between a query point $q$ and a data point $p$</td>
</tr>
<tr>
<td>$\text{dist}_{AO}(p, Q)$</td>
<td>The aggregate obstructed distance between a data point $p$ and a set of query points $Q$</td>
</tr>
<tr>
<td>IOR $(p, RT_O, d_{\text{max}})$</td>
<td>Incrementally retrieves nearest obstacles with respect to $p$, which are within distance $d_{\text{max}}$ from $p$. More specifically, if the obstacles, which are within distance $d'<em>{\text{max}}$ from $p$ are already retrieved in previous iterations, then in the current iteration, the function retrieves the obstacles which are within distance $(d'</em>{\text{max}} - d_{\text{max}})$ from $p$.</td>
</tr>
<tr>
<td>$SP_{p,q}$</td>
<td>Shortest path between points $p$ and $q$</td>
</tr>
<tr>
<td>$L_Q$</td>
<td>Subset of $Q$, whose obstructed distance from a data point $p$ needs to be computed</td>
</tr>
<tr>
<td>$\text{CompObsDist} (q,p, LVG)$</td>
<td>Finds the distance of the shortest path between $p$ and $q$ from the local visibility graph $LVG$ using Dijkstra algorithm</td>
</tr>
<tr>
<td>$c_Q$</td>
<td>Geometric centroid of the set of query points $Q$</td>
</tr>
<tr>
<td>$LVG$</td>
<td>A local visibility graph</td>
</tr>
</tbody>
</table>
a node in the visibility graph. In [12], Gao and Yang proves that if the obstructed distance $d_{O}(p,q)$ between a data point $p$ and a query point $q$ is computed by considering all the obstacles in the region bounded by a threshold $\lambda$ from the query point $q$, and $d_{O}(p,q) \leq \lambda$ and then $d_{O}(p,q)$ is the real obstructed distance between the data point $p$ and the query point $q$. We use this idea to compute obstructed distances between any query point $q$ and any vertex $v$ in the obstructed space.

The input of Algorithm 1 is a set of query points (i.e, the location of the group of users) $Q$, a data point $p$, an obstacle RTree $RT_{O}$ and a local visibility graph $LVG$. The output of the algorithm is the aggregate obstructed distance between the data point and the group of users.

The Algorithm divides all the vertexes in three lists

1. $L_{R}[1 \ldots n]$ : A list of vertices for a set of query points $Q$ with size $n$ whose obstructed distances from $i^{th}$ query point $q_{i} (1 \leq i \leq n)$ have been already computed. Each $L_{R}[i]$ contains a list of vertices for the $i^{th}$ query point whose obstructed distances from $q_{i}$ is already computed.

2. $L_{C}[1 \ldots n]$ : A list of vertices for a set of query points $Q$ with size $n$ whose obstructed distances from $i^{th}$ query point $q_{i} (1 \leq i \leq n)$ needs to be computed in the current iteration. Each $L_{R}[i]$ contains a list of vertices for the $i^{th}$ query point whose obstructed distance needs to compute in the current iteration.

3. $L_{N}[1 \ldots n]$ : A list of vertices for a set of query points $Q$ with size $n$ whose obstructed distance from $i^{th}$ query point $q_{i} (1 \leq i \leq n)$ will be computed in the next iteration. Each $L_{R}[i]$ contains a list of vertices for the $i^{th}$ query point whose obstructed distance with respect to the query point $q_{i}$ will be computed in the next iteration.

A threshold $t$ is maintained to keep trace of incremental obstacle retrieval and it is initialized as follows:

- $t$ : For the first iteration $t$ is initialized with the maximum value between the result of the summation of the Euclidean distance $d_{E}(c_{Q},q_{i})$ between the centroid $c_{Q}$ and the query point $q_{i}$ and the Euclidean distance $d_{E}(p,q_{i})$ between the data point $p$ and the query point $q_{i}$.
  $$t = \max_{1 \leq i \leq n}(d_{E}(c_{Q},q_{i}) + d_{E}(p,q_{i}))$$

For all the subsequent iterations, $t$ is initialized with the maximum value between the result of the summation of the Euclidean distance $d_{E}(c_{Q},q_{i})$ between the centroid $c_{Q}$ and the query
Algorithm 1: CompAggObsDist\((Q, p, RT_O, LVG)\)

**Input:** A set of query points \(Q = \{q_1, q_2, \ldots q_n\}\), a data point \(p\), an obstacle RTree \(RT_O\), a local visibility graph \(LVG\)

**Output:** The aggregate obstructed distance \(\text{dist}_{AO}(p, Q)\) of \(p\) from \(Q\)

1. Initialize \((L_R, \text{dist}_O(p, Q), L_C, t)\)
2. repeat
   3. \(O \leftarrow \text{IOR}(c_Q, RT_O, t)\)
   4. foreach \(q_i \in Q\) do
      5. if \(p \notin L_R[i]\) then
         6. foreach \(v \in L_R[i]\) do
            7. if isVisible \((v, p)\) then relax \(\text{dist}_O(p, q)\)
            8. if \(\text{dist}_O(p, q) \leq t\) then
               9. Add \(p\) in \(L_R[i]\); continue;
         10. else Add corner vertices of every \(o \in O\) in \(L_C[i]\) and in local visibility graph \(LVG\)
      11. foreach \(v \in L_N[i]\) do
         12. foreach \(o \in LVG\) do
            13. if intersects \((\text{adjacent}(v, o)) \text{ OR dist}_O(v, q) > t\) then
               14. Remove \(v\) from \(L_N[i]\) ; Add \(v\) in \(L_C[i]\)
            15. else if \(\text{dist}_O(v, q) \leq t\) then
               16. Remove \(v\) from \(L_N[i]\) ; Add \(v\) in \(L_R[i]\)
      17. Compute \(\text{dist}_O(p, q)\) and obstructed distances between every vertex in \(L_C\) and \(q\)
      18. foreach \(v \in L_C\) do
         19. Remove \(v\) from \(L_C[i]\)
         20. if \(\text{dist}_O(v, q) \leq t\) then Add \(v\) in \(L_R[i]\); else Add \(v\) in \(L_N[i]\)
      21. \(t \leftarrow \max_{i=1}^{n}(\text{dist}_E(c_Q, q_i) + \text{dist}_O(p, q_i))\)
   22. until \(\exists_{i=1}^{n}\text{dist}_O(p, q_i) > t\)
   23. \(\text{dist}_{AO} \leftarrow f_{i=1}^{n}\text{dist}_O(p, q_i)\)
24. return \(\text{dist}_{AO}\)
point $q_i$ and the obstructed distance $dist_O(p, q_i)$ between the data point $p$ and the query point $q_i$.

$$t = \max_{i=1}^n (dist_E(cQ, q_i) + dist_O(p, q_i))$$

The algorithm uses a function $Initialize(L_R, dist_O(p, Q), L_C, t)$ to initialize the list $L_R, L_C$, obstructed distance between data point $p$ and query points $Q$ and threshold $t$ as follows:

- Initialize $(L_R, dist_{OA}(p, Q), L_C, t)$: Initializes the variables as follows
  - Each list $L_R[i]$ is initialized with the $i^{th}$ query point. i.e., $\forall_{i=1}^n L_R[i] \leftarrow q_i$
  - Initializes the obstructed distance $dist_O(p, q_i)$ between a data point $p$ and all the query points $q \in Q$ as $\infty$. i.e., $\forall_{i=1}^n dist_O(p, q_i) \leftarrow \infty$
  - Initializes the aggregate obstructed distance $dist_O(p, Q)$ between a data point $p$ and a set of query points $Q$ as $\infty$. i.e., $dist_{OA}(p, Q) \leftarrow \infty$.
  - Initializes threshold $t$
  - Clears the set $L_C$, i.e, $L_C \leftarrow \emptyset$

The algorithm uses some new functions that are described below:

- isVisible $(v, p)$: Returns true if a vertex $v$ is visible to $p$. A vertex $v$ is visible to $p$ if the straight line between $v$ to $p$ does not cross any obstacle.

- relax $(dist_O(p, q))$: Relaxes the edge between two points $p$ and $q$.

  If $v$ is a vertex which is visible to $p$ and $v$’s obstructed distance from $q$, $dist_O(v, q)$ is already computed, then we update the obstructed distance $dist_O(p, q)$ between a data point $p$ and a query point $q$ with the new distance found by adding the two distances, $dist_O(v, q)$ and the Euclidean distance $dist_E(v, p)$, if the new distance $dist_O(p, q)$ is smaller than the previous.

$$dist_O(p, q) = \begin{cases} 
\text{if } dist_O(v, q) + dist_E(v, p) \leq dist_O(p, q), & dist_O(v, q) + dist_E(v, p) \\
dist_O(p, q), & \text{otherwise}
\end{cases}$$

- adjacent $(v)$: Function to find the adjacent edges of $v$.

- intersects (adjacent $(v), o$): Returns true if obstacle $o$ intersects any adjacent edges of $v$
The Algorithm retrieves new obstacles with respect to the geometric centroid $c_Q$ of the query points $Q$. Initially the algorithm retrieves the obstacles incrementally within the threshold $t$ with respect to $c_Q$ (Line 3). Then, it calculates the obstructed distance between $p$ and all the query points in $Q$ iteratively.

Without loss of generality, we explain the obstructed distance $dist_O(p, q_1)$ computation between the data point $p$ and one of the query points in $q_1 \in Q$. The Algorithm tries to reuse already computed obstructed distances to compute the obstructed distance between $p$ and $q_1$. For this reason the algorithm checks the visibility of the vertexes in $L_R[1]$ with $p$ and updates the obstructed distance between $p$ and $q_1$ (Lines 6-7). The Algorithm uses two functions isVisible ($v, p$) and relax ($dist_O(p, q)$) in this step.

According to Gao and Yang [12] if the obstructed distance $dist_O(p, q) \leq t$ then we are confirmed that we have found our actual obstructed distance between $p$ and $q_1$ and continue searching for the obstructed distance between $p$ and other query points in $Q$ (i.e, $q_2, q_3$ etc) (Lines 8-9).

However, if $dist_O(p, q) > t$, then the algorithm adds all the corner vertices of every obstacle retrieved in $L_C[1]$ and in local visibility graph $LVG$ for obstructed distance computation (Line 10). Addition of new obstacles may affect the visibility in the vertexes in $L_N[1]$. To ensure thus, the algorithm checks for three conditions (Lines 12-18) for each of the vertexes $v \in L_N[1] :$

1. If the adjacent edges of $v$ in the local visibility graph intersects any obstacle in the $LVG$ then the algorithm removes $v$ from $L_N[1]$ and adds $v$ in $L_C[1]$, so that the obstructed distance between $v$ and $q_1$ can be computed in this iteration

2. If the obstructed distance $dist_O(v, q_1) > t$ between $v$ and $q_1$ then, the algorithm removes $v$ from $L_N[1]$ and adds $v$ in $L_C[1]$, so that the obstructed distance between $v$ and $q_1$ can be computed in this iteration

3. If the obstructed distance $dist_O(v, q_1) \leq t$ between $v$ and $q_1$ then, the algorithm removes $v$ from $L_N[1]$ and adds $v$ in $L_R[1]$, because the actual obstructed distance between $v$ and $q_1$ is found.

Thereafter, the algorithm computes obstructed distance between $q_1$ and every vertex in $L_C$ and the data point $p$. If the obstructed distance for any vertex $v \in L_C[1]$, $dist_O(v, q) \leq t$ then the algorithm removes $v$ from $L_N[1]$ and adds $v$ in $L_R[1]$ as it’s obstructed distance with $q_1$ is actual. Otherwise, the vertex $v$ is added in $L_N[1]$ for future computation (Lines 19-22).
The first round iteration for obstructed distance computation between \( p \) and \( q_1 \) ends here. The Algorithm computes the obstructed distance between \( p \) and other query points in \( Q \) in the same technique and updates the threshold \( t \) accordingly (Line 23). The repeat loop (Lines 2-24) terminates when all the individual obstructed distances is less than the latest threshold \( t \). After the procedure finishes computing all the individual obstructed distances, the algorithm applies the aggregate function \( f \) on the obstructed distances and computes the aggregate obstructed distance between the data point \( p \) and the query points \( Q \) (Line 25) and returns the aggregate obstructed distance \( \text{dist}_{AO}(p, Q) \) (Line 26).

**Table 3.2: SPAOD Example**

<table>
<thead>
<tr>
<th>Variables</th>
<th>Initialization</th>
<th>Lines 10-11</th>
<th>Lines 19-22</th>
</tr>
</thead>
<tbody>
<tr>
<td>( L_R^2 )</td>
<td>( q_2 )</td>
<td>( q_2 )</td>
<td>( {b, c, e, f} )</td>
</tr>
<tr>
<td>( L_C^2 )</td>
<td>( \emptyset )</td>
<td>( {a, b, c, d, e, f, g, h} )</td>
<td>( \emptyset )</td>
</tr>
<tr>
<td>( L_N^2 )</td>
<td>( \emptyset )</td>
<td>( \emptyset )</td>
<td>( {a, d, g, h} )</td>
</tr>
<tr>
<td>( LVG )</td>
<td>( \emptyset )</td>
<td>( {a, b, c, d, e, f, g, h, p, q_1, q_2, q_3} )</td>
<td>( {a, b, c, d, e, f, g, h, p, q_1, q_2, q_3} )</td>
</tr>
</tbody>
</table>

Figure 3.4a shows the first step of Algorithm 1 for computing the aggregate obstructed distance...
between the data point \( p \) with 3 query points \( Q = \{ q_1, q_2, q_3 \} \). The Algorithm starts by initializing the threshold \( t \) with the Euclidean distance \( t_1 = \text{dist}_E(c_Q, q_2) + \text{dist}_O(p, q_2) \). The list \( L_R[1...3] \) initially contains the respective query points of each list. For example, \( L_R[2] = q_2 \).

The repeat loop at Lines 2-24 begins by retrieving 2 obstacles \( o_1 \) and \( o_2 \) (Figure 3.4b). The Algorithm then continue with a foreach (Lines 4-22) loop of all the query points. We give an example of the obstructed distance calculation between \( p \) and \( q_2 \). At Lines 6-9, the algorithm checks whether \( p \) is visible to any of the entries in the \( L_R[2] \) and calculate the obstructed distance between \( p \) and \( q_2 \). Since, \( L_R[2] \) only contains \( q_2 \) and \( p \) and \( q_2 \) are mutually invisible as they have an obstacle \( o_2 \) in between them so we continue with the next part of the algorithm.

At this stage, the algorithm updates \( L_C[2] = \{ a, b, c, d, e, f, g, h \} \) and local visibility graph \( LVG = \{ a, b, c, d, e, f, g, h, p, q_1, q_2, q_3 \} \). Note that, in this round, SPAOD skips foreach-loop in Lines 12-18 since \( L_N[2] = \emptyset \).

After that, the obstructed distance between the points in \( L_C[2] \) and \( q_2 \) is computed and checked whether any of the entries in \( L_C[2] \) can be added in \( L_R[2] \) or \( L_N[2] \). At this stage we update, \( L_R[2] = \{ b, c, e, f \} \) and \( L_N[2] = \{ a, d, g, h \} \). Table 3.2 summarizes these above steps. At the end of the foreach loop (Lines 4-22), the threshold is updated and the Algorithm continues finding the obstructed distance between \( p \) and query points \( Q = \{ q_1, q_2, q_3 \} \) and until the real aggregate obstructed distance is found.

3.3.1.1 Correctness of Algorithm CompAggObsDist (SPAOD)

**Lemma 3.3.1** Algorithm 1 (CompAggObsDist) finds the real aggregate obstructed distance that goes from a data point \( p \) to a set of query points \( Q = (q_1, q_2, \ldots q_n) \) and avoids all obstacles of the obstacle RTree RT\(_O\).

**Proof** (By contradiction)

Let, there is an obstacle \( o \) which is not retrieved by Algorithm 1 and \( o \) changes the aggregate obstructed distance between a data point \( p \) and a set of query points \( Q = (q_1, q_2, \ldots q_n) \).

Algorithm 1 (CompAggObsDist) incrementally retrieve obstacles that are within a threshold \( t \) from the centroid \( c_Q \) of the query points \( Q \), and adds them in the visibility graph. The Algorithm then compute the aggregate obstructed distance \( \text{dist}_{AO}(p, Q) \) between \( p \) and \( Q \) by applying an aggregate function \( f = \text{SUM}/\text{MAX} \) over all the query points \( \forall i \leq n \text{dist}_O(p, q_i) \). We stop retrieving more obstacles
when the obstructed distances $\forall_{i=1}^{n} \text{dist}_O(p, q_i)$ between a data point $p$ and all the query points in $Q = (q_1, q_2, \ldots q_n)$ is less than the threshold $t$.

Since $o$ is not retrieved by algorithm 1 so $o$ is located at a distance larger than the threshold $t$ from $cQ$.

In [12] Gao and Yang proves that if the obstructed distance $\text{dist}_O(p, q)$ between a data point $p$ and a query point $q$ is computed by considering all the obstacles $\forall_{k=1}^{n} o_k$ in the region bounded by a threshold $\lambda$ from the query point $q$, and $\text{dist}_O(p, q) \leq \lambda$ and then $\text{dist}_O(p, q)$ is the real obstructed distance between the data point $p$ and the query point $q$.

Since $o$ is not retrieved by Algorithm 1 so $o$ cannot change any of the individual obstructed distances between $p$ and $Q$, $\forall_{i=1}^{n} \text{dist}_O(p, q_i)$. Since, aggregate obstructed distance $\text{dist}_A O(p, Q)$ is computed by applying an aggregate function $(\forall_{i=1}^{n} \text{dist}_O(p, q_i))$ over all the individual obstructed distances $\text{dist}_O(p, q)$, so $o$ cannot change aggregate obstructed distance $\text{dist}_A O(p, Q)$ too.

Thus, our assumption that $o$ changes the aggregate obstructed distance $\text{dist}_A O(p, Q)$ was wrong.

Thus, we can say that, when Algorithm 1 terminates we have the actual aggregate obstructed distance $\text{dist}_A O(p, Q)$ between a data point $p$ and a set of query points $Q$.

\[\Box\]

### 3.3.1.2 Time Complexity of Algorithm CompAggObsDist (SPAOD)

The worst case time complexity of Algorithm 1 (CompAggObsDist) is: $O(|T| |Q| |O| |V|^3 \log |V|)$, where,

- $|T|$ is the number of times the threshold $t$ will be updated,
- $|Q|$ is the number of query points,
- $|O|$ is the number of obstacle vertices, and
- $|V|$ is the number of nodes in the visibility graph.

### 3.3.2 Multi Point Based Aggregate Obstructed Distance (MPAOD) Computation

Algorithm 2 shows the MPAOD computation strategy for the aggregate obstructed distance computation between a data point $p$ and multiple query points $Q$. The key idea of this algorithm is, when
Algorithm 2: CompAggObsDist($Q, p, RT_O, LVG$)

**Input:** A set of query points $Q = (q_1, q_2, \ldots, q_n)$, a data point $p$, an obstacle RTree $RT_O$, a local visibility graph $LVG$

**Output:** The aggregate obstructed distance $dist_{AO}(p, Q)$ of $p$ from $Q$

1. **foreach** $q \in Q$ **do**
   
2.   
   $dist_O(p, q_i) = dist_E(p, q_i)$

3. **repeat**

4.   
   $d_{max} \leftarrow \max_{1 \leq i \leq n} dist_O(p, q_i)$

5.   
   $O \leftarrow IOR(p, RT_O, d_{max})$

6. **foreach** $o \in O$ **do**

7.   
   **foreach** $q \in Q$ **do**

8.     
     if $o$ intersects $SP_{p,q}$ then

9.     
     Add $q$ in $L_Q$

10.    
     Add $o$ in $LVG$

11. **foreach** $q \in L_Q$ **do**

12.   
    $dist_O(p, q) = compObsDist(q, p, LVG)$

13. **until** $L_Q = \emptyset$

14. $dist_{AO} \leftarrow \sum_{i=1}^{n} dist_O(p, q_i)$

15. **return** $dist_{AO}$
computing aggregate obstructed distance between a data point \( p \) and a set of query points \( Q \), we retrieve obstacles having distances equal to a threshold with respect to the data point \( p \). The Algorithm repeats the same for the subsequent data points and retrieve a new set of obstacles. The new set of obstacles may include already retrieved obstacles, which invokes same obstacle retrieval multiple times, but it does not retrieve any unnecessary obstacle. MPAOD also checks the intersection of the retrieved obstacles with the latest shortest path. Thus prunes a huge number of obstacles and keeps the visibility graph. A small visibility graph makes the obstructed distance computations faster.

The input of Algorithm 1 is a set of query points (i.e., the locations of group of users) \( Q \), a data point \( p \), an obstacle RTree \( RT_O \) and a local visibility graph \( LVG \). The output of the algorithm is the aggregate obstructed distance between the data point and the group of users.

The Algorithm at first computes the individual Euclidean distances between the data point \( p \) and each of the query points \( q \in Q \) (Lines 1-2) and assign them as the initial obstructed distances. The Algorithm stores the maximum distance among all the distances computed in this step in a variable \( d_{\text{max}} \) (Line 4). It then incrementally retrieves all the obstacles that are within the distance \( d_{\text{max}} \), centering the data point \( p \) by using a function IOR \( (p, RT_O, d_{\text{max}}) \) (Line 5). The intuition behind retrieving obstacles centering the data point \( p \) is, it is expected that the obstacles that really effects the obstructed distances between \( p \) and \( Q \) is located near \( p \). An obstacle which is far away from \( p \) cannot actually effect the obstructed distance between the data point \( p \) and the query points \( Q \).

However, after retrieving the obstacles, the algorithm filters out those obstacles which does not intersects any of the already computed shortest path between the data point \( p \) and the query points \( Q \). It also stores the necessary query points in a set \( L_Q \), whose obstructed distances need to be recomputed. We denote the shortest path between a data point \( p \) and a query point \( Q \) as \( SP_{p,q} \) (Lines 6-10). The re-computation of obstructed distance between a data point \( p \) and a query point \( q \) is required only when the shortest path between \( p \) and \( q \) intersects any obstacles retrieved by the incremental obstacle retrieval.

After filtering out unnecessary obstacles the algorithm updates the visibility graph with the new obstacles and re computes the obstructed distances between \( p \) and all the query points \( q \in L_Q \) (Lines 7-12). The procedure repeats until the shortest path intersects no new obstacles or the \( L_Q \) is empty.
Finally after computing all the individual obstructed distances, the algorithm applies the aggregate function $f(\text{sum}/\text{max})$ on the obstructed distances and computes the aggregate obstructed distance between the data point $p$ and the query points $Q$ and returns the aggregate obstructed distance $\text{dist}_{AO}(p, Q)$ (Lines 14-15).

Figure 3.5: Example of MPAOD

Figure 3.5a shows the first iteration of algorithm 2, in this iteration incremental obstacle retrieval retrieves obstacle $o_1, o_2, o_3$. The shortest path between the data point $p$ and the query point $q_1$, intersects obstacle $o_2$ and the shortest path between the data point $p$ and the query point $q_3$, intersects obstacle $o_1$, so $q_1$ and $q_3$ are inserted in $L_Q$.

Figure 3.5b shows the second iteration of algorithm 2, where more obstacle ($o_4$) is retrieved by the incremental obstacle retrieval and checked for intersection with the query points $Q$. Since, the new shortest path does not intersect any of the obstacles, so the algorithm terminates after computing the aggregate obstructed distance.
3.3.2.1 Correctness of Algorithm CompAggObsDist (MPAOD)

Lemma 3.3.2 Algorithm CompAggObsDist finds the aggregate obstructed distance from a data point \( p \) to a set of query points \( Q = (q_1, q_2, \ldots q_n) \).

Proof (By contradiction)

Let, there be an obstacle \( o \), which is not retrieved by Algorithm CompAggObsDist (Algorithm 2) and \( o \) changes the aggregate obstructed distance \( dist_{AO}(p, Q) \) computed by algorithm CompAggObsDist. Algorithm CompAggObsDist incrementally retrieves obstacles that block current shortest path from \( p \) to all the query points in \( Q \). If there are no new obstacles retrieved that blocks the current shortest path from \( p \) to all the query points in \( Q \), the Algorithm computes the aggregate obstructed distance \( dist_{AO}(p, Q) \) between \( p \) and \( Q \).

In [10] Xia and Hsu proves that, if an obstacle does not intersect the current shortest path between two points in the visibility graph, then that obstacle cannot change the already computed obstructed distance between those points.

Since \( o \) is not retrieved by algorithm CompAggObsDist so it does not intersect any of the shortest paths from \( p \) to the query points in \( Q \).

Thus, the obstacle \( o \) cannot change any of the obstructed distances between \( p \) and a set of query points \( Q \). Since, aggregate obstructed distance \( dist_{AO}(p, Q) \) is computed by applying an aggregate function \( f \) (SUM/MAX) over all the individual obstructed distances \( dist_O(p, q) \), \( o \) cannot change aggregate obstructed distance \( dist_{AO}(p, Q) \) too, which contradicts our assumption.

Thus algorithm CompAggObsDist computes the actual aggregate obstructed distance \( dist_{AO}(p, Q) \) between a data point \( p \) and a set of query points \( Q \).

□

We want to mention here that, both the algorithms SPAOD and MPAOD are using the same visibility graph for all the obstructed distance computations, this may sound like we are storing a large visibility graph in our memory which is costly. But, we are constructing the visibility graph only when needed and we store only those nodes, edges and obstacles in the graph that are relevant to our query. At the very beginning the visibility graph is the minimal one and then we enlarge our graph incrementally. On the other hand it’s very costly to reconstruct the visibility graph every time from
scratch due to complex algorithms behind visibility graph construction. The incremental visibility graph construction strategy is described in the Section 3.1.1.

3.3.2.2 Time Complexity of Algorithm CompAggObsDist (MPAOD)

The worst case time complexity of Algorithm 1 (CompAggObsDist) is: $O(|O||Q|^2|V|^2 \log |V|)$, where, $|Q|$ is the number of query points, $|O|$ is the number of obstacle vertices, and $|V|$ is the number of nodes in the visibility graph.
We develop two algorithms: CBQM (Centroid Based Query Method) and GBQM (Group Based Query Method) for processing obstructed group nearest neighbor (OGNN) queries in this chapter. The key difference between them is, CBQM is based on the nearest neighbor with respect to the centroid of the query points and GBQM is based on the Euclidean GNN of the query points. CBQM incrementally retrieves Euclidean nearest neighbors with respect to the centroid of the query points $Q$ until the best $k$ OGNNs have been found. Whereas, GBQM, incrementally retrieves Euclidean GNNs with respect to the location of the group users $Q$ and compute its aggregate obstructed distance, until the best $k$ OGNNs have been found.

In Sections 4.1 and 4.2, we propose Algorithms CBQM$_\text{SUM}$ and CBQM$_\text{MAX}$ which finds OGNN for aggregate function SUM and aggregate function MAX, respectively, in the centroid based approach. Section 4.3 explains Algorithm GBQM which finds OGNN for both aggregate function SUM and MAX in the group based approach. For all of the algorithms we demonstrate the retrieval of a single ($k = 1$) OGNN and then extend it to find up to $k > 1$.

Table 4.1 summarizes the notations used in the rest of this chapter.

### 4.1 Centroid Based Query Method (CBQM) for SUM

In this section, we explain Algorithm CBQM$_\text{SUM}$ for retrieving OGNN for an aggregate function SUM. Algorithm CBQM$_\text{SUM}$ uses Euclidean nearest neighbor of the centroid $c_Q$ of the query points $Q$ to find out the obstructed group nearest neighbor between a data point $p$ and the query points $Q$. 
### Table 4.1: Notation used and their meanings

<table>
<thead>
<tr>
<th>Notation</th>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p$</td>
<td>A data point</td>
<td></td>
</tr>
<tr>
<td>$Q$</td>
<td>A set of query points $Q = {q_1, q_2, \ldots, q_n}$, where $q_i$ represents the $i^{th}$ query point</td>
<td></td>
</tr>
<tr>
<td>$RT_O$</td>
<td>An obstacle RTree</td>
<td></td>
</tr>
<tr>
<td>$RT_P$</td>
<td>A data point RTree</td>
<td></td>
</tr>
<tr>
<td>$A[1\ldots k]$</td>
<td>An array to store obstructed group nearest neighbors of $Q$</td>
<td></td>
</tr>
<tr>
<td>$d_{k_{\text{max}}}$</td>
<td>Distance of the current $k^{th}$ best obstructed group nearest neighbor e.g. $p$ computed so far.</td>
<td></td>
</tr>
<tr>
<td>$\text{update}(A, d_{k_{\text{max}}}, p, \text{dist}_{AO}(p, Q))$</td>
<td>Updates $A$ and $d_{k_{\text{max}}}$ with respect to the current best obstructed group nearest neighbor $p$ and $p$'s aggregate obstructed distance $\text{dist}_{AO}(p, Q)$</td>
<td></td>
</tr>
<tr>
<td>$\text{dist}_O(p, q)$</td>
<td>The obstructed distance between a query point $q$ and a data point $p$</td>
<td></td>
</tr>
<tr>
<td>$\text{dist}_{AO}(p, Q)$</td>
<td>The aggregate obstructed distance between a data point $p$ and a set of query points $Q$</td>
<td></td>
</tr>
<tr>
<td>$\text{dist}_{AE}(p, Q)$</td>
<td>The aggregate Euclidean distance between a data point $p$ and a set of query points $Q$</td>
<td></td>
</tr>
<tr>
<td>$c_Q$</td>
<td>Geometric centroid of the set of query points $Q$</td>
<td></td>
</tr>
<tr>
<td>$P_{EG}$</td>
<td>A list to store Euclidean GNNs of $c_Q$</td>
<td></td>
</tr>
<tr>
<td>$P_E$</td>
<td>A list to store Euclidean nearest neighbors of $c_Q$</td>
<td></td>
</tr>
<tr>
<td>$d_{E_{\text{max}}}$</td>
<td>The maximum Euclidean distance between the centroid $c_Q$ and the data points retrieved so far</td>
<td></td>
</tr>
<tr>
<td>$LVG$</td>
<td>A local visibility graph</td>
<td></td>
</tr>
</tbody>
</table>
The input of the algorithm is a set of query points \( Q \), an obstacle RTree \( RT_O \) and a data point RTree \( RT_P \). The output of the algorithm is a set of \( k \)-obstructed group nearest neighbors of \( Q \).

Following are a list of symbols used in this algorithm:

- \( \text{find} kNN_E(c_Q, RT_P, k) \) : Finds \( k \) Euclidean nearest neighbors of \( c_Q \) from the data point RTree \( RT_P \)

- \( \text{findNextNN}_E(c_Q, RT_P) \) : Finds the next Euclidean nearest neighbors \( p \) of \( c_Q \) from the data point RTree \( RT_P \)

In this approach the algorithm at first finds the Euclidean nearest neighbor \( p \). Next, the algorithm finds the aggregate obstructed distance \( dist_{AO}(p,Q) \) between the data point \( p \) and the query points \( Q \) (Lines 2-4). The aggregate obstructed distance calculation strategy is described in Chapter 3.

Then, the algorithm incrementally retrieves the next Euclidean nearest neighbor with respect to the centroid and calculate their aggregate obstructed distance. We demonstrate the searching of the subsequent Euclidean nearest neighbors with a function \( \text{findNextNN}_E(Q, RT_P, k) \) in Algorithm 3. The searching for Euclidean nearest data point with respect to the centroid is continued until the actual obstructed group nearest neighbor have been found (Lines 8-13).

The while loop terminates when the Euclidean distance \( dist_E(p,c_Q) \) for the \( k^{th} \) data point \( p \) is greater than our current OGNN’s average aggregate obstructed distance (Line 8). The intuition behind this termination condition is, In [4] Namnandorj and Chen proves that, the group Euclidean nearest neighbor for a set of query points \( Q \) lies within the radius \( r \) centered at the median \( c_Q \) of \( Q \), where, \( r \) is the average of aggregate Euclidean distance for the nearest neighbor of centroid \( c_Q \).

We use the same idea for finding the obstructed group nearest neighbor, we consider only those data points which have their average aggregate obstructed distance smaller than the Euclidean distance between the Euclidean nearest neighbor of the centroid \( c_Q \).

Algorithm 3 illustrates the complete algorithm for finding \( k (\geq 1) \) obstructed group nearest neighbors for a set of query points \( Q \). The \( k \) Euclidean nearest neighbors of the centroid \( c_Q \) are first obtained using \( RT_P \) using a function \( \text{find} kNN_E(c_Q, RT_P, k) \). Then the algorithm calculates the aggregate obstructed distance of all the candidate points just retrieved (Lines 1-5) and store the data points in the result set \( A \).


Algorithm 3: CBQM\textsubscript{SUM} $(Q,p,k,RT_{O},RT_{P})$

\textbf{Input:} A set of query points $Q = \{q_1, q_2, \ldots, q_n\}$, an obstacle RTree $RT_{O}$, a data point RTree $RT_{P}$

\textbf{Output:} $A$, a set of $k$-obstructed group nearest neighbors of $Q$

1. Initialize $(LVG, c_Q)$
2. $P_E \leftarrow findkNN_E(c_Q, RT_{P}, k)$
3. \textbf{foreach} $p \in P_E$ \textbf{do}
4. \hspace{1em} $\text{dist}_{AO}(p, Q) \leftarrow \text{CompAggObsDist}(Q, p, RT_{O}, LVG)$
5. \hspace{1em} $A \leftarrow P_E$
6. $d_{kmax} \leftarrow \max_{1 \leq i \leq |P_E|} \text{dist}_{AO}(p_i, Q)$
7. $d_{Emax} \leftarrow \max_{1 \leq i \leq |P_E|} \text{dist}_E(p_i, c_Q)$
8. \textbf{while} $d_{Emax} < \frac{d_{kmax}}{n}$ \textbf{do}
9. \hspace{1em} $p \leftarrow findNextNN_E(c_Q, RT_{P})$
10. \hspace{1em} $d_{Emax} \leftarrow dist_E(p, c_Q)$
11. \hspace{1em} $\text{dist}_{AO}(p, Q) \leftarrow \text{CompAggObsDist}(Q, p, RT_{O}, LVG)$
12. \hspace{1em} \textbf{if} $\text{dist}_{AO}(p, Q) < d_{kmax}$ \textbf{then}
13. \hspace{2em} \textbf{update}(A, d_{kmax}, p, \text{dist}_{AO}(p, Q))$
14. \textbf{return} $A$
Initially $d_{k_{max}}$ stores the $k^{th}$ data point’s aggregate obstructed distance, which is currently the farthest data point (Line 6).

Then, the algorithm incrementally retrieves the next Euclidean nearest data point with respect to centroid $c_Q$ using a function $\text{findNextNN}(c_Q, RT_P)$ (Line 9) until the actual obstructed group nearest neighbor has been found and update the result set $A$ and the best aggregate obstructed distance $d_{k_{max}}$ (Lines 8-13).

The while loop terminates when the Euclidean distance $\text{dist}_E(p, c_Q)$ for the $k^{th}$ data point $p$ is greater than our current OGNN’s average aggregate obstructed distance and the algorithm returns the result set $A$ containing the $k$OGNNs of the query point set (Line 14).

We illustrate Algorithm 3 with the example depicted in Figure 4.1, where the data point set $P = \{p_1, p_2, p_3, p_4\}$, query point set $Q = \{q_1, q_2, q_3\}$ and obstacle set $O = \{o_1, o_2, o_3, o_4, o_5\}$. The centroid

![Figure 4.1: Example of aggregate function Sum for CBQM](image)

**Table 4.2: CBQM$_{sum}$ Example**

<table>
<thead>
<tr>
<th>Data Point</th>
<th>$\text{dist}_E(p, c_Q)$</th>
<th>$\text{dist}_{AO}(p, Q)$</th>
<th>Average $\text{dist}_{AO}(p, Q)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p_1$</td>
<td>10</td>
<td>45</td>
<td>15</td>
</tr>
<tr>
<td>$p_2$</td>
<td>12</td>
<td>55</td>
<td>15</td>
</tr>
<tr>
<td>$p_3$</td>
<td>14</td>
<td>43</td>
<td>14.33</td>
</tr>
<tr>
<td>$p_4$</td>
<td>17</td>
<td>-</td>
<td>14.33</td>
</tr>
</tbody>
</table>

We illustrate Algorithm 3 with the example depicted in Figure 4.1, where the data point set $P = \{p_1, p_2, p_3, p_4\}$, query point set $Q = \{q_1, q_2, q_3\}$ and obstacle set $O = \{o_1, o_2, o_3, o_4, o_5\}$. The centroid
c_Q of the query points is also shown in the figure. The Algorithm at first retrieves the first nearest neighbor \( p_1 \) of \( c_Q \) which is at \( \text{dist}_E(p_1, c_Q) = 10 \) distance and calculate \( p_1 \)'s aggregate obstructed distance \( \text{dist}_{AO}(p_1, Q) = 45 \) using one of the algorithms described in Section 3.3 (Figure 4.1a). The Algorithm then incrementally retrieves the next nearest neighbor \( p_2 \) of \( c_Q \) and check whether \( p_2 \) is at a Euclidean distance less than the average aggregate obstructed distance \( (45/3=15) \). As \( p_2 \)'s Euclidean distance \( \text{dist}_E(p_2, c_Q) = 12 \) which is less than 15 we consider it as a candidate point and calculate \( p_2 \)'s aggregate obstructed distance \( \text{dist}_{AO}(p_2, Q) = 55 \). As \( p_2 \)'s aggregate obstructed distance is greater than \( p_1 \)'s so the algorithm rejects \( p_2 \) as an OGNN.

The algorithm keeps on retrieving the next nearest neighbors \( p_3 \) and \( p_4 \) of \( c_Q \). The steps are shown in Table 4.2. In this way, when the algorithm retrieves \( p_4 \) of \( c_Q \), which is at a distance 17 from \( c_Q \) the condition at Line 8 (Figure 4.1b) is false. So the algorithm terminates and returns \( p_1 \) as the OGNN.

4.1.1 Correctness of Algorithm CBQM\_SUM

**Theorem 4.1.1** If \( k \) is the number of required data points for a \( k \)-OGNN query with respect to a set of query points \( Q = \{q_1, q_2, \ldots, q_n\} \) for \( 1 \leq i \leq n \), then the answer set \( A \) from Algorithm 3 includes \( k \) data points that have the smallest \( k \) aggregate obstructed distance for the aggregate function \( \text{sum} \) among all the data points in the data point Rtree RT_p.

**Proof** (By contradiction)
Let, \( p' \) be a data point that is not in \( A \) but has the \( j^{th} \) minimum value \((1 \leq j \leq k)\) for aggregate obstructed distance \( \text{dist}_{AO}(p, Q) \) for function \( \text{sum} \) from \( p \) with respect to a set of query points \( Q(q_1, q_2, \ldots, q_n) \).

Algorithm \( \text{CBQM\_SUM} \) incrementally retrieves the Euclidean nearest neighbor \( p \) of centroid \( c_Q \) and then compute it’s aggregate obstructed distance \( d = \text{dist}_{AO}(p, Q) \) from \( Q \), until the Euclidean distance \( \text{dist}_E(p, c_Q) \) is greater than the average aggregate obstructed distance for the best data point \( p_r \) found so far.

In [4] Namnandorj and Chen proves that, the group Euclidean nearest neighbor for a set of query points \( Q \) lies within the radius \( r \) centered at the median of \( Q, c_Q \), where, \( r \) is the average of aggregate Euclidean distance for the nearest neighbor of centroid \( c_Q \). Any point outside the distance \( r \) from \( c_Q \) gives higher aggregate Euclidean distance than all the point inside \( r \).

Since \( p' \) is not retrieved by Algorithm \( \text{CBQM\_SUM} \), so it is outside of average aggregate obstructed
According to Euclidean lower bound property, the obstructed distance is always greater or equal than the Euclidean distance. Since point \( p' \)'s aggregate Euclidean distance is greater than any point inside \( r \) (i.e. \( p_r \)), there is no chance that \( p' \)'s aggregate obstructed distance will be smaller than \( p_r \)'s, which contradicts our assumption.

Thus, Algorithm \( CBQM_{\text{SUM}} \) finds the actual obstructed group nearest neighbor for a set of query points \( Q \) for the function \( \text{SUM} \).

\[
\square
\]

### 4.1.2 Time Complexity of Algorithm CBQM_{\text{SUM}}

The worst case time complexity of Algorithm 3 (Algorithm \( CBQM_{\text{SUM}} \)) when SPAOD (Algorithm 1) aggregate obstructed distance calculation is used is: \( O(|P||T||Q||O||V|^3 \log |V|) \).

And the worst case time complexity of Algorithm 3 (Algorithm \( CBQM_{\text{SUM}} \)) when MPAOD (Algorithm 2) aggregate obstructed distance calculation is used is: \( O(|P||O||Q|^2|V|^2 \log |V|) \)

where,
- \( |P| \) is the number of data points,
- \( |T| \) is the number of times the threshold \( t \) will be updated,
- \( |Q| \) is the number of query points,
- \( |O| \) is the number of obstacle vertices, and
- \( |V| \) is the number of nodes in the visibility graph

### 4.2 Centroid Based Query Method (CBQM) for MAX

In this section, we explain Algorithm \( CBQM_{\text{MAX}} \) for retrieving \( k \)OGNN for aggregate function \( \text{MAX} \). Algorithm \( CBQM_{\text{SUM}} \) uses Euclidean nearest neighbor of the centroid \( c_Q \) of the query points \( Q \) to find out the obstructed group nearest neighbor between a data point \( p \) and the query points \( Q \).

The input of the algorithm is a set of query points \( Q \), an obstacle RTree \( RT_O \) and a data point RTree \( RT_P \). The output of the algorithm is a set of \( k \)-obstructed group nearest neighbors of \( Q \).

In this approach the algorithm at first finds the Euclidean nearest neighbor \( p \) of centroid \( c_Q \). Next,
the algorithm finds the aggregate obstructed distance $\text{dist}_{AO}(p, Q)$ between the data point $p$ and the query points $Q$. The aggregate obstructed distance calculation strategy is described in Chapter 3.

Then, the algorithm incrementally retrieves the next Euclidean nearest neighbor with respect to the centroid and calculate their aggregate obstructed distance. We demonstrate the searching of the subsequent Euclidean nearest neighbors with a function $\text{findNextNN}_E(Q, RT_P, k)$ in Algorithm 4. The searching for Euclidean nearest data point with respect to the centroid is continued until the actual obstructed group nearest neighbor has been found.

We consider only those data points that are within the intersection of such circle that are within radius $r$, where $r=d_{kmax}$ (Lines 11-14). If we consider the intersection area as $C$ then,

- $C$ : Intersection of the circles centering $q_i$ with radius $d_{kmax}$

  \[ C = (C_1 \cap C_2 \cap C_3 \ldots \cap C_n) \]

The while loop (Lines 8-14) terminates when the Euclidean distance $\text{dist}_E(p, c_Q)$ for the $k^{th}$ data point $p$ is greater than our current OGNN’s aggregate obstructed distance.

Algorithm 4 illustrates the complete algorithm for finding $k(\geq 1)$ obstructed group nearest neighbors for a set of query points $Q$. The $k$ Euclidean nearest neighbors of the centroid $c_Q$ are first obtained using $RT_P$ and a function $\text{findkNN}_E(c_Q, RT_P, k)$. Then the algorithm calculates the aggregate obstructed distance of all the candidate points (Lines 1-5) and store the data points in the result set $A$.

Initially $d_{kmax}$ stores the $k^{th}$ data point aggregate obstructed distance, which is currently the farthest data point (Line 6).

Then, the algorithm incrementally retrieves the next Euclidean nearest data point with respect to centroid $c_Q$ using a function $\text{findNextNN}_E(c_Q, RT_P)$ (Line 9) until the actual obstructed group nearest neighbor has been found and update the result set $A$ and the best aggregate obstructed distance $d_{kmax}$ (Lines 8-14).

The while loop terminates when the Euclidean distance $\text{dist}_E(p, c_Q)$ for the $k^{th}$ data point $p$ is greater than our current OGNN’s average of aggregate obstructed distance and returns the result set $A$ containing the $k$OGNNs (Line 15).
Algorithm 4: CBQM\_MAX (Q, p, k, RT\_O, RT\_P)

**Input:** A set of query points Q = \{q_1, q_2, \ldots, q_n\}, an obstacle RTree RT\_O, a data point RTree RT\_P

**Output:** A, a set of k-obstructed group nearest neighbors of Q

1. Initialize (LVG)
2. \( P_E \leftarrow \text{findkNN}(c_Q, RT_P, k) \)
3. foreach \( p \in P_E \) do
4. \( \text{dist}_{AO}(p, Q) \leftarrow \text{CompAggObsDist}(Q, p, RT_O, LVG) \)
5. \( A \leftarrow P_E \)
6. \( d_{kmax} \leftarrow \max_{1 \leq i \leq |P_E|} \text{dist}_{AO}(p_i, Q) \)
7. \( d_{E_{max}} \leftarrow \max_{1 \leq i \leq |P_E|} \text{dist}_{E}(p_i, c_Q) \)
8. \( d_{max} \leftarrow \max_{1 \leq i \leq |Q|} \text{dist}_{E}(q_i, c_Q) \)
9. while \( d_{E_{max}} < (d_{max} + d_{kmax}) \) do
10. \( p \leftarrow \text{findNextNN}(c_Q, RT_P) \)
11. \( d_{E_{max}} \leftarrow \text{dist}_{E}(p, c_Q) \)
12. if \( p \in \bigcap_{j=1}^n C_j \) then
13. \( \text{dist}_{AO}(p, Q) \leftarrow \text{CompAggObsDist}(Q, p, RT_O, LVG) \)
14. if \( \text{dist}_{AO}(p, Q) < d_{kmax} \) then
15. \( \text{update}(A, d_{kmax}, p, \text{dist}_{AO}(p, Q)) \)
16. return A

<table>
<thead>
<tr>
<th>Data Point</th>
<th>( \text{dist}_E(p, c_Q) )</th>
<th>( \text{dist}_{AO}(p, Q) )</th>
<th>( d_{kmax} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( p_1 )</td>
<td>5</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>( p_2 )</td>
<td>7</td>
<td>-</td>
<td>10</td>
</tr>
<tr>
<td>( p_3 )</td>
<td>9</td>
<td>15</td>
<td>10</td>
</tr>
<tr>
<td>( p_4 )</td>
<td>25</td>
<td>-</td>
<td>10</td>
</tr>
</tbody>
</table>
CHAPTER 4. OBSTRUCTED GROUP NEAREST NEIGHBOR QUERIES

We illustrate Algorithm 4 with the example depicted in Figures 4.2a and 4.2b, where the data point set \( P = \{p_1, p_2, p_3, p_4\} \) and query point set \( Q = \{q_1, q_2, q_3\} \). The centroid \( c_Q \) of the query points is shown in the figure. The obstacle set \( O \) is not shown in the figure for simplicity.

The Algorithm at first retrieves the first nearest neighbor \( p_1 \) at distance 5 from \( c_Q \) and calculate \( p_1 \)'s aggregate obstructed distance \( \text{dist}_{AO}(p_1, Q)=10 \) using one of the algorithms described in Section 3.3 and updates \( d_{k_{\text{max}}}=10 \). For the first iteration of the while loop (Line 8) the algorithm finds the intersection area of the three circles centering \( q_1, q_2 \) and \( q_3 \) with radius \( d_{k_{\text{max}}} \). For aggregate function \( \text{max} \) we consider only those data points as candidate points that are within the intersection area of the three circles.

The Algorithm then incrementally retrieves the subsequent nearest neighbors \( p_2, p_3, p_4 \) etc. Since, \( p_2 \) does not reside in the intersection area of the three circle so the algorithm omits it. We can see from Figure 4.2b that \( p_3 \) is a candidate OGNN for aggregate function \( \text{max} \) that’s why the algorithm calculates the aggregate obstructed distance \( \text{dist}_{AO}(p_3, Q)=15 \). The steps are shown in Table 4.3. The Algorithm terminates and returns \( p_1 \) as OGNN when the 4th nearest neighbor \( p_4 \) of \( c_Q \) is retrieved as it’s Euclidean distance 25 is greater than \( d_{k_{\text{max}}} \) 10 and the condition at Line 8 is false.

**Figure 4.2: Example of aggregate function \( \text{max} \) for CBQM**
4.2.1 Correctness of Algorithm CBQM_MAX

**Theorem 4.2.1** If \( k \) is the number of required data points for a \( k \)-OGNN query with respect to a set of query points \( Q = \{q_1, q_2, \ldots, q_n\} \) for \( 1 \leq i \leq n \), then the answer set \( A \) from Algorithm 4 (Algorithm CBQM_MAX) includes \( k \) data points that have the smallest \( k \) aggregate obstructed distance for the aggregate function \( \text{max} \) among all the data points in the data point Rtree \( R_T_P \).

**Proof** (By contradiction)
Let, \( p' \) be a data point that is not in \( A \) but has the \( j^{\text{th}} \) minimum value \( (1 \leq j \leq k) \) for aggregate obstructed distance \( \text{dist}_{AO}(p, Q) \) for \( \text{max} \) from \( p \) with respect to a group of query points \( Q = \{q_1, q_2, \ldots, q_n\} \).
Algorithm \( \text{CBQM}_\text{MAX} \) incrementally retrieves the Euclidean nearest neighbor \( p \) of centroid \( c_Q \) and then compute aggregate obstructed distance \( d = \text{dist}_{AO}(p, Q) \) for those data points that lies within the intersection of the circles \( C = (C_1 \cap C_2 \cap C_3 \ldots \cap C_n) \) centering each of the query points \( \forall i \in \{1, \ldots, n\} (q_i) \in Q \) with radius \( d \), where \( d \) is the maximum aggregate obstructed distance \( d = \max_{1 \leq i \leq n} \text{dist}_O(p, q_i) \) of \( Q \), until the Euclidean distance \( \text{dist}_E(p, c_Q) \) is greater than the current aggregate obstructed distance \( d \) for the best data point \( p_r \) found so far.
In [4] Namnandorj and Chen proves that, the group Euclidean nearest neighbor (\( \text{max} \)) of \( Q \) lies within the intersection of the circles \( C \) centering each of \( \forall i \in \{1, \ldots, n\} \) with radius \( r \), where \( r = \text{aggregate Euclidean distance of } Q \).
Since \( p' \) is not retrieved by Algorithm \( \text{CBQM}_\text{MAX} \), it is outside of the intersection area \( C \). Thus, for at least one \( q_i \), \( p' \)'s maximum distance is greater than current aggregate obstructed distance \( d \) which contradicts our assumption.
\[
\exists i \in \{1, \ldots, n\} (q_i) \in Q (\text{dist}_O(p', q_i) \geq \text{dist}_O(p_r, q_i))
\]
Thus, Algorithm \( \text{CBQM}_\text{MAX} \) finds the actual obstructed group nearest neighbor for a set of query points \( Q \) for the function \( \text{max} \).

\[\square\]

4.2.2 Time Complexity of Algorithm CBQM_MAX

The worst case time complexity of Algorithm 4 (Algorithm CBQM_MAX) when SPAOD (Algorithm 1) aggregate obstructed distance calculation is used is: \( O(|P||T||Q||O||V|^3 \log |V|) \).
And the worst case time complexity of Algorithm 4 (Algorithm CBQM\_MAX) when MPAOD (Algorithm 2) aggregate obstructed distance calculation is used is: $O(|P||O||Q|^2|V|^2 \log |V|)$

where,

- $|P|$ is the number of data points,
- $|T|$ is the number of times the threshold $t$ will be updated,
- $|Q|$ is the number of query points,
- $|O|$ is the number of obstacle vertices, and
- $|V|$ is the number of nodes in the visibility graph.

### 4.3 Group Based Query Method (GBQM)

In this section we describe Algorithm GBQM. Algorithm GBQM (Algorithm 5) uses Euclidean GNN to find out the obstructed group nearest neighbor between a data point $p$ and multiple query points $Q$. The input of the algorithm is a set of query points $Q$, an obstacle RTree $RT_O$ and a data point RTree $RT_P$. The output of the algorithm is a set of $k$-obstructed group nearest neighbors of $Q$.

Following are a list of symbols used in this algorithm:

- $\text{find}k\text{GNN}_E(Q, RT_P, k)$: Finds $k$ Euclidean GNN for a set of query points $Q$ with respect to a data point $p$ from RTree $RT_P$.
- $\text{findNextGNN}_E(Q, RT_P)$: Incrementally retrieves the next Euclidean GNN for a set of query points $Q$ with respect to a data point $p$ from RTree $RT_P$.
- $\max_{1 \leq i \leq n} \text{dist}_{AE}(p_i, Q)$: Maximum distance of all the Euclidean aggregate distances between the data points $\forall_{i=1}^n p_i$ and a set of query points $Q$.

In this approach the algorithm at first finds the Euclidean GNN $p$ with group Euclidean distance $\text{dist}_{AE}(p, Q)$.

$$\text{dist}_{AE}(p, Q) = f_{i=1}^n \text{dist}_E(p, q_i),$$

where $q_i$ is the $i$th query point and $f$ is the aggregate function (4.1)
Next, the algorithm finds the aggregate obstructed distance $\text{dist}_{AO}(p, Q)$ between the data point $p$ and the query points $Q$ (Lines 3-4). The aggregate obstructed distance calculation strategy is described in Chapter 3. If the aggregate obstructed distance and aggregate Euclidean distance is equal for the first Euclidean GNN ($p$), then we can say that there are no obstacles in between the data point $p$ and the set of query points $Q$. So, we conclude that we have found our obstructed group nearest neighbor and return the result (Lines 7-8).

If the two distances: aggregate obstructed distance and aggregate Euclidean distance varies that is, there are obstacles in between the data point $p$ and the set of query points $Q$ then the algorithm search for the next Euclidean GNN and calculate it’s aggregate obstructed distance. We demonstrate the searching of the subsequent Euclidean GNNs with a function $\text{findNextGNN}_E(Q, RT_P, k)$ in Algorithm 5 (Line 10).

The Algorithm incrementally retrieves the next Euclidean group nearest data point until the actual obstructed group nearest neighbor has been found and update the result set $A$ and the best aggregate obstructed distance $d_{kmax}$, with a function $\text{update}(A, d_{kmax}, p, \text{dist}_{AO}(p, Q))$ (Lines 9-14).

The repeat until loop (Lines 9-14) terminates when the Euclidean group distance $\text{dist}_{AE}(p, Q)$ for the $k^{th}$ data point $p_k$ is greater than our current OGNN’s aggregate obstructed distance $d_{kmax}$. Finally, the Algorithm returns the answer set $A$ containing $k$ OGNNs (Line 15).

Algorithm 5 illustrates the complete algorithm for finding $k (\geq 1)$ obstructed group nearest neighbors for a set of query points $Q$. The $k$ Euclidean GNNs are first obtained using $RT_P$ using a function $\text{findkGNN}_E(Q, RT_P, k)$ and stored in the result set $A$. Then the algorithm calculates the aggregate obstructed distance of all the candidate points just retrieved (Lines 1-5).

Initially $d_{kmax}$ stores the $k^{th}$ data point’s aggregate obstructed distance, which is currently the farthest data point (Line 6). To test whether we have already found our candidate $k$ obstructed group nearest neighbor (Lines 7-8) we compare $d_{kmax}$ with the maximum Euclidean aggregate distance for the $k$ Euclidean GNNs retrieved.

Otherwise, the algorithm incrementally retrieves the next Euclidean group nearest data point using a function $\text{findNextGNN}_E(Q, RT_P)$ (Line 10) until the actual obstructed group nearest neighbor has been found and update the result set $A$ and the best aggregate obstructed distance $d_{kmax}$ (Lines 9-13).
Algorithm 5: GBQM \((Q, p, k, RT_O, RT_P)\)

**Input:** A set of query points \(Q = \{q_1, q_2, \ldots, q_n\}\), an obstacle RTree \(RT_O\), a data point RTree \(RT_P\)

**Output:** \(A\), \(k\)-obstructed group nearest neighbors of a set of query points \(Q\)

1. \(P_{EG} \leftarrow \text{findKGNNE}(Q, RT_P, k)\)
2. Initialize \((LVG)\)
3. \(\textbf{foreach} \ p \in P_{EG} \textbf{ do}\)
   4. \(\text{dist}_{AO}(p, Q) \leftarrow \text{CompAggObsDist}(Q, p, RT_O, LVG)\)
5. \(A \leftarrow P_{EG}\)
6. \(d_{kmax} \leftarrow \max_{1 \leq i \leq |P_{EG}|} \text{dist}_{AO}(p_i, Q)\)
7. \(\textbf{if} \ \max_{1 \leq i \leq |P_{EG}|} \text{dist}_{AE}(p_i, Q) \neq d_{kmax} \textbf{ then}\)
   8. \(\text{return} \ A\)
9. \(\textbf{repeat}\)
10. \(p \leftarrow \text{findNextGNN}(Q, RT_P)\)
11. \(\text{dist}_{AO}(p, Q) \leftarrow \text{CompAggObsDist}(Q, p, RT_O, LVG)\)
12. \(\textbf{if} \ \text{dist}_{AO}(p, Q) < d_{kmax} \textbf{ then}\)
13. \(\text{update}(A, d_{kmax}, p, \text{dist}_{AO}(p, Q))\)
14. \(\textbf{until} \ \text{dist}_{AE}(p, Q) \geq d_{kmax}\)
15. \(\text{return} \ A\)
CHAPTER 4. OBSTRUCTED GROUP NEAREST NEIGHBOR QUERIES

The Algorithm terminates when the Euclidean group distance $dist_{AE}(p, Q)$ for the next data point $p_k$ is greater than our current OGNN’s aggregate obstructed distance $d_{k\text{max}}$.

Figure 4.3: Example of GBQM

Table 4.4: GBQM for sum Example

<table>
<thead>
<tr>
<th>Data Point</th>
<th>$dist_{AE}(p, Q)$</th>
<th>$dist_{AO}(p, Q)$</th>
<th>$d_{k\text{max}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p_1$</td>
<td>10</td>
<td>18</td>
<td>18</td>
</tr>
<tr>
<td>$p_2$</td>
<td>14</td>
<td>19</td>
<td>18</td>
</tr>
<tr>
<td>$p_3$</td>
<td>16</td>
<td>17</td>
<td>17</td>
</tr>
<tr>
<td>$p_4$</td>
<td>19</td>
<td>-</td>
<td>17</td>
</tr>
</tbody>
</table>

We illustrate Algorithm 5 with the example depicted in Figure 4.3, where the data point set $P = \{p_1, p_2, p_3, p_4\}$, query point set $Q = \{q_1, q_2, q_3\}$ and obstacle set $O = \{o_1, o_2, o_3, o_4, o_6\}$. The Algorithm at first retrieves the first Euclidean GNN $p_1$ of $Q$ and calculate $p_1$’s aggregate obstructed distance $dist_{AO}(p_1, Q)=18$ using one of the algorithms described in Section 3.3. The algorithm sets the current $d_{k\text{max}} = 18$. The Algorithm then incrementally retrieves the subsequent group nearest neighbors $p_2, p_3, p_4$ etc of $Q$. 

a) Retrieving NN’s with respect to $Q$
b) Computing $p_3$’s aggregate obstructed distance
Table 4.4 shows the corresponding steps. Since, \( p_2 \) and \( p_3 \) gives an aggregate Euclidean distance lower than \( d_{k_{\text{max}}} = 18 \), so the algorithm calculates there aggregate obstructed distance 19 and 17, respectively. As \( p_3 \)'s aggregate obstructed distance is lower than current \( d_{k_{\text{max}}} \) so the algorithm updates the \( d_{k_{\text{max}}} = 17 \). The algorithm terminates after retrieving \( p_4 \), as it’s aggregate Euclidean distance is greater than our current \( d_{k_{\text{max}}} \) and returns \( p_3 \) as the OGNN.

### 4.3.1 Correctness of Algorithm GBQM

**Theorem 4.3.1** Let \( Q \) be a set of query points and \( p_1, p_2, p_3, \ldots, p_n \) be the data points in RTree \( RT_P \) sorted according to their Euclidean group distance \( \text{dist}_{AE} \) such that \( \text{dist}_{AE}(p_1, Q) \leq \text{dist}_{AE}(p_2, Q) \leq \ldots \leq \text{dist}_{AE}(p_n, Q) \). Let, \( c_1, c_2, \ldots, c_k \) are the \( k \) candidate points with smallest group obstructed distance to \( Q \) among the first \( m \) points \( p_1, p_2, \ldots, p_m \) in \( RT_P \). If \( \text{dist}_{AE}(p_m, Q) \geq \max_{1 \leq i \leq k} \text{dist}_{AO}(c_i, Q) \), then \( c_1, c_2, \ldots, c_k \) are the \( k \) obstructed group nearest neighbors of \( Q \) among all points in \( RT_P \).

**Proof** (By contradiction)

Let, \( p' \) be a data point that is not in \( A \) but has the \( j^{\text{th}} \) minimum value \( (1 \leq j \leq k) \) for aggregate obstructed distance \( \text{dist}_{AO}(p, Q) \) from \( p \) with respect to a group of query points \( Q = (q_1, q_2, \ldots, q_n) \).

Algorithm GBQM incrementally retrieves the first \( m \) Euclidean GNNs \( p_1, p_2, \ldots, p_m \) of \( Q \) and choose \( k \) candidate obstructed group nearest neighbors \( c_1, c_2, \ldots, c_k \). We terminate Algorithm GBQM when \( \text{dist}_{AE}(p_m, Q) \geq \max_{1 \leq i \leq k} \text{dist}_{AO}(c_i, Q) \). Since \( p' \) is not retrieved by Algorithm GBQM so it’s Euclidean aggregate distance \( \text{dist}_{AE}(p, Q) \) is greater than \( \forall_{1 \leq i \leq k} \text{dist}_{AO}(c_i, Q) \).

According to Euclidean lower bound property, for a data point \( p_i \), \( \text{dist}_{AO}(p_i, Q) \geq \text{dist}_{AE}(p_i, Q) \). Since \( p' \)'s Euclidean aggregate distance is greater than all the points retrieved by Algorithm GBQM, so it’s aggregate obstructed distance is also greater, which contradicts our assumption.

Thus, Algorithm GBQM finds the actual obstructed group nearest neighbor for a set of query points \( Q \).

\[ \Box \]

### 4.3.2 Time Complexity of Algorithm GBQM

The worst case time complexity of Algorithm 5 (Algorithm CBQM_sum) when SPAOD (Algorithm 1) aggregate obstructed distance calculation is used is: \( O(|P||T||Q||O||V|^3 \log |V|) \).
And the worst case time complexity of Algorithm 5 (Algorithm CBQM_sum) when MPAOD (Algorithm 2) aggregate obstructed distance calculation is used is: \( O(|P||Q|^2|V|^2 \log |V|) \)

where,

- \(|P|\) is the number of data points,
- \(|T|\) is the number of times the threshold \( t \) will be updated,
- \(|Q|\) is the number of query points,
- \(|O|\) is the number of obstacle vertices, and
- \(|V|\) is the number of nodes in the visibility graph.
Chapter 5

Experiments

In this chapter, we evaluate the performance of our proposed algorithms through extensive experiments. We vary the group size, number of nearest neighbor retrieved and the area of the query rectangles. We also vary the cardinalities of the synthetic dataset i.e. we control the density of the data point Rtree ranging from $0.01|RT_O|$ to $10|RT_O|$ with respect to the obstacle Rtree $RT_O$. We experimentally evaluate the CPU time and IO cost of the proposed algorithms, using an Intel Core i7-2920XM quad-core CPU (2.50GHz) PC with 16GB RAM. We used an Rtree with default node size of 1KB.

We use both real and synthetic datasets in our experiments. The dataspace is normalized into an area of $10,000 \times 10,000$ square units. We use the real dataset of Germany [55] which consists of 30674 MBRs of railway lines (rrlines) and 76999 MBRs of hypsography data (hypsogr). We generate synthetic data sets $U$ and $Z$ using a uniform and Zipfian distribution respectively, and we vary the density of the data point Rtree ranging from $0.01|RT_O|$ to $10|RT_O|$ with respect to the obstacle Rtree $RT_O$.

Default data points are respectively end points of hypsogr and obstacles are rrlines in Germany. In this way, data points and obstacles are adjacently deployed into the same geographical space and thus simulate our scenario. The distribution of the entity data points follows the obstacle distribution. The entity data points are allowed to lie on the boundary of the obstacles but not inside the obstacles. Though we have used minimum bounded rectangle our methods support any arbitrary shaped polygons.

We consider 100 OGNN queries for each of the CBQM and GBQM based approaches and calculate the
average experimental results. These query points also follow obstacle distribution. Table 5.1 displays the experimental setup for our experiments.

Table 5.1: Experimental Setup

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Range</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>Group size</td>
<td>2, 4, 8, 16, 32</td>
<td>8</td>
</tr>
<tr>
<td>$k$</td>
<td>2, 4, 8, 16, 32</td>
<td>4</td>
</tr>
<tr>
<td>Query rectangle area</td>
<td>.002% to .01%</td>
<td>.005%</td>
</tr>
<tr>
<td>Cardinalities</td>
<td>$0.01</td>
<td>O</td>
</tr>
</tbody>
</table>

5.1 Comparison of OGNN and Obstructed Distance Computation Algorithms

We evaluate and compare the single point based approach $SPAOD$ computation and the multi point based approach $MPAOD$ computation for aggregate obstructed distance computation. We also evaluate the Group Based Query Method (GBQM) and Centroid Based Query Method (CBQM) for OGNN queries. We run experiments for GBQM (SPAOD), GBQM (MPAOD), CBQM (SPAOD) and CBQM (MPAOD). All the comparisons are performed in terms of IO cost and computational time. We compare the IO cost for both the obstacle Rtree and the data point Rtree separately.

To differentiate combinations of OGNN queries and aggregate obstructed distance computation algorithms in the graph we use some notations, i.e, GBQM (SPAOD) denotes the experimental result of Group Based Query Method (GBQM) where the Single Point Aggregate Obstructed Distance (SPAOD) computation is used for the aggregate obstructed distance computation. Other three notations : GBQM (MPAOD), CBQM (SPAOD) and CBQM (MPAOD) also follow corresponding meanings.

We present the experimental results for varying group size, $k$, query area and cardinality ratio in Sections 5.1.1 to 5.1.4 using the real Germany dataset, the $U$ dataset and the $Z$ dataset and then analyze the result in Section 5.2.
5.1.1 Effect of Group Size

We study the impact of group size on the performance of OGNN query by varying the group size using 2, 4, 8, 16 and 32 and measuring the required running time and number of data point access from the data point RTree and number of obstacle access from the obstacle RTree.

5.1.1.1 Aggregate function sum

Experiments on Real Germany Dataset

Figure 5.1a shows the time required by GBQM and CBQM in combination with the SPAOD and MPAOD for the aggregate function \texttt{sum}. The Figure shows that the performance of all the algorithms degrade as the group size increases. We observe that, for both GBQM and CBQM, SPAOD performs better than MPAOD. For both SPAOD and MPAOD, GBQM performs better than CBQM.

Figure 5.1b and 5.1c respectively shows the IO access of the data point RTree and the obstacle RTree for the aggregate function \texttt{sum}. For all the algorithms both data point and obstacle RTree access significantly increases with the increase in group size. For example, for an increase of the group size from 4 to 32, the number of data point access increases approximately 1.83 times and the number of obstacle access increases approximately 1.4 times for all the algorithms. The reason behind such behavior: we expect that the increase in the group size increases the number of aggregate obstructed distance computations and hence increases more obstacle retrieval from the obstacle RTree.

For both data point and obstacle RTree, GBQM and CBQM algorithms with combination of SPAOD
gives lower IO access than MPAOD. For both SPAOD and MPAOD, GBQM gives lower IO access than CBQM. For example, for group size 8, GBQM (SPAOD) combination gives almost 2 times lower data point Rtree access than GBQM (MPAOD).

Experiments on U Dataset
The experimental results for varying group size for the aggregate function sum in U dataset shows a similar trend as aggregate function sum in real dataset. Figure 5.2a, Figure 5.2b and 5.2c shows the corresponding figures.

Experiments on Z Dataset
The experimental results for varying group size for the aggregate function sum in Z dataset shows a similar trend as aggregate function sum in real dataset. Figure 5.3a, Figure 5.3b and 5.3c shows the corresponding figures.
Figure 5.4: Effect of group size on \( \text{MAX} \) (Germany dataset)

### 5.1.1.2 Aggregate function \( \text{MAX} \)

**Experiments on Real Germany Dataset**

Figure 5.4a, shows the time required by GBQM and CBQM obstructed group nearest neighbor queries in combination with the SPAOD and MPAOD obstructed distance calculation algorithms for the aggregate function \( \text{MAX} \). For \( \text{MAX} \), the performance of both the OGNN algorithms in combination with the aggregate obstructed distance calculation shows almost similar behavior. For higher values of group size, the difference between the algorithms is less than 10 sec.

Figure 5.4b and 5.4c respectively shows the IO access of the data point Rtree and the obstacle Rtree for the aggregate function \( \text{MAX} \). For both data point and obstacle Rtree, GBQM in combination with SPAOD gives significantly lower IO access than other combinations. For both SPAOD and MPAOD, GBQM gives lower IO access than CBQM. For example, for group size 32, GBQM (SPAOD) combination retrieves 40 obstacle from the obstacle Rtree whereas GBQM (MPAOD) retrieves almost 55 obstacles.

**Experiments on \( U \) Dataset**

The experimental results for varying group size for the aggregate function \( \text{MAX} \) in \( U \) dataset shows a similar trend as aggregate function \( \text{MAX} \) in real dataset. Figure 5.5a, Figure 5.5b and 5.5c shows the corresponding figures.

**Experiments on \( Z \) Dataset**

The experimental results for varying group size for the aggregate function \( \text{MAX} \) in \( Z \) dataset shows a similar trend as aggregate function \( \text{MAX} \) in real dataset. Figure 5.6a, Figure 5.6b and 5.6c shows the
Figure 5.5: Effect of group size on \( \text{max} \) (\( U \) dataset)

Figure 5.6: Effect of group size on \( \text{max} \) (\( Z \) dataset)

corresponding figures.

5.1.2 Effect of \( k \)

In this set of experiments, we evaluate the impact of \( k \) on the performance of OGNN algorithms by varying \( k \) from 2 to 32 (2, 4, 8, 16, 32).

5.1.2.1 Aggregate function sum

Experiments on Real Germany Dataset

The computation time and RTree access increases quite fast as \( k \) increases for all the algorithms. For example, the increase of \( k \) from 4 to 8 increases the computation time almost 2 times. Figure 5.7b shows that, for an increase of \( k \) from 2 to 4 the number of data point access increases approximately 1.8
times and Figure 5.7c shows, for an increase of $k$ from 8 to 16 the number of obstacle access increases approximately 1.18 times. The reason behind such increase, as $k$ increases, number of candidate data points also increase, as a result more data points are retrieved from the data point RTree. Increased candidate data points invoke more aggregate obstructed distance computation which as a consequence increases the IO of obstacle RTree.

We see in Figure 5.7a that the time required by CBQM is always almost 2.5 times higher than the GBQM algorithm in both the combination of SPAOD and MPAOD for every values of $k$ for $\text{sum}$. Figure 5.7b shows that for $\text{sum}$ the IO access of data point Rtree of CBQM is approximately 2.0 times higher than that of GBQM. From the curves we can also find that, the GBQM (SPAOD) and CBQM (MPAOD) does not show a constant increasing/decreasing trend. We observe in Figure 5.7c that, GBQM performs better than CBQM and SPAOD performs better than MPAOD for all the values of $k$.

**Experiments on $U$ Dataset**

The experimental results for varying $k$ for the aggregate function $\text{sum}$ in $U$ dataset shows a similar trend as aggregate function $\text{sum}$ in real dataset. Figure 5.8a, Figure 5.8b and 5.8c shows the corresponding figures.

**Experiments on $Z$ Dataset**

The experimental results for varying $k$ for the aggregate function $\text{sum}$ in $Z$ dataset shows a similar trend as aggregate function $\text{sum}$ in real dataset. Figure 5.9a, Figure 5.9b and 5.9c shows the corresponding figures.
5.1.2.2 Aggregate function max

Experiments on Real Germany Dataset

We see in Figure 5.10a that the time required by CBQM is always almost 2.5 times higher than the GBQM algorithm in both the combination of SPAOD and MPAOD for every values of \( k \) for sum. Among the aggregate obstructed distance calculation algorithms, SPAOD performs almost 2.5 times better than the MPAOD for both the combinations of GBQM and CBQM. The algorithms performance degrades with an increase in \( k \) for all the algorithms which is a similar behavior like sum. We observe in Figure 5.10b and Figure 5.10c that, none of the combinations shows any specific increasing/decreasing trend. Though, on an average GBQM performs better than CBQM.

Experiments on U Dataset

The experimental results for varying \( k \) for the aggregate function max in U dataset shows a similar
## 5.1.3 Effect of Query Area

In this set of experiment, we vary the query area, i.e., the area to which the group of query objects are confined to as 0.002%, 0.004%, 0.006%, 0.008% and 0.01% of the entire data space.
CHAPTER 5. EXPERIMENTS

63

1.1

4

8

12

16

2  4  8  16  32

Time (sec)

k

GBQM(SPAOD)

CBQM(SPAOD)

GBQM(MPAOD)

CBQM(MPAOD)

Figure 5.12: Effect of $k$ on max $(Z$ dataset)

a) CPU time

b) Data point IO

c) Obstacle IO

5.1.3.1 Aggregate function sum

Experiments on Real Germany Dataset

The effect of varying query area is quite significant in sum. Here, we see that the processing time and RTree access increases with the increase of the query space for all the proposed methods. Figure 5.13a shows that, the times for MPAOD in combination with both GBQM and CBQM remains almost constant for different percentage of query area. However, SPAOD aggregate obstructed distance calculation strategy shows a clear increasing trend with the increase of query area.

We see in Figure 5.13b and 5.13c the curves for IO access of the data point and obstacle Rtree. Both the data point RTree and obstacle RTree access increases quite fast with the query area. For example, with an increase of query area from 0.004% to .006% the data point RTree access increases almost 1.5 times and obstacle RTree access increases twice. With obstacle RTree all the combinations shows an
increasing IO access and GBQM performs better than CBQM with both the combination of SPAOD and MPAOD. However, with the increase of query area obstacle Rtree IO access increases quite fast for all the combinations.

Experiments on U Dataset
The experimental results for varying query area for the aggregate function sum in U dataset shows a similar trend as aggregate function sum in real dataset. Figure 5.14a, Figure 5.14b and 5.14c shows the corresponding figures.

Experiments on Z Dataset
The experimental results for varying query area for the aggregate function sum in Z dataset shows a similar trend as aggregate function sum in real dataset. Figure 5.15a, Figure 5.15b and 5.15c shows the corresponding figures.
5.1.3.2 Aggregate function \texttt{max}

Experiments on Real Germany Dataset

The experimental results for varying query area for the aggregate function \texttt{max} shows a similar trend as aggregate function \texttt{sum}. Figure 5.16a shows the CPU time for \texttt{max} and Figure 5.16b and 5.16c respectively shows the IO access for the data point Rtree and the obstacle Rtree for the aggregate function \texttt{max}.

Experiments on \textit{U} Dataset

The experimental results for varying query area for the aggregate function \texttt{max} in \textit{U} dataset shows a similar trend as aggregate function \texttt{max} in real dataset. Figure 5.17a, Figure 5.17b and 5.17c shows the corresponding figures.

Experiments on \textit{Z} Dataset
CHAPTER 5. EXPERIMENTS

The experimental results for varying query area for the aggregate function \( \text{max} \) in \( Z \) dataset shows a similar trend as aggregate function \( \text{max} \) in real dataset. Figure 5.18a, Figure 5.18b and 5.18c shows the corresponding figures.

5.1.4 Effect of Cardinality Ratio

We study the impact of cardinality ratio as functions of \( |P|/|O| \) (i.e., the ratio of data points to obstacle dataset cardinalities) on the performance of OGNN query by varying the cardinality using 0.1, 0.5, 1, 2 and 10.

5.1.4.1 Aggregate function sum

Experiments on \( U \) Dataset

Figure 5.19b illustrates the costs of the OGNN algorithms as a function of the ratio \( |P|/|O| \) on dataset \( U \). The IO accesses of the data point R-tree remains almost constant with \( |P|/|O| \) because, as the density increases, the range around the set of query points where the Euclidean neighbors or Euclidean GNN are found decreases. As a result Figure 5.19c shows that the number of obstacle retrieval in the aggregate obstructed distance computations also decreases.

Figure 5.19a provides the observation that the CPU time decreases with the data points density. The reason behind this is, with the increase of cardinality obstacle retrieval decreases as a consequences CPU time also decreases.

However all the combinations of obstructed group nearest neighbor queries and aggregate obstructed

Figure 5.18: Effect of query area on \( \text{max} \) (\( Z \) dataset)
distance calculation shows almost similar performance with the cardinality ratio change. The effect of varying cardinality ratio is not significant for \textit{SUM}.

**Experiments on Z Dataset**

The experimental results for varying cardinality ratio on dataset \textit{Z} for the aggregate function \textit{SUM} shows that GBQM performs almost 1.5 times faster then CBQM algorithms. Figure 5.20a shows the CPU time for \textit{SUM} and Figure 5.20b and 5.20c respectively shows the IO access for the data point Rtree and the obstacle Rtree for the aggregate function \textit{SUM}.

### 5.1.4.2 Aggregate function \textit{max}

**Experiments on U Dataset**

The experimental results for varying cardinality ratio on dataset \textit{U} for the aggregate function \textit{MAX}
shows a similar trend as aggregate function sum. Figure 5.21a shows the CPU time for MAX and Figure 5.21b and 5.21c respectively shows the IO access for the data point Rtree and the obstacle Rtree for the aggregate function MAX.

**Experiments on Z Dataset**

The experimental results for varying cardinality ratio on dataset Z for the aggregate function MAX shows similar pattern as aggregate function SUM. Figure 5.22a shows the CPU time for SUM and Figure 5.22b and 5.22c respectively shows the IO access for the data point Rtree and the obstacle Rtree for the aggregate function SUM.
5.2 Comparative Analysis

In all experiments, the CPU time and IO access of GBQM is always lower than that of CBQM for both sum and max. For the aggregate obstructed distance calculation, we observe that the CPU time and IO access of both the data point Rtree and obstacle Rtree of SPAOD is lower than that of MPAOD in most of the cases.

The reason behind lower cost of GBQM is that, GBQM incrementally retrieves the Euclidean GNN of the query point set \( Q \). The probability of the Euclidean GNN being the obstructed group nearest neighbor is much higher. Whereas, CBQM incrementally retrieves the nearest neighbor of centroid which incurs more data point to be retrieved from the data point Rtree. The increase of more candidate data point incurs more aggregate obstructed distance calculation. As a result, the IO access of obstacle Rtree also increases. For this reason, GBQM performs better than CBQM for both sum and max in terms of CPU time and IO cost.

On the other hand, the reason behind better performance of SPAOD with respect to MPAOD is that, in SPAOD we incrementally retrieve the obstacles from the centroid of the query points and no obstacle is retrieved more than once. As a result, the IO access of the obstacle Rtree is much lower than that of MPAOD. In case of MPAOD, a new candidate data point may invoke same obstacle retrieval multiple times, which increases the IO cost. The increase of IO also increases the CPU time as more obstacle is retrieved from the obstacle R-Trre. Another reason behind lower CPU time of SPAOD is, SPAOD reuse some previously computed shortest path distance, which lowers the CPU time than that of MPAOD.
Chapter 6

Conclusion

Algorithms have been developed for processing spatial queries in an obstructed space that includes, obstructed nearest neighbor (ONN) queries ([9–11]), continuous obstructed nearest neighbor (CONN) queries ([32, 33]), moving $k$ nearest neighbor (MNN) queries ([13, 34]) and obstructed reverse nearest neighbor (ORNN) queries ([12]). However, none of them considered a group of pedestrians and their travel path, which may include blocking obstacles such as lakes and moving vehicles on the road. In this thesis, we developed the first comprehensive solution to find the $k$ group nearest neighbors in presence of obstacles which we call $k$ obstructed group nearest neighbor queries ($k$OGNN).

We proposed two approaches: Centroid Based Query Method (CBQM) and Group Based Query Method (GBQM) for answering OGNN queries. CBQM is based on the Euclidean nearest neighbor of the centroid of the query points and GBQM is based on the Euclidean GNN of the query points. The first approach $CBQM$ incrementally finds the Euclidean nearest neighbor with respect to the centroid of the query points until the best $k$ OGNNs have been found. GBQM incrementally retrieves Euclidean GNNs of the set of query points $Q$ and computes it’s aggregate obstructed distance, until the best $k$ OGNNs have been found.

We have also shown algorithms for measuring the aggregate obstructed distance in terms of the total (sum) and the maximum travel distance (max) of the group members. We proposed two approaches: Single Point Aggregate Obstructed Distance (SPAOD) computation and Multi Point Aggregate Obstructed Distance (MPAOD) computation for finding aggregate obstructed distance between a data
point \( p \) and a set of query points \( Q \). The key difference is in the way the two algorithms retrieve obstacles from the database. SPAOD does not retrieve the same obstacle multiple times like MPAOD, but SPAOD may retrieve some extra obstacles. MPAOD may retrieve the same obstacle multiple times but does not retrieve any obstacle that is not required for the aggregate obstructed distance computation.

Our experimental results show the performance analysis of our algorithms for different parameters. We compare two of our proposed OGNN query algorithms: GBQM and CBQM. We find that CBQM incurs higher CPU time and IO cost than GBQM. The reason behind lower cost of GBQM is that, the probability of the Euclidean GNN being the obstructed group nearest neighbor is much higher than that of the nearest neighbor of centroid. As a result, CBQM retrieves more candidate data points, which as a result incurs more aggregate obstructed distance calculation. For this reason, GBQM performs better than CBQM for both \textit{sum} and \textit{max} in terms of CPU time and IO cost.

We also show a comparative analysis between our proposed algorithms for aggregate obstructed distance computation: SPAOD and MPAOD. We conclude with a result that SPAOD performs better than MPAOD in most of the cases. Since, SPAOD incrementally retrieves the obstacles from the centroid of the query points and no obstacle is retrieved multiple times, the IO access for SPAOD of the obstacle R-tree is much lower than that of MPAOD. Whereas, MPAOD may invoke same obstacle retrieval multiple times, which increases the IO cost. The increase of IO also increases the CPU time as more obstacle is retrieved from the obstacle R-Tree.

We have only considered static obstacles and query points in this thesis. We have plan to study OGNN queries considering moving obstacles (i.e, vehicles in pedestrians walking path). As a future work, we intend to investigate OGNN queries where the query points i.e, the group of users can move. In this thesis, we have considered only predefined data points or utilities where the group of users want to meet. Expanding OGNN queries for finding the optimal point or utility in a continuous space without any predefined data point is included in our future plan. Preserving the privacy of the group members while answering OGNN queries can be another field of study for our future works.
References


72
REFERENCES


REFERENCES


