M.Sc. Engg. Thesis

Efficient Processing of Maximum Visibility Facility Selection Query in Spatial Databases

by
Md. Ishat - E - Rabban

Submitted to

Department of Computer Science and Engineering

in partial fulfillment 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

June 2017

The thesis titled "Efficient Processing of Maximum Visibility Facility Selection Query in Spatial Databases", submitted by Md. Ishat - E - Rabban, Roll No. **1014052029 P**, Session October 2014, 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 June 21, 2017.

Board of Examiners

1. _____

Dr. Mohammed Eunus Ali — Chairman (Supervisor)
Professor
Department of Computer Science and Engineering
Bangladesh University of Engineering and Technology, Dhaka.

2. _____

Dr. M. Sohel Rahman — Member (Ex-Officio)
Head and Professor
Department of Computer Science and Engineering
Bangladesh University of Engineering and Technology, Dhaka.

3. _____

Dr. M. Kaykobad — Member
Professor
Department of Computer Science and Engineering
Bangladesh University of Engineering and Technology, Dhaka.

4. _____

Dr. Muhammad Abdullah Adnan — Member
Assistant Professor
Department of Computer Science and Engineering
Bangladesh University of Engineering and Technology, Dhaka.

5. _____

Dr. Shazzad Hosain — Member (External)
Associate Professor
Department of Electrical and Computer Engineering
North South University, Dhaka.

Candidate's Declaration

This is hereby declared that the work titled "Efficient Processing of Maximum Visibility Facility Selection Query in Spatial Databases" is the outcome of research carried out by me under the supervision of Dr. Mohammed Eunus Ali, 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.

Md. Ishat - E - Rabban

Candidate

Acknowledgment

Foremost, I am thankful to the Almighty for his blessings for the successful completion of my thesis. I would like to express my heartiest gratitude, profound indebtedness, and deep respect to my supervisor, Dr. Mohammed Eunus Ali, Professor, Dept. of CSE, BUET, Dhaka, Bangladesh, for his constant supervision, affectionate guidance and great encouragement and motivation. His keen interest on the topic and valuable advices throughout the study were of great help in completing this thesis.

I would also want to thank the members of my thesis committee for their valuable suggestions. I thank Dr. M. Sohel Rahman, Dr. M. Kaykobad, Dr. Muhammad Abdullah Adnan, and specially the external member Dr. Shazzad Hosain.

I am especially grateful to Department of Computer Science and Engineering (CSE) of Bangladesh University of Engineering and Technology (BUET) for providing their support during the thesis work. My sincere thanks goes to CSE Office staffs for providing logistic support to me to successfully complete the thesis work.

Finally, I would like to thank my family, my friends, and all of those who supported me for their appreciable assistance, patience, and suggestions during the course of my thesis.

Abstract

The widespread availability of realistic 3D models of cities, buildings etc. provides an opportunity to answer many real-life queries involving visibility in the presence of 3D obstacles. For example, *"Where to place security cameras to ensure better surveillance of a building?"*, or *"Where to place billboards in a city to maximize visibility from the surrounding space?"*, these applications require measuring and maximizing visibility of the data space in the presence of obstacles. In this paper, we formulate the above problem of maximizing visibility by introducing the *MVFS* query. In the *MVFS* query, we are given a set of obstacles, a set of n locations where a facility (i.e., camera, billboard, watch tower etc.) can be established, the visibility range of the facility, and an integer k, we select k locations from the given n locations to establish facilities such that the aggregated visibility coverage of the surrounding data space is maximized. We develop two exact algorithms for the *MVFS* problem that use various acceleration techniques to speedup the computation. We also outline a greedy approximation algorithm with proven approximation ratio of $1-\frac{1}{e}$. Usually real 3D models consist of a huge number of objects/obstacles. Consequently we develop several scalable algorithms for the *MVFS* problem which are suitable for datasets with huge number of obstacles that do not fit in main memory. To deal with the huge obstacle set, first we propose a naive disk resident implementation of the above greedy approximation algorithm and then we develop two improved algorithms that result in less IO overhead than the naive greedy implementation. We also address several variants of the *MVFS* problem so that our proposed algorithms can be applied to more generalized and realistic scenarios. We conduct comprehensive empirical analysis to investigate the performance of our proposed algorithms. The experimental results show that the greedy approximation algorithm runs orders of magnitude faster than the exact algorithms and incurs an average approximation error of less than

0.1%. In case of disk resident algorithms, the experiments show that the acceleration techniques used in the improved algorithms considerably reduce the IO overhead in comparison with the naive greedy implementation.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

3D city models are becoming increasingly available through popular mapping services such as Google Maps, Google Earth, Bing Maps, and OpenStreetMap. These map based services also allow users to upload 3D models representing buildings of their own cities. With the crowd-powered data collection of 3D datasets, spatial database community in recent years have witnessed a huge growth of 3D spatial data, which opens a new avenue of research involving 3D datasets. We envision that these 3D datasets will provide a new platform for answering many real-life user queries, e.g., visibility queries in the presence of 3D obstacles, that form the basis of a large class of location based applications.

A number of visibility related problems is currently being investigated by researchers in the field of visual sensor networks and spatial database. For example, the Optimum Camera Placement (*OCP*) problem [1] [2] [3], the Maximum Visibility Query (*MVQ*) [4] [5], the construction of the Visibility Color Map (*VCM*) [6] [7] [8] etc. All the above problems involve computing visibility or visible distance in an obstructed data space. In this thesis work, we introduce a novel visibility query and propose several main memory based and disk resident solutions to efficiently process the query.

In this chapter, first we introduce the problem and discuss the motivation behind the problem formulation. Next we briefly discuss the current visibility related works related to our problem. Then we provide a high level overview of the methodology we have developed to solve the problem. Next we list the contributions of this thesis work. Finally we provide an outline of the

organization of this thesis.

1.1 Problem and Motivation

In this work, we focus on a new type of query that we name Maximum Visibility Facility Selection (*MVFS*) query for facility placement in the presence of obstacles. Given a set of obstacles in the data space, a set of n potential locations in the data-space, D, where a new facility can be placed, and the visibility range of the facilities, we select k locations from the set of potential locations for placing new facilities such that the combined visibility coverage is maximized. We also address several extensions of the aforementioned problem so that our proposed techniques can be applied to more generalized and realistic scenarios.

In the *MVFS* problem formulation, we use a closed set of candidate locations, D, where we can place a facility. This is because in a practical scenario, it might be infeasible to establish a facility anywhere in the data space. Consequently, we restrict the positions where a facility can be established by introducing the set of data points, D, from which we chose k positions that maximizes the aggregated visibility coverage.

The motivation of the *MVFS* query comes from the following applications. A corporate office may want to place security cameras (i.e., facilities) to keep the office area under surveillance; now they may need to choose a set of locations to place the security cameras from a set of potential locations where a camera can be placed, so that the overall surveillance of the office through the security cameras is maximized. Similarly, an advertisement company may want to select locations to place a set of new billboards so that the area from which a billboard is visible is maximized. A tourism office may want to establish a set of new decks to watch a beautiful sea-beach where the goal is to maximize the visibility coverage of the sea-beach through all the decks. Our proposed methodologies can be used to serve all these queries efficiently.

1.2 State of the Art

Researchers in the fields of visual sensor networks and spatial databases are working on several visibility related problems currently. Examples include the Optimum Camera Placement (*OCP*) problem [1] [2] [3], the Maximum Visibility Query (*MVQ*) [4] [5], the construction of the Visibility Color Map (*VCM*) [6] [7] [8] etc.

In the *OCP* problem, we determine the minimum number of cameras required to cover a given area of interest. The *OCP* problem is similar to the *MVFS* problem. The existing exact solutions of the *OCP* problem are based on binary integer programming techniques. The exact algorithms perform poorly and consequently can not be used to solve a large scale practical scenario. In the current literature, the data space is discretized by representing the area of interest as a set of control points. There is no existing solution that calculates the actual area/volume of the visibility coverage. Besides, currently there is no methodology to handle a disk resident large set of obstacles. In this thesis work, we address these limitations of the existing works on the *OCP* problem and introduce novel techniques to overcome them.

In the *MVQ*, we select a query point q from a set of given query points Q, such that q provides the maximum visibility of a given extended target object. In the *VCM* problem, we construct a visibility map of the data space, where each point p in the data space is given a color value indicating the visibility of a given extended target object from p. In both problems, visibility of the extended target object T is measured by the visible surface area of T. Note that, the objective of the *MVFS* query is to select a subset of data points that maximizes aggregated visibility coverage, which is intrinsically different from the objectives of the *MVQ* problem and the *VCM* problem. Consequently the ideas used in these problems can not be directly applied to solve the *MVFS* problem.

We use original ideas and novel techniques to solve the *MVFS* problem. However, we adopt a projection based method from computer graphics [9] and borrow some ideas from computational geometry [10] to solve some subproblems which are used as building blocks of our solution to the *MVFS* problem.

1.3 Overview of Methodology

In practice, the model of a city or a building contain a large number of objects or obstacles. If the number of obstacles in the dataset is too large to fit in main memory, performing *MVFS* query will be prohibitively costly because of high IO overhead. Consequently we propose both main memory based and disk resident algorithms for the *MVFS* problem.

We discuss the main memory based algorithms as follows. First we propose an exact solution of the *MVFS* problem, namely, the *continuous exact algorithm*. The continuous exact algorithm considers the k element subsets of D, calculates the visible area/volume for the subsets, and reports the optimum choice. It uses several acceleration techniques to speedup the computation. The continuous exact algorithm constructs a triangulation of the data space to calculate the visible area/volume from data points. Next we present the *discrete exact algorithm*. Instead of calculating the visible area/volume from data points accurately, the discrete exact algorithm partitions the data space into equal sized cells and uses cell counts to approximate the area of a region. It computes the optimum solution of the *MVFS* query in a data space discretized into cells by using the idea of projection. Both the continuous and discrete exact algorithm require exponential time with respect to n, as they consider all length k subsets of D in the worst case. Consequently, we develop a *greedy approximation algorithm* that runs in time polynomial to n. The greedy algorithm, at each step, chooses the data point that maximizes the visibility of the still uncovered region and reports the first k such choices. This algorithm has theoretically proven approximation ratio and generates near optimal results in practice.

We discuss the disk resident algorithms as follows. In the scalable solutions, we index the obstacles in a persistent data structure, R-tree, and perform range queries on the R-tree to retrieve the obstacles located within the visibility range of the data points. First we present the disk resident version of the above greedy approximation algorithm, which issues n range queries on the R-tree, one for each data point in D. Issuing a range query is IO expensive as it requires several disk accesses. Hence, we propose the *best first algorithm*, which reduces the number of range queries by performing some preprocessing. In the preprocessing step, it calculates a heuristic that guides the best first search and thus reduces the number of range queries. Finally

we propose the *batch processing algorithm*, which gains farther performance acceleration by organizing closely situated data points into clusters. In the batch processing algorithm, we process each cluster of data points as a whole instead of processing each data point separately.

1.4 Contributions

We make the following contributions as listed below:

- We introduce the *MVFS* query to select k out of n data points to place facilities that maximizes aggregated visibility coverage in the presence of obstacles.
- We propose efficient exact algorithms to solve the *MVFS* problem. We use several acceleration techniques to reduce the computational cost.
- We develop main memory based solutions to the *MVFS* problem for continuous data space. We introduce novel techniques to calculate the area/volume of the aggregated visibility coverage.
- We propose several disk resident algorithms for the *MVFS* problem which are suitable for large datasets. We use pruning and clustering techniques to reduce the IO overhead.
- We conduct extensive experimentation to evaluate the effectiveness and efficiency of our proposed algorithms.

1.5 Organization

We outline the organization of this paper below. First we discuss some previous works related to the *MVFS* problem in Chapter 2. Next, we formally define the *MVFS* problem (Chapter 3). In Chapter 4, we present some background knowledge necessary for an uninitiated reader. As discussed above, we propose both main memory based and disk resident solutions to the *MVFS* problem. In Chapter 5, we describe the in memory solutions of the *MVFS* problem. The disk resident solutions of the *MVFS* problem are discussed in Chapter 6. The algorithms proposed

in Chapter 5 and Chapter 6 are described for 2D scenario. In Chapter 7, we discuss how the algorithms for 2D space can be modified to solve the *MVFS* problem for a 3D scenario. In Chapter 8, we address and solve some variants of the basic *MVFS* query. In Chapter 9, we present the empirical results to evaluate the effectiveness and efficiency of our proposed algorithms. Finally we make some concluding remarks in Chapter 10.

Chapter 2

Literature Review

A huge amount of research regarding visibility has been conducted in different domains [11]. In this chapter, we discuss previous research works that involve computing visibility in the presence of obstacles. We explore the visibility related works in several fields, i.e., visual sensor networks, spatial databases, computer graphics, computational geometry etc. In Section 2.1, we present the research works related to the *MVFS* problem in the domain of visual sensor networks. In Section 2.2, we discuss the visibility queries investigated in the field of spatial databases as this is the main focus of our research work. In Section 2.3 and Section 2.4, we present the visibility related works in computer graphics and computational geometry respectively.

2.1 Visibility in Visual Sensor Networks

Several visibility related problems are studied in the field of wireless network of visual sensors, i.e., Visual Sensor Networks (*VSN*). The Optimum Camera Placement (*OCP*) problem is very closely related to the *MVFS* problem, which is extensively studied by researchers in the field of *VSN*.

In the *OCP* problem, an area of interest is provided that is to be observed by visual sensors, i.e., cameras. The area of interest is represented discretely by a set of control points. The camera parameters, i.e., the viewing range, the field of view etc. are also provided. The *OCP* problem is to determine the minimum number of cameras required to cover the area of interest along with

the corresponding camera positions and their viewing directions.

The existing solutions to the *OCP* problem can be categorized into exact algorithms and approximate solutions. The exact solutions to the *OCP* problem are based on binary integer programming (*BIP*) techniques [12] [1] [13] [2]. In the *BIP* formulation of the *OCP* problem, the number of binary variables equals the sum of the number of data points and the number of control points. As a result, the performance of the exact solutions of the *OCP* problem is poor even for small instances. It is assumed in the literature that finding the exact solution of the *OCP* problem is infeasible for a practical large scale scenario. Consequently the contemporary works on the *OCP* problem focus on finding approximate solutions. The approximate solutions of the *OCP* problem are based on techniques used in artificial intelligence, such as, evolutionary algorithms [14] [15], particle swarm optimization [16] [17], and simulated annealing [3].

Observe that the *OCP* problem and the *MVFS* problem are related to each other in a similar way to the set cover problem and the maximum coverage problem. Consequently the *BIP* formulations of the *OCP* problem and the *MVFS* problem are similar to each other. Several variants of the *MVFS* problem are addressed in [1] [2] that use *BIP* formulation to find the exact solution. Horster et al. [1] presents an exact solution to the *MVFS* problem without considering the presence of obstacles. Debaque et al. [2] proposed an exact solution to the *MVFS* problem that performs visibility analysis offline, i.e., assumes that the occlusion effect of the obstacles is precalculated. As mentioned above, the performance of the exact solutions are poor because of large number of binary variables in the *BIP* formulation. For example, the *BIP* based exact algorithm proposed by Debaque et al. in [2] takes 2205 seconds on average to solve an *MVFS* instance with 6 data points. In the current literature, there is no efficient exact algorithm to solve the *MVFS* problem.

Note that, the existing solutions to the *OCP* problem and its variants assume a discretized data space. They sample the area of interest to form a set of control points and measure the visibility coverage in terms of the number of visible control points, instead of calculating the actual area/volume of the visibility coverage. Besides, there is no existing solution that can handle a set of obstacles too large to fit in main memory. In this work, we address these limitations of the state of the art and propose novel techniques to overcome these limitations.

2.2 Visibility in Spatial Queries

Widespread availability of large scale 3D models has stimulated research works in visibility related spatial queries. The spatial database community is currently working with several queries that involve calculating the visibility of an object in the presence of obstacles. Examples include the maximum visibility query (*MVQ*), visibility color map (*VCM*) construction, and visible nearest neighbor (*VNN*) query. We discuss each problem as follows.

Recently the concept of Maximum Visibility Query (*MVQ*) is tossed in a work of Sarah et al. [4] that considers the effect of obstacles to quantify the visibility of an extended target object. Given a target object T, a set O of m obstacles, and a set Q of n candidate locations (i.e., query points), the *MVQ* finds the query point in Q that provides the maximum visibility of T. They generalize this query through *k-Maximum Visibility Query* (*kMVQ*), that finds the k query points that maximize the visibility of T. They measure the visibility of the target object T by the area on the surface of T that is visible from the query points. Haider et al. [5] address the problem of answering the *MVQ* for a moving target object. To answer the *MVQ*, the obstacles are retrieved according to their distance from the target object, and the surface area of T visible from the query points is calculated. They use several techniques to accelerate the query processing.

The construction of the Visibility Color Map (*VCM*) is a highly studied problem [6] [7] [8] in the field of spatial databases. In the *VCM* problem proposed by Choudhury et al. [6], an extended target object T, and a set of obstacle O are given. The problem is to construct a color map that assigns a color to each point p in the data space that indicates the visibility of T from p. To assign a visibility color to a point, they consider the distance, visual angle, and obstruction by the obstacles. Rabban et al. [7] consider partial visibility of the target object while constructing the *VCM* and also propose a method to construct the *VCM* for a moving target object. To construct the *VCM*, they retrieve the obstacles in increasing order of distance from the target object and determine the region in the data space obstructed by the obstacles. Thus the visibility color values are assigned to the points in the data space.

Note that, both *MVQ* and *VCM* problem deal with calculating what portion of an extended target object is visible from a point in the data space. In the *MVQ* problem, they do not construct

and/or aggregate the visibility coverage of the query points. In the *VCM* problem, if we consider the target object to be a point, their proposed methodology can be used to construct the region in the data space visible from a data point. We mention the ideas we adopt from the work of Rabban et al. [7] in Chapter 5.2.2. For the subsequent steps in the *MVFS* problem, that aggregates the visibility coverage of separate data points, we use original ideas that are out of the scope the *VCM* problem.

Other queries related to visibility studied in spatial databases involve finding nearest neighbors, such as, Visible Nearest Neighbor (*VNN*) query [18] [19] [20], Continuous Obstructed Nearest Neighbor (*CONN*) query [21], and Continuous Visible Nearest Neighbor (*CVNN*) query [22], and Visible Reverse Nearest Neighbor (*VRNN*) [23] query. In general the nearest neighbor query finds the k nearest data points with respect to a query point based on a given distance measure. Nutanong et al. introduce the visible nearest neighbor (*VNN*) problem where they find the nearest neighbor that is visible to a query point in the presence of obstacles. The basic idea is to perform nearest neighbor search and check its visibility condition incrementally.

There are several variants of the nearest neighbor problem that consider visibility between the query point and the data points in the presence of obstacles. The Continuous Visible Nearest Neighbor (*CVNN*) query finds the nearest neighbor results for a moving query point [22]. Given a set of data points, a set of obstacles, and a line segment, the *CVNN* query returns a set of (point, interval) tuples, such that the point is the nearest neighbor of all points in the corresponding interval of the segment. Gao et al. studied a variation of nearest neighbor query namely Continuous Obstructed Nearest Neighbor (*CONN*) query [21]. Given a set of data points, a set of obstacles, and a query line segment q in a 2D space, a *CONN* query retrieves the nearest neighbor of each point on q according to the obstructed distance, i.e., the length of the shortest path that avoids the obstacles. Another variant of the nearest neighbor query is the Visible Reverse Nearest Neighbor (*VRNN*) [23]. Given a set of data points P, a set of obstacles, and a query point q, the *VRNN* query retrieves the points in P that have q as their nearest neighbor and are visible to q.

The problem of finding the nearest neighbor is fundamentally different from the *MVFS* problem. In the aforementioned nearest neighbor queries, they find the nearest object in an obstructed space from a given query point where query results are ranked according to the distances or visi-

ble distances from the query point. Whereas in the *MVFS* problem, we determine and aggregate the visibility coverage of a set of data points in the space. Thus the methodology used to determine the nearest neighbor can not be applied to answer the *MVFS* query.

2.3 Visibility in Computer Graphics

The idea of visibility is central to the field of computer graphics, as computer graphics deals with generating realistic image of a virtual model from a given viewpoint. A number of problems regarding visibility is actively studied by graphics researchers, including hidden surface removal, occlusion culling, global illumination, ray tracing, radiosity etc. [24] [25] In this section, we focus on the hidden surface removal and occlusion culling problem. The other problems mentioned above are used to simulate light realistically and are dealt in an image-precision manner. Consequently they are out of the scope of our work.

Hidden surface removal and occlusion culling methods determine the surfaces in the model that are visible from the viewpoint. We discuss the object-precision solutions of these problems, as the image-precision solutions are irrelevant to the *MVFS* problem. The state of the art method to determine the visible surfaces from a moving viewpoint computes the Potentially Visible Set (*PVS*) [26]. The *PVS* is usually a small subset of the model that contains all visible surfaces. The *PVS* is calculated by fusing obstacles to form large virtual obstacles that facilitate occlusion culling [27] [28] [29]. Durand et al. [9] propose a projection based method to accelerate the *PVS* computation. Koltun et al. [30] introduce a hardware assisted method that farther improves the performance for a 2.5D scene. We use the projection based idea proposed by Durand et al. in [9] to determine the visible region of a data point in a discretized data space.

Note that, the hidden surface removal and occlusion culling techniques used in computer graphics focus on determining the visible surfaces from a single viewpoint in the data space. We can not apply these methods to compute and/or maximize the combined visibility coverage of multiple data points. Thus the methodologies used in computer graphics can not be directly applied to solve the *MVFS* problem.

2.4 Visibility in Computational Geometry

The problems in computational geometry that involve visibility computation include visibility polygon construction [10] [31] [32] [33], visibility graph construction [34] [35], and art gallery problems [36] [37] and its variants. We discuss each as follows.

The visibility polygon of a point q inside a polygon P consists of all points in P that are visible from q. Suri et al. and Asano [10] [31] addressed the problem of computing the visibility polygon of a query point inside a non-simple polygon with holes. Their approach performs a rotational plane sweep around q and determine the visible edges of P to construct the visibility polygon. Zarei et al. [32] [33] propose a method where they add new edges and vertices to the non-simple polygon to unfold it along those edges and convert it into a simple polygon.

In the visibility graph problem, we are given a set P of n points inside a polygon Q. The objective is to construct a graph whose nodes are the points in P and there exists an edge between two nodes/points p and q, if the p and q are visible from each other, i.e., the line segment pq is not intersected by any edge of the polygon Q. Ben et al. [34] proposed near optimal solutions to the problem of constructing the visibility graph for simple and non-simple polygons with holes.

The art gallery problem asks to determine the number of guards necessary to cover an art gallery. The art gallery is modeled as a simple polygon and each guard is considered to be a point in the polygon. A set of guards S is said to cover the art gallery if, for every point p in the polygon, there is some guard $q \in S$ such that the line segment between p and q does not leave the polygon. The art gallery problem is a highly studied problem in computational geometry [36] [37] and there are several variants. The solution of the basic art gallery problem involves constructing triangulation of the art gallery/polygon.

As part of the solution to *MVFS* problem, we require to determine the region visible from a data point. We reduce the problem of constructing the visible region of a data point to the problem of constructing the visibility polygon as discussed in Chapter 5.1.1. We adopt the methodology proposed by Asano [10] to construct the visibility polygon. The other works in computational geometry regarding visibility discussed above are not related to the *MVFS* problem.

Chapter 3

Problem Formulation

In this chapter, first we define some terms regarding visibility that are necessary for our formulation. Next we provide the formal definition of our problem. Then we discuss some assumptions on which our solution is built upon. Finally we present an illustrative example to clearly explain the *MVFS* problem.

We define the visibility of a point in the data space to/from a data point and the visible region of a set of data points as follows.

Definition 1 ***Visibility between a Point and a Data Point****: Given the set of obstacles, O, and the viewing range of a facility, r, a point p in the data-space is defined as visible to/from a data point, d, if the line segment joining p and d intersects no obstacle in O and the distance between p and d, $dis(p, d)$, is less than or equal to r. Otherwise, p is defined as non-visible to/from d.*

Definition 2 ***Visible Region of a Set of Data Points****: Given the set of obstacles, the viewing range of a facility, and a set of data points, D, the visible region of D comprises all points p in the data space, such that, p is visible to/from at least one data point $d \in D$.*

We formally define our problem as follows.

The MVFS Problem: Given an m-dimensional data-space R^m (m=2 or 3), the set of obstacles in the data-space, the set of n data points where a facility can be placed, D, the viewing range of a facility, and an integer k ($1 \leq k \leq n$), the problem is to determine a subset S of D,

$S \subseteq D$, of size k, $|S| = k$, to establish k facilities, such that, the area (or volume, in 3D) of the visible region of S is maximized. Here the notation $|.|$ stands for the cardinality of a set.

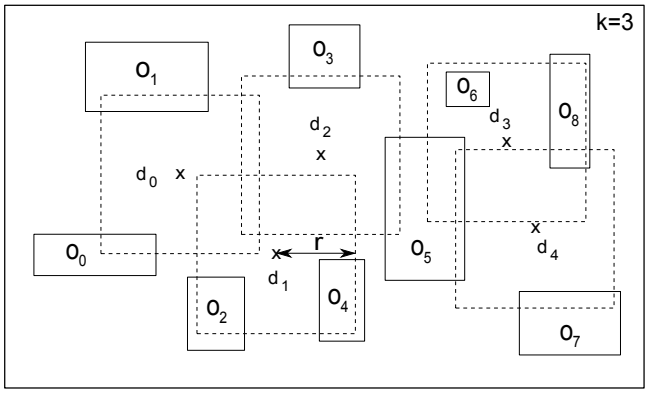
However, we also address several other variations of the *MVFS* problem that generalize the above primary formulation. We discuss those extensions in Chapter 8.

We adopt a distance metric other than the commonly used euclidean distance metric to measure the distance between two points in the data-space. Let the cartesian coordinate of a point, p, in the data space be denoted by (p_x,p_y,p_z), where p_z=0, if the data-space is 2D, and the distance between two points, p and q, be denoted by dis(p,q). Then,

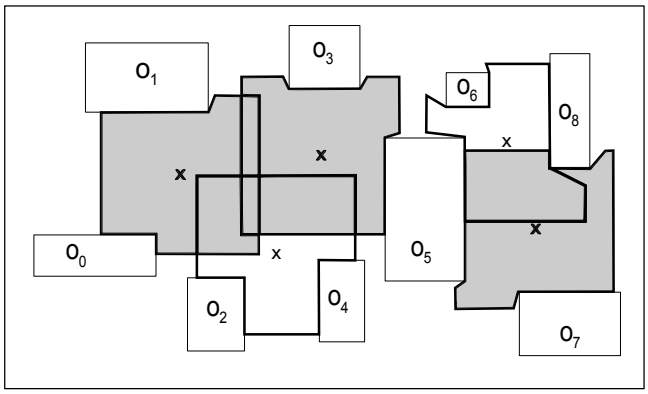
$$\text{dis}(p,q)=\max(|p_x\text{-}q_x|,|p_y\text{-}q_y|,|p_z\text{-}q_z|)$$

Here, the notation $|.|$ stands for absolute value of a number. The distance between two points defined by the above equation is called the *supremum distance*. According to supremum distance metric, the region that a facility with visibility range r covers is an axis aligned square (in 2D) or cube (in 3D) with side length $2r$ centered at the facility location. We adopt the supremum distance metric for the purpose of ease of implementation and explanation. Our solution can be easily adjusted to work for a more conventional distance metric, i.e., euclidean distance metric.

Now we clarify the ideas established thus far with an illustrative example. Figure 3.1(a) shows a simple 2D instance of the *MVFS* problem. In the scene, there are 5 data points (d_0 to d_4, cross marked), and 9 obstacles (o_0 to o_8). The viewing range of the camera is shown as r and it is given that $k = 3$. Under the supremum distance metric, the viewing range of each data point is a square (shown as dotted squares in Figure 3.1(a)). In Figure 3.1(b), the visible regions of each data point is shown separately with bold boundaries. The three element subset of D having visible region of maximum area is $\{d_0,d_2,d_4\}$. The visible region of $\{d_0,d_2,d_4\}$ is shown in light grey in Figure 3.1(b). d_3 is not selected because the area of its visible region is small due to occlusion by obstacles. d_1 is not selected because its obstructed by the obstacles and also its visible region highly overlaps with the visible regions of d_0 and d_2. Table 3.1 lists the commonly used notations and their meanings.



(a)



(b)

Figure 3.1: (a) An instance of the *MVFS* problem. (b) Visible regions from the data points and the optimum choice for k=3.

Table 3.1: Commonly Used Notations and Their Meanings

Notation	**Meaning**
n	Number of data points
D	Set of data points
d_i	i^{th} data point in D
O	Set of obstacles
G	Grid partitioning of the data space
T	R-tree indexing the obstacles
V	Visibility matrix
c_{size}	Cluster radius
r	Viewing range of a facility
p_s	Visibility Status of point p
c_s	Visibility Status of cell c
t_s	Visibility Status of triangle t
t_a	Area of triangle t

Chapter 4

Background Study

In this chapter, we provide some background knowledge on some topics used in our solution. First we briefly discuss *R-tree*, a persistent spatial data structure extensively used in our disk resident algorithms for the *MVFS* problem (Section 4.1). Then we provide an introduction to hierarchical clustering algorithms (Section 4.2). We use a distance based hierarchical clustering algorithm in one of our disk resident algorithms.

4.1 R-tree

R-tree is a tree data structure that stores spatial objects and can process spatial queries on the objects efficiently. The R-tree was proposed by Antonin Guttman in 1984 [38]. The key idea of the R-tree is to group nearby objects together and form their minimum bounding rectangle (*MBR*). R-tree stores a hierarchy of minimum bounding rectangles where the spatial objects are situated at the leaf nodes. All leaf nodes are stored in the same height. Thus R-tree is a balanced tree (Figure 4.1).

The basic queries on the R-tree, i.e., intersection, containment, nearest neighbor search etc., are simple. The bounding boxes are used to decide whether or not to search inside a subtree. Thus a lot of nodes are pruned out and never read during a query. This characteristics of R-tree makes it suitable for large data sets and databases. R-tree is designed for storage on disk. It assumes that the whole data cannot be stored in main memory and data is cached to memory

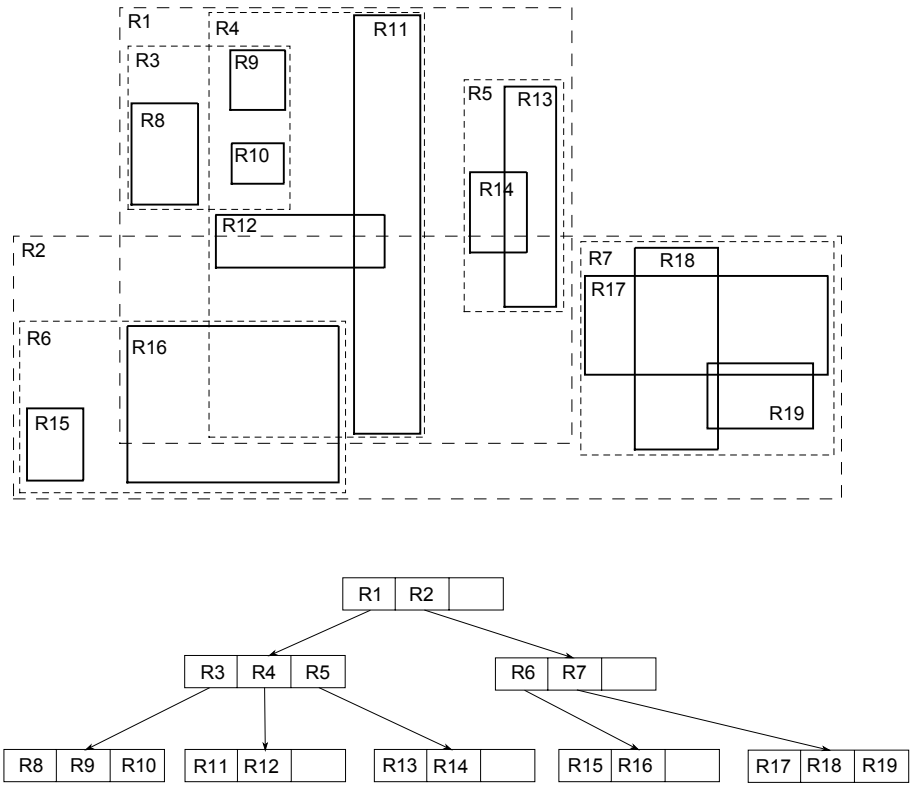


Figure 4.1: An example of an R-tree storing rectangles.

when it is required.

To build an efficient R-tree, one has to ensure that the tree remains balanced and also the rectangles do not cover too much empty space and do not overlap too much. For example, during the insertion operation, to obtain an efficient tree, the element is inserted into the subtree that requires least enlargement of its bounding box. Once a page is full, the data is split into two sets such that each cover minimal area. Different heuristics are used to reduce the empty space covered by the rectangles and the overlap between rectangles. These heuristics result in several variants of the R-tree, namely, R^*-tree [39], R^+-tree [40], and X-tree [41].

4.2 Hierarchical Clustering

Clustering is the task of organizing a set of objects into separate groups or clusters such that objects in the same cluster are more similar to each other than to objects in another cluster. In

clustering, similarity can be assessed by different metrics. A specific class of clustering algorithms, called *Hierarchical Clustering* [42] [43] [44], uses distance as the metric to determine similarity between objects and connect objects to form clusters based on their distance. Hierarchical clustering methods differ by the distance function.

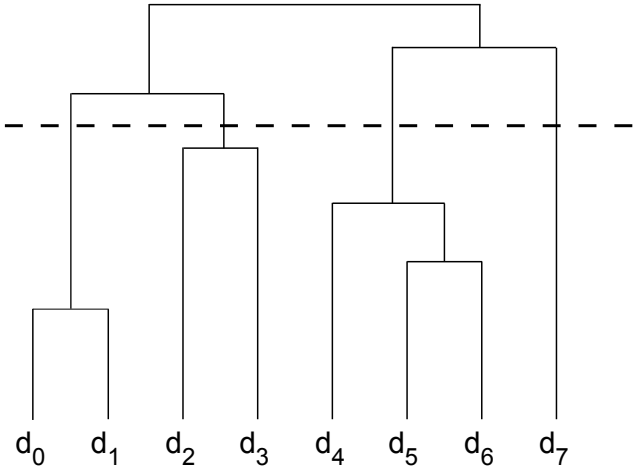


Figure 4.2: A dendogram.

A cluster can be defined by the maximum distance needed to connect parts of the cluster. At different cutoff values, different set of clusters form. This idea is represented by a *dendrogram* (Figure 4.2). In a dendrogram, the vertical axis marks the distance at which the clusters merge, while the objects are placed along the horizontal axis such that the clusters do not mix. The hierarchy can be formed in an agglomerative way (starting with single elements and aggregating them into clusters) or in a divisive way (starting with the complete data set and dividing it into partitions).

Note that, hierarchical clustering algorithms do not provide a specific partitioning of the data set. Instead it provides a hierarchy of clusters that merge with each other at certain distances as shown in the dendogram. Thus the user gets to choose the cutoff distance and obtains the appropriate set of clusters.

Chapter 5

Main Memory based Algorithms

In this chapter, we present three main memory based solutions to the *MVFS* problem. In these three main memory based algorithms, it is assumed that the number of obstacles in O or the dataset is small enough to fit in main memory. First, in Section 5.1, we describe the *continuous exact algorithm* for the *MVFS* problem. The continuous exact algorithm uses the idea of triangulation to compute the area of visible regions and reports the optimum result of the *MVFS* problem. The *discrete exact algorithm*, discussed in Section 5.2, discretizes the data space into small equal sized cells and determines the optimum solution to the *MVFS* problem in a data space partitioned into cells. The discrete exact algorithm achieves considerable speedup over the continuous exact algorithm by considering the cells as the building blocks of the data space and thus avoiding actual area computations. Both the continuous and discrete exact algorithms find the optimum solution of the *MVFS* problem and take exponential time with respect to n, the number of data points, in the worst case. Finally, in Section 5.3, we present a greedy approximation algorithm, that, at each step, makes a greedy choice by selecting the data point that maximizes the area of the still uncovered region and determines an approximate solution to the *MVFS* problem. The greedy approximation algorithm has theoretically proven approximation error bound and practically generates outputs very close to the optimum results.

5.1 The Continuous Exact Algorithm

In this section, we discuss the continuous exact algorithm that determines the optimum output of the *MVFS* problem. The key idea of the continuous exact algorithm is to partition the data space into disjoint triangles such that all points within a triangle are visible from the same subset of data points in D. After the triangulation is constructed, we can calculate the area of the region visible from a subset of D by adding up the area of appropriate triangles. A straightforward solution of the *MVFS* problem is to generate all subsets of size k of the set of n data points, compute the visible area for each such subset, and report the subset resulting in the visible region with maximum area. But we use several acceleration techniques to avoid processing all subsets of size k and thus speed up the computation.

We discuss the organization of this section as follows. First, in Section 5.1.1, we discuss the process of determining the visible region from a data point in the presence of obstacles and the triangulation of the visible region. Next, in Section 5.1.2, we describe the methodology to construct the triangulation of the data space with respect to the n data points in D. Finally, in Section 5.1.3, we present the continuous exact algorithm in details.

5.1.1 Constructing the Visible Region of a Data Point

In this section, we discuss the process of constructing the visible region of a data point, given the obstacles within its range. We also describe how to triangulate the visible region. The problem of constructing the visible region of a data point is similar to the problem of constructing the visibility polygon with respect to a query point in a polygon with holes. We will refer to this problem as the *visibility polygon* problem. The visibility polygon problem is a well studied problem in computational geometry. We adopt the work of Asano [10] on the visibility polygon problem to construct and triangulate the visible region of a data point.

First we define the visibility polygon problem. Then we state the correspondence between the visibility polygon problem and our problem of determining the visible region of a data point in the presence of obstacles. Next we provide a brief outline of the process of constructing the visible region according to the work of Asano and discuss some implementation issues. Finally

we discuss the process of triangulating the visible region of a data point.

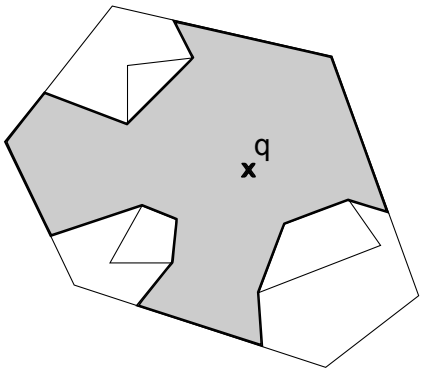


Figure 5.1: Visibility polygon for query point q in a polygon with 3 holes.

The Visibility Polygon Problem: Let P be a polygon containing m vertices and h holes. Let q be a query point in P that is not inside any hole. Then, the *visibility polygon* with respect to q, V, is the set of points, such that for every point $p \in V$, the segment pq does not intersect any edge of the polygon P.

Figure 5.1 explains the visibility polygon problem. The figure shows a polygon with 17 vertices and 3 holes. The visibility polygon for query point q (cross marked) is shown in grey.

Note that, the problem of constructing the visible region of a data point is reducible to the visibility polygon problem. Given a data point d and a set of obstacles within the range of d, O_d, we can construct a polygon P, whose exterior is an axis aligned square S of side length $2r$ centered at d. Here r is the camera range. For each obstacle in O_d, we add a hole in P. If the obstacle spans outside S, we clip off the portion of the obstacle outside S. Thus, the holes will stay inside S. Finally the data point d corresponds to the query point q in the visibility polygon problem. The output of the visibility polygon problem is the sequence of vertices bordering the visible region from q. Thus we can reduce an instance of the problem of determining the visible region of a data point in the presence of obstacles to an instance of the visibility polygon problem.

Refer to Figure 5.2 for better understanding of the process of constructing visible region of a data point by reducing to visibility polygon problem. Here we simulate the process of constructing the visible region of data point d_3 in Figure 3.1. There are 3 obstacles inside the visibility

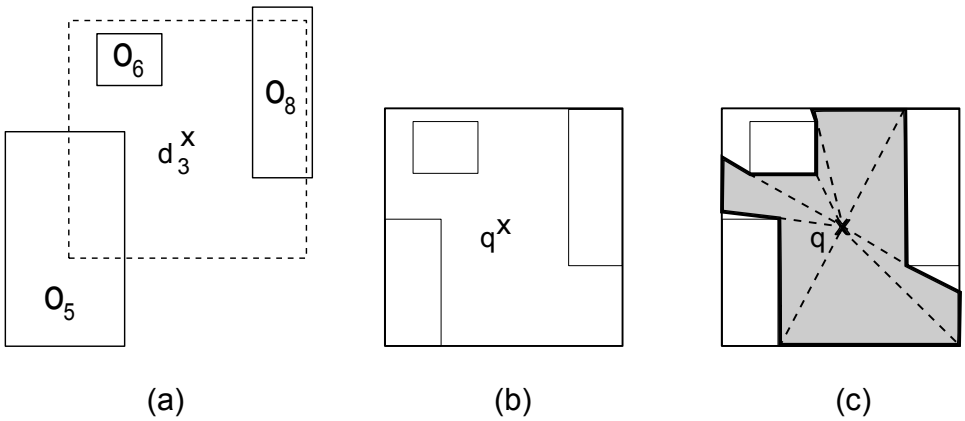


Figure 5.2: Constructing visible region of D_3 by reducing to visibility polygon problem.

range of d_3 as shown in Figure 5.2(a). We reduce this scenario to an instance of visibility polygon problem, where the polygon has 16 vertices and 3 holes (Figure 5.2(b)). The data point d_3 becomes the query point q. Finally Figure 5.2(c) shows the visible region of d_3.

We briefly outline the procedure of determining the visibility polygon as proposed by Asano in [10] as follows. The idea is based on angular sweep or rotational line sweep around the query point q. First we sort all the hole segment endpoints by their respective polar angle around q and iterate over the endpoints in that order. During the rotational sweep, we maintain the segments intersecting with the sweep line in a balanced binary search tree T based on their order of intersections with the sweep line. As the sweep proceeds, T is updated and a new output vertex is generated each time the smallest element (segment closest to q) in T changes. This implementation has a time complexity of O(n log n), where n is the number of vertices in the polygon.

Now we describe the process of triangulating the visible region of the data point. From Figure 5.2(c), notice that connecting the data point with each pair of adjacent vertices of the visibility polygon creates a triangulation of the visible region. In some of these triangles, all three vertices lie in a straight line resulting in an area of 0. These triangles are degenerate, and hence are ignored.

5.1.2 Constructing the Visibility Triangulation of the Data Space

In this section, we describe the process of partitioning the data space into disjoint triangles such that all points within a triangle are visible from the same subset of data points in D. We call this triangulation the *Visibility Triangulation* of the data space. First we discuss some basic definitions and notations useful for the construction of the visibility triangulation. Then we present the algorithm that creates the visibility triangulation of the data space.

We define the *visibility status* of a point in the data space and the *visibility triangulation* of the data space as follows.

Definition 3 ***Visibility Status of a Point****: Given a set of n data points D, and a set of obstacles O, the visibility status of a point p in the data space is defined to be a bitmap of length n, where, for $0 \leq i < n$, the i^{th} bit is set if p is visible from the i^{th} data point in D, and off otherwise.*

Definition 4 ***Visibility Triangulation****: Given a set of data points D, and a set of obstacles O, the visibility triangulation, T_v, of the data space is a partitioning of the visible region of D into disjoint triangles, such that for each triangle $t \in T_v$, every point inside t has the same visibility status.*

If every point inside a triangle t has the same visibility status, we define the *Visibility Status of* t to be the visibility status of a point inside t. We denote the visibility status of a point p by p_s and the visibility status of a triangle t in the visibility triangulation by t_s.

We define two boolean operations on triangles as listed below.

- Given two triangles, t_a and t_b, the boolean *intersection* operation on t_a and t_b returns the region common to both t_a and t_b. We denote the boolean *intersection* operation on two triangles t_a and t_b by $t_a \cap t_b$.
- Given two triangles, t_a and t_b, the boolean *subtraction* operation on t_a and t_b returns the region of t_a not inside t_b. We denote the boolean *subtraction* operation on two triangles t_a and t_b by t_a/t_b.

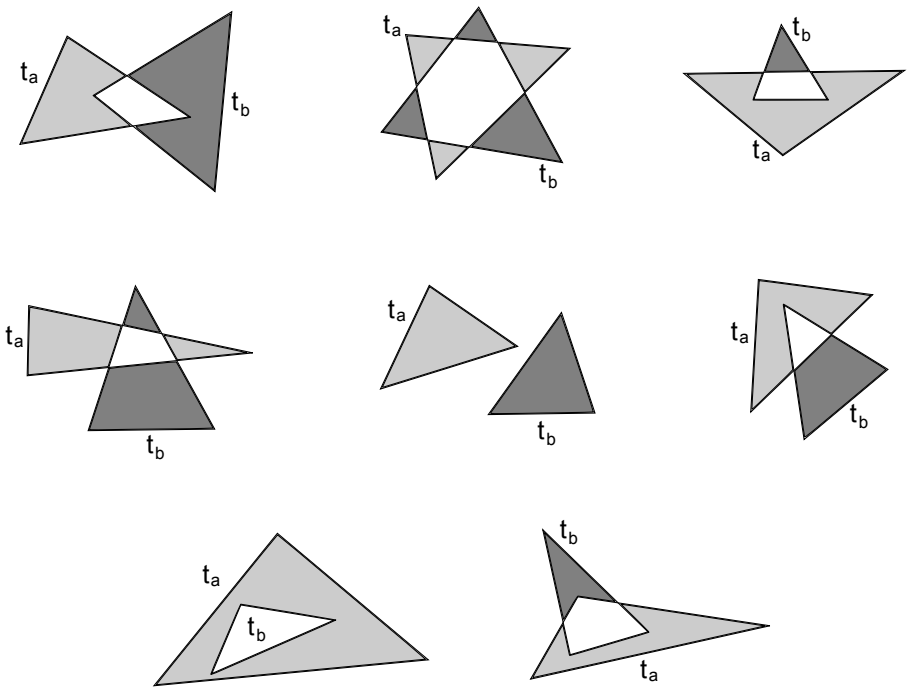


Figure 5.3: Boolean intersection and boolean subtraction operation on two triangles.

Figure 5.3 illustrates the idea of boolean intersection and subtraction operation on two triangles. The white region is the boolean intersection of t_a and t_b. t_a/t_b is shown in light grey and t_b/t_a is shown in dark grey.

To construct the visibility triangulation of the data space, we implement routines to calculate the boolean intersection and subtraction region of triangles and also triangulate the intersection and subtraction regions. Now we briefly discuss how to determine and triangulate intersection and subtraction region of two triangles.

Determining Boolean Intersection of Triangles: We observe that, clipping one triangle with respect to another triangle is similar to finding the boolean intersection of the two triangles. We use the Sutherland-Hodgeman polygon clipping algorithm [45] to determine the boolean intersection of two triangles. Triangulating the boolean intersection region is a trivial task as, if nonempty, the common region of two triangles is always a convex polygon.

Determining Boolean Subtraction of Triangles: Given two triangles, t_a and t_b, to determine the boolean subtraction region, t_a/t_b, first we find all intersecting points of the two triangles. Next we list the vertices of t_a and the intersecting points in order they appear on the boundary

of t_a. Then we traverse the list and keep track of whether the points are inside or outside t_b. We discard the region which is inside t_b and thus subtract t_b from t_a. To triangulate the subtraction region, we add cords to the region as long as the cord does not go outside the region and does not intersect any existing cord.

We present the algorithm *visTriangulation* that constructs the visibility triangulation of the data space below. First we discuss the function of the routines and variables used in the algorithm.

rangeQuery: The *rangeQuery* routine takes a data point, the set of obstacles, and the visibility range as input parameters and returns the subset of obstacles that intersect or falls within the visibility range of a facility placed at the data point. As it is assumed that the set of obstacles is small, we naively implement the *rangeQuery* routine by checking the query box against all obstacles sent as the parameter. This routine is necessary to set appropriate input parameter of the *visRegion* routine.

visRegion: The *visRegion* routine takes a data point, the set of obstacles within the visibility range of the data point, and the visibility range as input parameters and returns a set of triangles defining the visible region of the data point. We have discussed the process of constructing and triangulating the visible region of a data point in Section 5.1.1. The routine *visRegion* is implemented accordingly.

triangulate: The *triangulate* routine takes a polygonal region as input parameter and returns a list of triangles defining a triangulation of the region. This routine is used to triangulate the boolean intersection and subtraction region of two triangles. The routine *triangulation* is implemented as discussed earlier in this chapter.

In the algorithm *visTriangulation*, d_i denotes the i^{th} data point in D, t_s denotes the visibility status of triangle t, and subscripted L's denote lists of triangles. Now we describe the algorithm *visTriangulation* in details.

The algorithm *visTriangulation* constructs the visibility triangulation of the data space incrementally, including one data point at a time. Initially, in Lines 2-5, we construct the visibility triangulation for the first data point. Then, in the for loop spanning from Lines 6-30, we add one data point at each iteration and incrementally reconstruct the visibility triangulation. At the

Algorithm 1: visTriangulation(O,D,r)

```
   input  : O,D,r
   output: Visibility Triangulation of the Data Space
1  begin
2      O_rq ⟵ rangeQuery(d_0, O, r)
3      L_old ⟵ visRegion(d_0, O_rq, r)
4      for each triangle t in L_old do
5          t_s ⟵ 1
6      for i ⟵ 1 to n − 1 do
7          O_rq ⟵ rangeQuery(d_i, O, r)
8          L_new ⟵ visRegion(d_i, O_rq, r)
9          L_common ⟵ ∅
10         for each triangle t_old in L_old do
11             for each triangle t_new in L_new do
12                 L_1 ⟵ triangulate(t_old ∩ t_new)
13                 if L_1 = ∅ then
14                     continue
15                 L_2 ⟵ triangulate(t_old/t_new)
16                 L_3 ⟵ triangulate(t_new/t_old)
17                 for each triangle t in L_1 do
18                     t_s ⟵ t_s + 2^i
19                     L_common.insert(t)
20                 L_old.remove(t_old)
21                 for each triangle t in L_2 do
22                     L_old.insert(t)
23                 L_new.remove(t_new)
24                 for each triangle t in L_3 do
25                     L_new.insert(t)
26         for each triangle t in L_new do
27             t_s ⟵ 2^i
28             L_old.insert(t)
29         for each triangle t in L_common do
30             L_old.insert(t)
31     return L_old
```

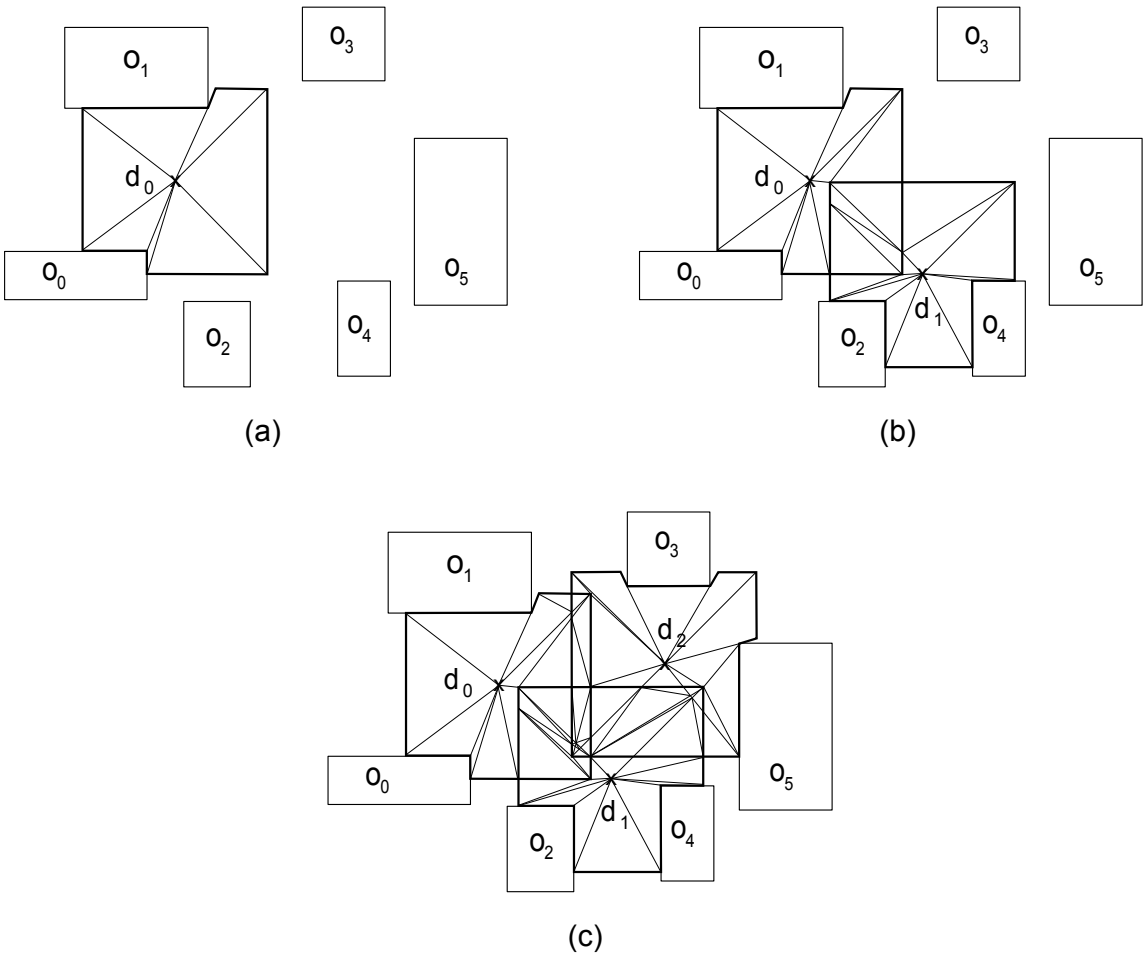


Figure 5.4: Construction of visibility triangulation for 3 data points.

beginning of each iteration of the for loop, the list L_{old} holds the visibility triangulation of the first i data points. In Lines 7-8, we construct the visibility triangulation for the $(i+1)^{st}$ data point and put it in the list L_{new}. In Lines 10-25, we consider all possible pair of triangles, (t_{old}, t_{new}), where $t_{old} \in L_{old}$ and $t_{new} \in L_{new}$. We create three lists of triangles L_1, L_2, and L_3, where L_1 contains the triangulation of the boolean intersection of t_{old} and t_{new} (Line 12), L_2 contains the triangulation of the boolean subtraction of t_{new} from t_{old} (Line 15), and L_3 contains the triangulation of the boolean subtraction of t_{old} from t_{new} (Line 16). We put the triangles in L_1, i.e., the common region of t_{old} and t_{new}, in the list L_{common} (Lines 17-19). Note that, the regions t_{old}/t_{new} and t_{new}/t_{old} may intersect with other triangles in L_{new} and L_{old} respectively. Consequently, we remove t_{old} from L_{old} and insert the triangulation of t_{old}/t_{new} in L_{old} (Lines

20-22), and we remove t_{new} from L_{new} and insert the triangulation of t_{new}/t_{old} in L_{old} (Lines 23-25). After the termination of the outer loop spanning Lines 10-25, L_{old} holds the triangles that are not in the visible region of the $(i+1)^{st}$ data point, L_{new} holds the triangles that are not in the visible region of the first i data points, and L_{common} holds the triangles that are in the common visible region of the first i data points and the $(i+1)^{st}$ data point. In Lines 26-30, we update L_{old} to construct the visibility triangulation of the first $i+1$ data points. Lines 5, 18, and 27 are added to maintain the visibility status of the triangles correctly.

Figure 5.4 shows the construction of visibility triangulation for the first 3 data points of the scenario depicted in Figure 3.1. Initially the visibility triangulation holds the triangulation of the visible region of d_0 (Figure 5.4(a)). In the next two iteration, data points d_1 and d_2 are added and the visibility triangulation is constructed incrementally as shown in Figure 5.4(b) and Figure 5.4(c).

5.1.3 The Algorithm

In this section, we propose the continuous exact algorithm for the *MVFS* problem. First we present a straightforward solution to the *MVFS* problem using the visibility triangulation of the data space. Then we address the bottlenecks of the straightforward solution. Finally we present an improved algorithm that removes the drawbacks of the naive solution.

We present the algorithm *naiveContinuousExact* that provides a straightforward solution to the *MVFS* problem. Here, t_a denotes the area of triangle t, $\sim$ and $|$ denote the bitwise *not* and bitwise *or* operators respectively. In this algorithm, first we construct the visibility triangulation of the data space. Then we generate all possible subsets of size k of the n data points and determine the visible area from each subset by adding up the area of the appropriate triangles from the visibility triangulation. Finally we report the subset with maximum area.

The algorithm *naiveContinuousExact* has two major performance bottlenecks as listed below.

- To calculate the visible area of a subset of data points, we traverse all triangles in the visibility triangulation.
- We consider all length k subsets of the n data points.

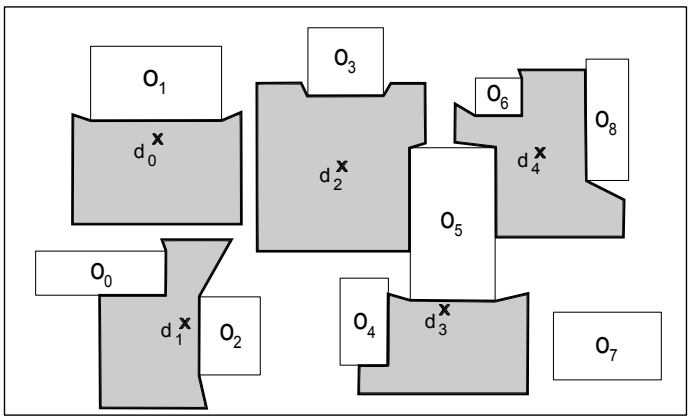
Algorithm 2: naiveContinuousExact(O,D,r,k)

input : O,D,r,k
output: Optimum Choice of k Data Points

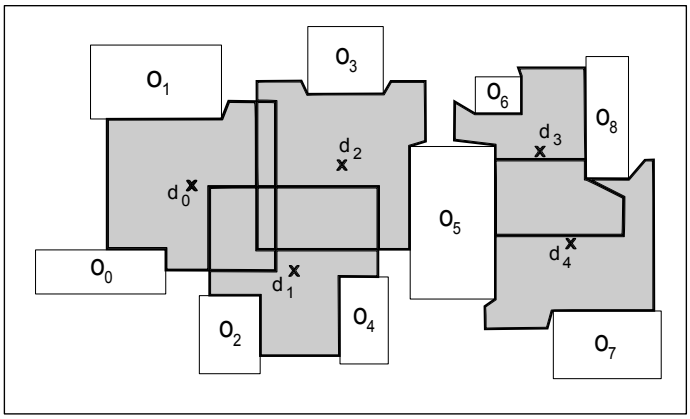
```
1  begin
2      L ⟵ visTriangulation(O, D, r)
3      maxArea ⟵ −1
4      maxChoice ⟵ 0
5      for each bitmap m of length n with k set bits do
6          area ⟵ 0
7          for each triangle t ∈ L do
8              if ∼ m|t_s = 0 then
9                  area ⟵ area + t_a
10         if area > maxArea then
11             maxArea ⟵ area
12             maxChoice ⟵ m
13     return maxChoice
```

We propose some acceleration techniques to remove these bottlenecks. To address the first bottleneck mentioned above, we observe that in the visibility triangulation, the number of unique visibility status values is far less than the number of triangles in the visibility triangulation. Consequently, for each unique visibility status value, we create an *element*. Each element has a key and a value. The value of an element with key s equals the sum of areas of all triangles in the visibility triangulation having visibility status s. To speed up the algorithm *naiveContinuousExact*, we execute a preprocessing step after the visibility triangulation is constructed, where we create a list of elements as described above. While calculating the visible area for a subset of data points, instead of searching all triangles in the visibility triangulation, we traverse the list of elements. Thus we remove the first bottleneck mentioned above. For example, in Figure 5.4(c), there are 68 triangles but only 7 distinct visibility status values.

We present the idea to avoid calculating the visible area for all length k subsets of the n data points below. First consider the scenario in Figure 5.5(a). The visible regions of the 5 data points are shown in grey. Suppose we want to generate the optimum output for k=3. Note that the visible regions of the data points do not overlap. As the visible regions do not overlap, a greedy approach where at each step the data point having maximum visible area is chosen works



(a)



(b)

Figure 5.5: Distribution of data points with (a) non-overlapping visible regions and (b) overlapping visible regions.

optimally. Consequently there is no need to calculate the visible area for all three element subsets of the 5 data points. Instead we calculate the visible areas of the 5 data points and sort the data points in non-increasing order of visible area, $< d_2$,d_4,d_0,d_1,$d_3 >$, and we report the first 3 data points $< d_2$,d_4,$d_0 >$, which is the optimum result.

Consider the *MVFS* instance illustrated in Figure 5.5(b). Notice that, here the data points are separated in 2 connected components, c_1 and c_2, where c_1 contains d_0, d_1, and d_2 and c_2 contains d_3, and d_4. No point in the data space is visible simultaneously from one data point in c_1 and one data point in c_2. Consequently, calculating visible area for subsets having data points from both c_1 and c_2 is unnecessary, as there is no such triangles or elements. Instead, we calculate

visible areas for all subsets of the data points in c_1 and all subsets of the data points in c_2. Then we make 3 ($= k$) greedy choices, where at each step, we select the connected component that results in the maximum increment of the visible area and increase that component's count. The process terminates when the sum of the counts of the components reaches 3 ($= k$). Let the count of a component c is denoted by c_c. Then for each component c, we choose the subset of length c_c with maximum visible area to construct the optimum solution.

In the scenario depicted in Figure 5.5(b), for component c_1, the one, two, and three element subsets having maximum area visible region are $\{d_2\}$, $\{d_0, d_2\}$, and $\{d_0, d_1, d_2\}$ respectively. The area of the visible regions of the three subsets are 32, 61, 78 respectively. In component c_2, the one, and two element subsets having maximum area visible region are $\{d_4\}$, and $\{d_3, d_4\}$ respectively. The area of the visible regions of the two subsets are 30, and 44 respectively. Now we make 3 greedy choices as follows. Initially the counts of both components are 0, i.e., $c_{c_1} = 0$ and $c_{c_2} = 0$. In the first greedy step, we select component c_1 because it increases the visible area by 32, greater than that of c_2, which is 30. After the first greedy selection, $c_{c_1} = 1$ and $c_{c_2} = 0$. In the second greedy step, we select c_2 because it increases the visible area by 30, greater than the increment of c_1, which is 29 (=61-32). After the second greedy iteration, $c_{c_1} = 1$ and $c_{c_2} = 1$. In the third greedy step, we choose c_1, as the area increment of c_1 (61-32=29) is greater than that of c_2 (44-30=14). After the third greedy step, we have $c_{c_1} = 2$ and $c_{c_2} = 1$, and the greedy loop terminates. Finally to generate the optimum solution, we select the two element subset of c_1 with maximum area, i.e., $\{d_0, d_2\}$ and the one element subset of c_1 with maximum area, i.e., $\{d_4\}$. Thus the optimum solution is $< d_0$,d_2,$d_4 >$.

Now we present the algorithm *continuousExact* for the *MVFS* problem that incorporates the above discussed acceleration techniques to eradicate the bottlenecks of the naive solution. First we describe the routines used in the algorithm.

connectedComponent: The *connectedComponent* routine takes a set of data points, a set of obstacles, and the visibility range as input parameters and returns the overlapping components as illustrated in the Figure 5.5(b). To determine the components, first we construct a graph G. The data points in D are the nodes of G. There exists and edge between two data points if their visible regions overlap. We use the *visRegion* routine used in the algorithm *visTriangulation*

to determine the visible region of each data point. To determine whether there exists an edge between two data points, d_i and d_j, we calculate boolean intersection of all pair of triangle (t_i,t_j), where $t_i \in visRegion(d_i)$ and $t_j \in visRegion(d_j)$. There is an edge between d_i and d_j, if there exists at least one pair of triangles whose boolean intersection is not null. After constructing the graph G, we perform Depth First Search (DFS) on G to find the connected components.

bitmask: The *bitmask* routine takes a connected component c of data points as input parameter and returns a bitmap of length n whose i^{th} bit is set if $d_i \in c$, and off otherwise.

generateResult: The *generateResult* routine takes list of tuples as input and generates the optimum result of the *MVFS* problem. The routine processes the tuples as simulated in Figure 5.5(b). The routine makes k greedy choices to generate the solution. At each step, the routine considers each component and selects the component that results in maximum increase in visible area.

We explain the algorithm *continuousExact* as follows. In Lines 3-8, we create the list of elements M to remove the first bottleneck discussed previously. M is implemented as a balanced binary search tree storing (key, value) pairs. Thus we can search an element in M by its key in logarithmic time. Here, the $M[s]$ notation denotes the value of the element having key s. We construct the connected components of D (Line 9). For each component, we consider all subsets of length less than or equal to k (Lines 11-23) and populate the list of tuples R (Line 24). Finally we call the *generateResult* routine to produce the optimum solution of the *MVFS* problem.

Algorithm 3: continuousExact(O,D,r,k)

input : O,D,r,k
output: Optimum Choice of k Data Points

```
1  begin
2      L ⟵ visTriangulation(O, D, r)
3      M ⟵ ∅
4      for each triangle t ∈ L do
5          if t_s ∈ M then
6              M[t_s] ⟵ M[t_s] + t_a
7          else
8              M.insert(t_s, t_a)
9      C ⟵ connectedComponent(D, O, r)
10     R ⟵ ∅
11     for each component c ∈ C do
12         l ⟵ min(c.size(), k)
13         for i ⟵ 1 to l do
14             maxArea ⟵ −1
15             maxChoice ⟵ 0
16             for each bitmap m with i set bits in bitmask(c) do
17                 area ⟵ 0
18                 for each element e ∈ M do
19                     if ∼ m|e.key = 0 then
20                         area ⟵ area + e.value
21                 if area > maxArea then
22                     maxArea ⟵ area
23                     maxChoice ⟵ m
24             R.insert(tuple(c, i, maxArea, maxChoice))
25     generateResult(R)
26     return
```

5.2 The Discrete Exact Algorithm

In this section, we propose the *discrete exact algorithm* for the *MVFS* problem, which partitions the data space into a grid of equal sized *cell*s and considers the notion of visibility of the data space in terms of the grid cells. First we reformulate the *MVFS* problem for a grid partitioned data space (Section 5.2.1). Next we describe a projection based idea to determine visibility of cells to/from a data point (Section 5.2.2). Finally we present the discrete exact algorithm (Section 5.2.3).

5.2.1 *MVFS* in Grid Partitioned Data Space

In this section, first we describe grid partitioning and introduce the notion of visibility in a grid partitioned data space by providing some necessary definitions. Next we reformulate the *MVFS* problem according to the new idea of visibility in a discretized data space. Finally we discuss the motivation behind this formulation

We discuss grid partitioning of data space into cells as follows. In grid partitioning, we visualize the data space to be homogeneously partitioned into equal sized *cell*s as in a grid. We consider the midpoint of each cell as its representative. Now we define the visibility of a cell to/from a data point in a grid partitioned data space.

Definition 5 ***Visibility between a Cell and a Data Point****: Given a set of obstacles, and the visibility range of a facility, a cell c is defined as visible to/from a data point d, if the midpoint of c is visible to/from d. Otherwise, c is defined as non-visible to/from d.*

Figure 5.6 illustrates the idea of cell visibility by showing a sample grid partitioning around the vicinity of the data point d_3 from Figure 3.1. In the figure, the visibility range of d_3 is shown in dashed lines, the visible region of d_3 is shown in bold lines, and the visible cells from d_3 are shown in grey. According to the above definition of visibility between a cell and a data point, the cells whose midpoints are visible from d_3 are declared visible from d_3. Now we redefine the *MVFS* problem for a data space partitioned into cells.

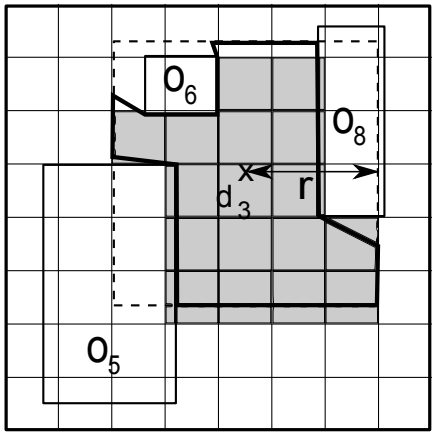


Figure 5.6: Grid Partitioning into Cells.

The Discrete MVFS Problem: Given an m-dimensional data-space R^m (m=2 or 3), the set of obstacles in the data-space, the set of n data points where a facility can be placed, D, the viewing range of a facility, an integer k ($1 \leq k \leq n$), and a grid partitioning of the data space into cells, the problem is to determine a subset S of D, $S \subseteq D$, of size k, $|S| = k$, to establish k facilities, such that, the number of cells visible from at least one data point in S is maximized. Here the notation $|.|$ stands for the cardinality of a set. We call this version the *discrete MVFS problem*.

We discuss the motivation behind this new formulation of the *MVFS* problem as follows. The continuous exact algorithm, discussed in the previous section, accurately determines the area of the visible region of data points by summing the area of the triangles in the visibility triangulation according to their visibility status. If the data points are closely located, the visibility triangulation consists of a huge number of small triangles. Consequently the construction of visibility triangulation becomes computationally expensive and the performance of the continuous exact algorithm deteriorates. Now, in Figure 5.6, notice that, the area of the visible (grey) cells is a close estimation of the area of the visible region of d_3. According to the modified problem definition, the discrete exact algorithm does not calculate the area of the visible regions. Instead, it counts the number of visible cells which serves as an estimation of the visible area of a data point. Thus the discrete exact algorithm achieves considerable speedup over the continuous exact algorithm by avoiding actual area computations. Also the number of cells within the visibility

range of a data point remains fixed irrespective of the density of the data points in the surrounding space. Thus the discrete exact algorithm performs steadily regardless of the distribution of the data points in the data space.

5.2.2 Determining Visibility of Cells using Projection

In the discrete exact algorithm, the visible region of a data point d is estimated by counting the number of cells visible from d. In this section, we describe the process of determining the set of cells visible from a data point. We outline a routine, namely *visibleCells*, that takes as input a data point d, the set of obstacles located within the visibility range of d, the visibility range of a facility, and a grid partitioning of the data space, that determines the set of cells in the grid which are visible from d. We use a projection based idea [7] [9] and perform plane sweep in each principal axis direction to determine the set of visible cells from a data point. Now we outline the methodology used in the routine *visibleCells* in brief.

To determine the set of visible cells of a data point, we need to determine the occlusion effect of obstacles around it. We perform line sweep along each principal axis direction and calculate the projections of the obstacles on each sweep line position within the visibility range to evaluate the occlusion effect. The projection of obstacles on sweep lines can be determined according to the following rules:

- Projection is calculated at each sweep line position where the midpoints of the cells are situated.
- Initially the projection is null.
- Projection at the current sweep line position is obtained by aggregating the projection of the obstacles situated between the previous and current sweep line positions and the re-projection of the previous sweep line projection on the current sweep line.

For example, consider the 2D scenario depicted in Figure 5.7, where we demonstrate the process of calculating projection of obstacles on sweep lines around data point d. Here, we simulate the line sweep along the +X direction only, because line sweep along the other 3 directions

can be performed in a similar manner. Along the +X direction, the projection of obstacles is calculated at 5 successive vertical sweep line position where the midpoints of the cells are situated, i.e., L_1 to L_5. There is no obstacle between d and L_2. Thus projections on L_1 and L_2 are null. Obstacle o_2 starts between L_2 and L_3. So projection on L_3 is AB (shown as bold line segment). Projection on L_4 is created by re-projecting the projection on L_3 (AB) on L_4, which is CD. Obstacle o_3 starts between L_4 and L_5. Thus the projection on L_5 is obtained by aggregating EF (the re-projection of CD on L_5) and GH (the projection of o_3 on L_5), which is EH. A detailed description of the process of calculating aggregated projections of obstacles on successive sweep line positions can be found in [9] and [7].

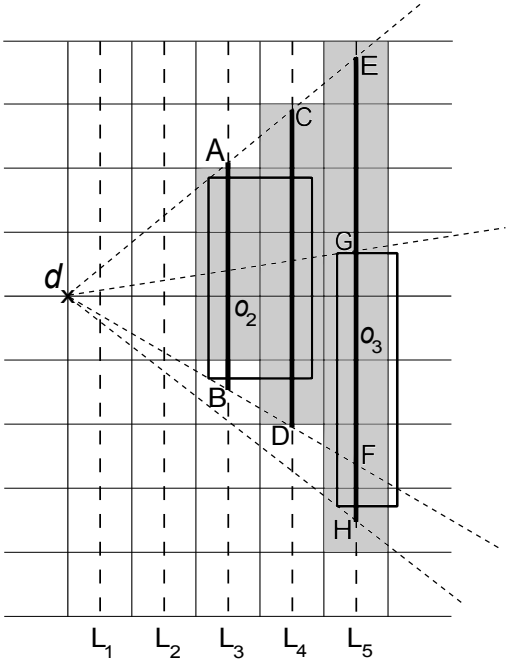


Figure 5.7: Calculating projections on successive sweep line positions.

The cells whose midpoints lie on the projections are occluded by the obstacles (gray cells in Figure 5.7) and thus not visible from d. Line sweep is performed along the other 3 principal axes to find all occluded cells around d. Finally the set of cells visible from d are the cells which are situated inside a distance r from d and which are not occluded. Thus the set of visible cells from a data point can be determined by performing line sweep and calculating aggregated projections.

5.2.3 The Algorithm

In this section, we formally present the discrete exact algorithm, which determines the optimum solution of the *MVFS* problem in a grid partitioned data space. First we define *visibility status* of cell c, denoted by c_s, an idea which is necessary for the development of the discrete exact algorithm.

Definition 6 ***Visibility Status of a Cell****: Given a set of n data points D, and a set of obstacles, the visibility status of a cell c in the grid partitioning is defined to be a bitmap of length n, where, for $0 \leq i < n$, the i^{th} bit is set if c is visible from the i^{th} data point in D, and off otherwise.*

We briefly define the basic idea underlying the discrete exact algorithm below. The discrete exact algorithm is similar to the continuous exact algorithm in the sense that in the discrete version, cells serve the same purpose as triangles do in the continuous version. Notice that, in the continuous exact algorithm, we partitioned the visible region of the data space into disjoint triangles. Triangles were the building block of the data space and all points within each triangle had the same visibility status. In the discrete exact algorithm, we partition the data space into disjoint cells. Each cell has a visibility status which equals the visible status of its midpoint. Thus cells serve as the building block of the data space in the discrete exact algorithm.

We explain the algorithm *discreteExact* briefly as follows. This algorithm takes as input the grid partitioning of the data space, G, which indicates cells size, extent of the data space etc. In Lines 2-4, we use the routine *visibleCells* outlined in Section 5.2.2 to construct the visibility status of the cells in the grid partitioning. Notice that, in the continuous exact algorithm, a call to *visTriangulation* served similar purpose by assigning visibility status to triangles. The rest of the algorithm *discreteExact* is similar to that of the algorithm *continuousExact*, with the exception that in the discrete version we calculate the count of cells, whereas in the continuous version we determine area of the triangles.

Algorithm 4: discreteExact(O,D,r,k,G)

input : O,D,r,k,G
output: Optimum Choice of k Data Points

```
1  begin
2      for i ⟵ 0 to n − 1 do
3          O_rq ⟵ rangeQuery(D_i, O, r)
4          visibleCells(D_i, O_rq, r, G)
5      M ⟵ ∅
6      for each cell c with c_s ≠ 0 do
7          if c_s ∈ M then
8              M[c_s] ⟵ M[c_s] + 1
9          else
10             M.insert(c_s, 1)
11     C ⟵ connectedComponent(D, O, r)
12     R ⟵ ∅
13     for each component c ∈ C do
14         l ⟵ min(c.size(), k)
15         for i ⟵ 1 to l do
16             maxCount ⟵ −1
17             maxChoice ⟵ 0
18             for each bitmap m with i set bits in bitmask(c) do
19                 count ⟵ 0
20                 for each element e ∈ M do
21                     if ∼ m|e.key = 0 then
22                         count ⟵ count + e.value
23                 if area > maxCount then
24                     maxCount ⟵ count
25                     maxChoice ⟵ m
26             R.insert(tuple(c, i, maxCount, maxChoice))
27     generateResult(R)
28     return
```

5.3 The Greedy Approximation Algorithm

The continuous exact algorithm and the discrete exact algorithm both determines the optimum solution of the *MVFS* problem in continuous and discrete data space respectively. But in the worst case, their time complexity is exponential to the number of data points n, as they consider all possible subsets of the n data points. Consequently, in this section, we propose a polynomial time greedy approximation algorithm for the *MVFS* problem. We reduce the *MVFS* problem to the well known weighted maximum coverage (*WMC*) problem. The *WMC* problem is known to be NP-hard. There is a greedy solution to the *WMC* problem with proven approximation ratio of $1 - \frac{1}{e}$. First we state the *WMC* problem and describe the process of reducing an instance of the discrete *MVFS* problem to an instance of the *WMC* problem (Section 5.3.1). Finally we outline the greedy approximation approach to the *WMC* problem and propose a greedy approximation algorithm for the discrete *MVFS* problem by adopting the greedy approach (Section 5.3.2).

5.3.1 Reduction to Weighted Maximum Cover Problem

First we state the well known weighted maximum coverage problem.

The Weighted Maximum Coverage Problem: In the weighted maximum coverage problem, an integer k and a collection of n sets S=S_0, S_1, ...S_{n-1} are given. Each element of the set has an associated weight. The objective is to find a subset $S' \subseteq S$, such that $|S'| \leq k$ and sum of the weights of the elements covered by S' is maximized. Here, the notation $|.|$ stands for the cardinality of a set.

We describe the process of reduction from the discrete *MVFS* problem to the *WMC* problem as follows. Consider an instance of the discrete *MVFS* problem. The set of cells visible from the n data points can be denoted by S_0, S_1, ...S_{n-1}. The weight of each cell is 1. The objective is to maximize the weighted coverage of k sets among the n sets.

We describe another way of reducing the discrete *MVFS* problem to the *WMC* problem below. Consider the contents of the map of elements M in the algorithm *discreteExact* after the execution of Line 10. M contains (key,value) pairs, where the key s is a distinct visibility status and the corresponding value is the number of cells having the visibility status s. Now, given an

instance of the discrete *MVFS* problem, for each (key,value) pair in M, we create an element e in the *WMC* instance, where e.key is the identifier of an element and e.value is the weight of the element. For each element e, if the i^{th} bit of e.key is set, we place e in the i^{th} subset S_i. Thus we populate S_0 through S_{n-1}.

5.3.2 The Algorithm

First we discuss a greedy approach to the *WMC* problem. The *WMC* problem is known to be NP-hard. The best known approximation algorithm for the *WMC* problem is a greedy algorithm that at each step chooses the set that maximizes the sum of the weights of the uncovered elements. This greedy algorithm has been proved to be the best-possible polynomial time approximation algorithm for the weighted maximum coverage problem [46] [47] and has an approximation ratio of $1 - \frac{1}{e}$.

A similar greedy approach for the discrete *MVFS* problem is to select, at each step, the data point that turns the maximum number of cells from still non-visible to visible. Now we present the greedy approximation algorithm *greedy* which implements the above strategy to generate an approximate solution of the discrete *MVFS* problem. In this algorithm, after constructing the map of elements M, in Lines 11-14, we populate the subsets S_0 through S_{n-1}. Here, e.key[i] denotes the i^{th} bit of e.key. The loop in Lines 16-23, makes a greedy choices at each of the k iterations. At each iteration, the *findMaxS* routine in Line 17 returns the index of the subset with maximum sum of weight of the elements. In Lines 19-22, we remove the elements of the newly found subset from all the subsets so that the subsets only contain the still uncovered elements at the beginning of the next iteration.

Algorithm 5: greedy(O,D,r,k,G)

input : O,D,r,k,G
output: Greedy Choice of k Data Points

```
1  begin
2      for i ⟵ 0 to n − 1 do
3          O_rq ⟵ rangeQuery(D_i, O, r)
4          visibleCells(D_i, O_rq, r, G)
5      M ⟵ ∅
6      for each cell c with c_s ≠ 0 do
7          if c_s ∈ M then
8              M[c_s] ⟵ M[c_s] + 1
9          else
10             M.insert(c_s, 1)
11     for each element e ∈ M do
12         for i ⟵ 0 to n − 1 do
13             if e.key[i] ≠ 0 then
14                 S_i.insert(e)
15     R ⟵ ∅
16     while k ≠ 0 do
17         index ⟵ findMaxS()
18         R.insert(index)
19         for i ⟵ 0 to S_index.size − 1 do
20             for j ⟵ 0 to n − 1 do
21                 if S_j.isFound(S_index[i]) then
22                     S_j.remove(S_index[i])
23         k ⟵ k − 1
24     return R
```

Chapter 6

Disk Resident Algorithms

A real dataset modeling a building complex or a city area usually contains a huge number of obstacles. If the size of the dataset is too large to fit in main memory, the main memory based algorithms issue disk access requests to fetch data from the persistent storage. This results in high IO overhead and consequently the performance of the main memory based algorithms degrades. To reduce the IO overhead in case of large datasets, in this chapter, we propose several disk resident algorithm for the discrete *MVFS* problem. The disk resident algorithms use a persistent spatial data structure, *R-tree*, where the obstacles are stored. The R-tree reduces the number of disk accesses by indexing the obstacles according to their spatial properties, such that closely situated objects in the model are located close to each other in the disk.

We discuss the organization of this chapter as follows. In Section 6.1, we present a straightforward disk resident version of the algorithm *greedy* described in Chapter 5.3. In Section 6.2, we propose a heuristic driven best first search based technique that reduces the IO cost by avoiding redundant issues of range query. To farther improve the performance of the best first strategy, in Section 6.3, we formulate an approach based on batch processing, where we group the data points that are in close proximity of each other into clusters and process each cluster of data points together instead of processing each data point individually.

6.1 The Naive Greedy Algorithm

In this section, we present a naive disk resident algorithm for the discrete *MVFS* problem. First we identify the bottleneck of the main memory based greedy algorithm that causes high IO overhead. Then we present the disk resident straightforward greedy algorithm.

Consider Line 3 of the algorithm *greedy* of Chapter 5.3. Here a range query is issued for each data point d to retrieve the obstacles situated within the visibility range of d. The *rangeQuery* routine traverses the obstacle set and returns the obstacles intersecting or falling within the visibility range of d. For a huge obstacle set, which is too large to fit in main memory, the range query routine causes disk access requests to retrieve all the obstacles from the disk. Thus the n calls to the routine *rangeQuery* incurs huge IO overhead for a large disk resident obstacle set. This is the bottleneck of the main memory based greedy algorithm.

We present the algorithm *naiveGreedy* below, that reduces the IO overhead by speeding up the range query routine. In this algorithm, we assume that the obstacles in O are indexed in an R-tree, T, a persistent spatial data structure, that can process spatial queries quickly on large set of spatial objects. It is not required to build T each time an *MVFS* query arrives as T is persistent. In the algorithm *naiveGreedy*, we perform one range query for each data point and store the results in an auxiliary data structure Z (Lines 3-5). Z contains (data point, obstacle set) pairs, (d,O), where O is the result of a range query on d. We assume that the size of Z is small enough to fit in main memory. We pass Z as a parameter to the algorithm *greedy* instead of the obstacle set O (Line 6). During the execution of *greedy*, we skip the routine *rangeQuery* as the results are already available in Z.

Algorithm 6: naiveGreedy(T,D,r,k,G)

input : T,D,r,k,G
output: Greedy Choice of k Data Points

```
1 begin
2     Z ⟵ ∅
3     for each data point d ∈ D do
4         O_rq ⟵ T.rangeQuery(d, r)
5         Z.insert(d, O_rq)
6     return greedy(Z, D, r, k, G)
```

6.2 The Best First Algorithm

The main drawback of the naive greedy approach discussed in the previous section is that it issues n range queries, one for each data point, on the R-tree irrespective of the value of k. In this section, we propose a heuristic driven method based on best first search technique that enables us to make the k greedy choices without issuing range queries for all data points in D. First, we discuss some preliminary ideas necessary for the development of the algorithm (Section 6.2.1). Finally, we present the algorithm *bestFirst*, which issues less range queries than the naive greedy algorithm (Section 6.2.2).

6.2.1 Preliminaries

In this section, we introduce an auxiliary data structure, provide necessary definitions, and describe the heuristic. In the algorithm *bestFirst*, we use a matrix, which stores one boolean value for each cell in the grid partitioning. The boolean value corresponding to a cell c indicates whether c visible from any previously chosen data point, as we make the greedy choices of data points. For a cell which is visible from any previously selected data point, the boolean value stored in the matrix is 0, and otherwise the value is 1. We call this matrix of boolean values, the *Visibility Matrix*. The visibility matrix is denoted by V. The value in V corresponding to a cell c is denoted by $V[c]$.

We define the event of *transition* as follows. Consider the event in which a cell, c, that is visible from no previously chosen data point, becomes visible after a greedy choice (i.e., the placement of a new facility). In this event, $V[c]$ changes from 1 to 0. We call this event a transition. Note that, during each of the k iterations of a greedy algorithm, we choose the data point that causes maximum number of transitions.

We describe the heuristic which drives the best first search below. Each data point in D has a heuristic value. While making the greedy choices, we expand the data points according to their heuristic values. The *heuristic value* of a data point d is the sum of the visibility matrix values of all cells having midpoints within the visibility range of d. The heuristic value of a data point d is denoted by d_h. Note that, according to the above definition, the heuristic value of a data point d

indicates the number of still non-visible cells within the visibility range of d.

6.2.2 The Algorithm

In this section, we present the algorithm *bestFirst*, which uses greedy best first search technique to make k greedy choices. At each of the k iterations, we select the data point that results in maximum number of transitions. First we describe the functions of the routines and data structures used in the algorithm *bestFirst*. Then we explain the algorithm. Finally we simulate the algorithm with an illustrative example.

Routines and Data Structures

calculateHeuristic: The *calculateHeuristic* routine takes a data point, the visibility range of a facility, and the visibility matrix as input parameter and returns the heuristic value of the data point. The routine calculates the sum of the visibility matrix values of all cells situated within the visibility range of the data point and returns the sum.

transitionCount: The *transitionCount* routine takes a data point, the set of obstacles situated within the range of the data point, and the visibility matrix as input and returns the number of transitions that occur if a facility is placed at the data point. In other words, it returns the number of those cells, which are visible from the data point and whose visibility matrix value is 1. The *transitionCount* routine uses the projection based idea described in Chapter 5.2.2 to determine the set of cells C that visible from the data point and returns the number of cells in C that have visibility matrix value of 1. Note that, the routine *transitionCount* is similar to the routine *visibleCells* used in the discrete exact algorithm.

updateV: The *updateV* routine takes a data point, and the set of obstacles within the visibility range of the data point as input and updates the visibility matrix as follows. The visibility matrix value of all cells that are visible from the data point is set to 0. Thus we maintain whether a cell is visible from any greedily chosen data point. The implementation of this routine is similar to the implementation of the routine *visibleCells* or *transitionCount*.

V: V is the visibility matrix discussed in the previous section. It is implemented as a 2D

Algorithm 7: bestFirst(T,D,r,k,G,V)

input : T,D,r,k,G,V
output: Greedy Choice of k Data Points

```
1  begin
2      R ⟵ ∅
3      Z ⟵ ∅
4      while k ≠ 0 do
5          Q ⟵ ∅
6          for each data point d ∈ D do
7              d.key ⟵ calculateHeuristic(d, r, V)
8              d.state ⟵ OPEN
9              Q.push(d)
10         while true do
11             d ⟵ Q.pop()
12             if d.state = CLOSED then
13                 R.insert(d)
14                 Q.remove(d)
15                 updateV(d, Z[d])
16                 break
17             if !Z.find(d) then
18                 O_rq ⟵ T.rangeQuery(d, r)
19                 Z.insert(d, O_rq)
20             else
21                 O_rq ⟵ Z[d]
22             d.key ⟵ transitionCount(d, O_rq, V)
23             d.state ⟵ CLOSED
24             Q.push(d)
25         k ⟵ k − 1
26     return R
```

matrix. In the algorithm *bestFirst*, we assume that V can fit in main memory and V is cached while *MVFS* queries are being processed. Consequently, V is passed as a parameter to the algorithm *bestFirst*. When V is passed as parameter to an *MVFS* query, it stores 0 for all the cells whose midpoints lie inside an obstacle. This is because those cells can not be visible from any data point. The cells whose midpoints do not lie inside any obstacle, have a visibility matrix value of 1.

Z: Z is a map that contains (key, value) pairs, (d,O), where O is the set of obstacles situated within the visibility range of d. The notation $Z[d]$ indicates the obstacle set corresponding to data point d.

Q: Q is a max priority queue that stores data points according to their heuristic value. In a pop operation, Q returns the data point with maximum key or heuristic value.

Explanation

We explain the algorithm *bestFirst* as follows. First we make two key observations as follows:

- Placing a facility at a data point having high heuristic value is more likely to cause more transitions than placing a facility at a data point having low heuristic value.
- The number of transitions caused by placing a facility at a data point d is upper bounded by the heuristic value of d, d_h.

At each iteration of the greedy loop spanning from Line 4 to 25, we select the data point that causes maximum number of transitions. According to the above two observations, we expand the data points in non-decreasing order of their heuristic value. At the start of each iteration, we calculate the heuristic value of each data point and put all the data points in a max priority queue Q according to their heuristic value (Lines 6-9). All the data points are divided into two disjoint sets, the *OPEN* set and the *CLOSED* set. The data points for which range query is yet not issued belong to the *OPEN* set. On the other hand, the data points whose range query is already issued belong to the *CLOSED* set and their key values are set to their respective transition counts. Initially all the data points are in the *OPEN* set.

For each data d point being popped from Q, we search the map Z to check whether range query result of d is available. If not, we issue a range query on T to determine the set of obstacles located within the visibility range of d and save the results in Z (Lines 17-21). Note that, we use Z to avoid redundant issues of range query. We update the key value of d by its transition count and add it to the *CLOSED* set (Lines 22-23). As soon as a data point d belonging to the *CLOSED* set is retrieved from Q, the algorithm select d as the greedy choice, updates the visibility matrix accordingly, and proceeds to the next iteration (Lines 12-16).

Note that, the algorithm *bestFirst* never issues range query for a data point d, if the heuristic value of d is less than or equal to the transition count of any data point belonging to the *CLOSED* set. Thus the greedy best first technique avoids issuing range queries for all the data points in D and achieves better performance than the naive greedy approach.

Simulation

Figure 6.1 provides an illustrative example simulating the algorithm *bestFirst* for a scene with 3 data points, 5 obstacles, and $k = 2$. The visibility range of the data points are shown in dashed lines. The visible region from the data points are shown in bold lines. The cells having visibility matrix value of 0 and 1 are shown in white and grey respectively. Initially the cells having midpoints within an obstacle are white and all other cells are grey (Figure 6.1(a)). At the beginning of the first iteration, the heuristic values of the data points are as follows: $d_{0h} = 21$, $d_{1h} = 15$, and $d_{2h} = 23$. The algorithm *bestFirst* expands the data point with maximum heuristic value, d_2, issues a range query for d_2, and determines its transition count to be 22. As the transition count of d_2 is greater than the heuristic value of d_0 and d_1, the first greedy choice d_2 is made without issuing range query for d_0 and d_1. Then the algorithm updates V for d_2. The state of the visibility matrix after the end of the first iteration is shown in Figure 6.1(b). At the beginning of the second iteration, the heuristic values of the data points are as follows: $d_{0h} = 17$, and $d_{1h} = 7$. Thus the algorithm expands d_0 and determine its transition count to be 17. As the transition count of d_0 is greater than the heuristic value of d_1, the second greedy choice d_0 is made without issuing range query for d_1 and the algorithm terminates. The final state of V is shown in Figure 6.1(c).

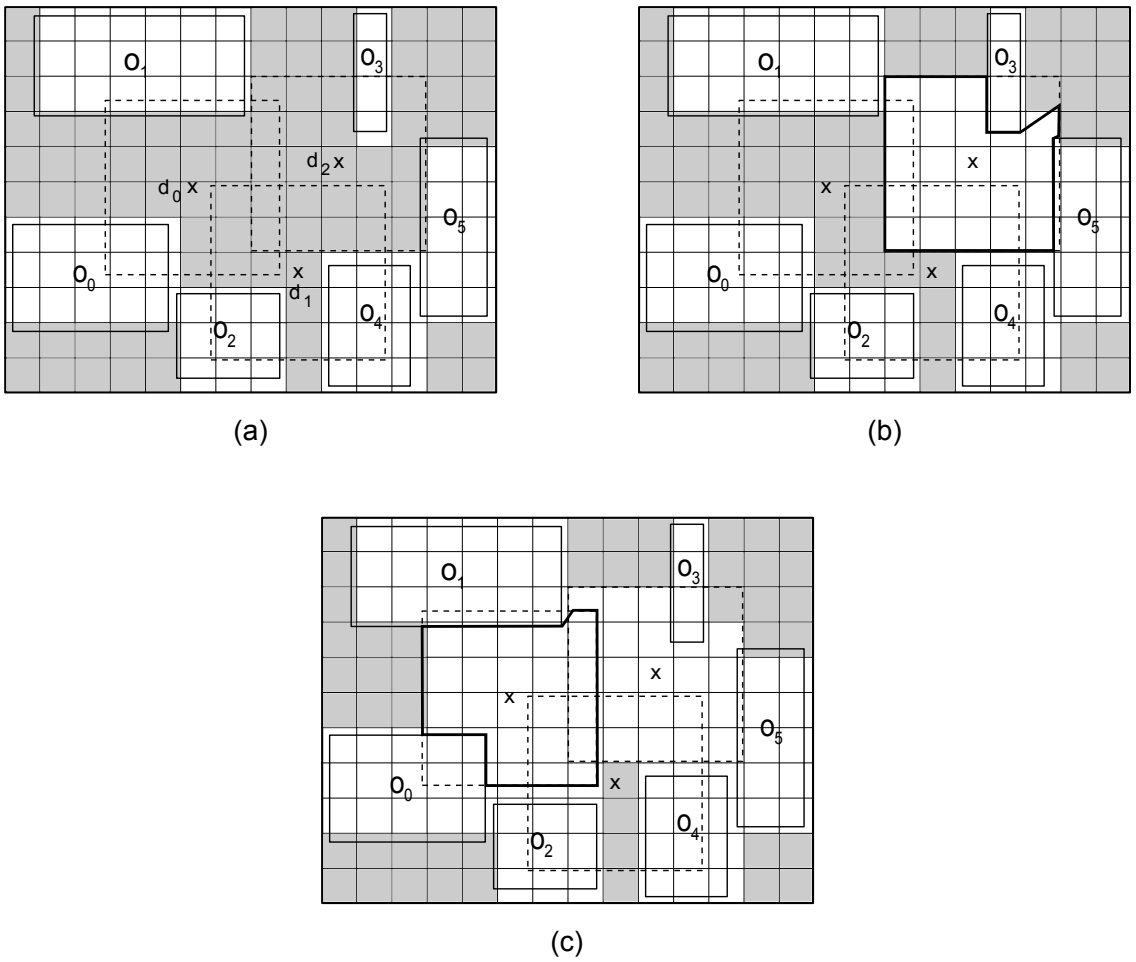


Figure 6.1: Simulation of best first greedy approach.

6.3 The Batch Processing Algorithm

In this section, we present the algorithm *batchProcessing*, that uses the idea of clustering to gain farther speedup over the best first search based algorithm discussed in the previous section. First we identify a drawback of the algorithm *bestFirst*. Then we briefly outline the clustering technique we have adopted in this algorithm. Finally we propose the algorithm *batchProcessing*.

Although the best first search based algorithm reduces the IO overhead considerably in comparison with the naive greedy algorithm for small values of k, there is an issue that the algorithm *bestFirst* fails to address. Consider two closely situated facilities. The region covered by the two facilities in the data space may overlap each other. Consequently issuing range queries for the data points is redundant, as their combined coverage is close to their individual coverage.

To address the above discussed issue, we adopt the idea of batch processing. Instead of issuing separate range queries for each data points, we group data points situated spatially close to each other into clusters and process each cluster of data points together as a batch.

We discuss the clustering algorithm we have used to group spatially closely located data points together as follows. We adopt a simplified version of the widely used distance based hierarchical clustering method to cluster the data points of D in the data space. A hierarchical clustering method is usually dependent on some parameters and use some cutoff value of the parameter as the terminating condition. We provide a cutoff value of the cluster radius to the clustering algorithm. The value is denoted by c_{size}. Consequently we limit the maximum distance between two data points located within the same cluster component. Note that higher values of c_{size} results in fewer number of components and thus the size/extent of the components is larger. So we can adjust the size and number of components by varying the cluster radius parameter, c_{size}. We do not describe the implementation details of the clustering algorithm here for brevity.

We present the algorithm *batchProcessing* as follows. This algorithm is similar to the algorithm *bestFirst* to a great extent except for some differences as we mention now. First we call the routine *hierarchicalClustering* to determine the cluster components (Line 1). In the algorithm *batchProcessing*, instead of storing one tuple for each data point, we store one tuple for each cluster in the priority queue Q (Lines 7-11). The key value for each cluster c in the *OPEN* (*CLOSED*) set is set to the maximum heuristic value (transition count) of all the data points belonging to c. The clusters are retrieved from Q in non-increasing order of the key values. For each retrieved cluster c, we issue one range query for c (Lines 19-23). Then the routine *maxTransitionCount* is called to determine the optimum data point within c and the corresponding transition count (Line 24). We add an additional field named *point* in the tuple to store the data point in a cluster that causes the maximum transition count. The implementation of *maxTransitionCount* is trickier than that of *transitionCount*, as all the data points belonging to a cluster are to be considered. The detailed process of implementing *maxTransitionCount* efficiently is discussed later in this section.

Note that, the algorithm *batchProcessing*, range queries are issued per cluster, instead of per

Algorithm 8: batchProcessing(T,D,r,k,G,V,c_{size})

input : T,D,r,k,G,V,c_{size}
output: Greedy Choice of k Data Points

begin
  $C \longleftarrow hierarchicalClustering(D, c_{size})$
  $R \longleftarrow \emptyset$
  $Z \longleftarrow \emptyset$
  while $k \neq 0$ **do**
    $Q \longleftarrow \emptyset$
    for *each cluster* $c \in C$ **do**
      $c.key \longleftarrow heuristic(c, r, V)$
      $c.point \longleftarrow NULL$
      $c.status \longleftarrow OPEN$
      $Q.push(c)$
    while *true* **do**
      $c \longleftarrow Q.pop()$
      if $c.status = CLOSED$ **then**
        $R.insert(c.point)$
        $c.remove(c.point)$
        $updateV(c.point, Z[c])$
        break
      if $!Z.find(c)$ **then**
        $O_{rq} \longleftarrow T.rangeQuery(c, r)$
        $Z.insert(c, O_{rq})$
      else
        $O_{rq} \longleftarrow Z[c]$
      $(c.key, c.point) \leftarrow maxTransitionCount(c, O_{rq}, V)$
      $c.status \longleftarrow CLOSED$
      $Q.push(c)$
    $k \longleftarrow k - 1$
  return R

data point as in the algorithm *bestFirst*. Thus if the data points are located in closely knitted groups, the IO cost reduces. Also observe that the size of the priority queue Q is also smaller in comparison with the algorithm *bestFirst*. Consequently the algorithm *batchProcessing* gains computational speedup.

We discuss the efficient implementation of the routine *maxTransitionCount* below. The process of determining transition count of one data point described in the previous section (the routine *transitionCount*) can be adopted to find the data point in a cluster that causes the maximum transition count within a cluster component of data points. We make the following two observations:

- For all data points in a cluster, there is a finite number of sweep line positions where the projections are calculated.
- The same set of obstacles is used for all the data points in a cluster.

Consequently, we perform one line sweep in each principal axis direction, and maintain a list of projections (one for each data point), which are updated at each sweep line position. Thus, the transition count of all the data points are calculated by performing 4 line sweeps (6 plane sweeps, in case of 3D) and hence the data point causing the maximum transition count.

Chapter 7

Handling 3D Scenarios

The methodology we have presented thus far applies to 2D scenarios. But the ideas used in our solution are transferrable to 3D data space. In this chapter, we discuss how the 2D techniques described earlier can be modified to handle 3D scenarios. The methodology we have used to solve the *MVFS* problem are fundamentally different for continuous and discrete data space. We discuss both separately as follows.

7.1 Handling a Continuous 3D Scene

First we discuss how to answer the 3D *MVFS* qeury in a continuous space. To represent the visible region of a data point in 3D space, we use a set of tetrahedrons, as opposed to a set of triangles in 2D space. Thus tetrahedrons are the building block of the visible region in a continuous 3D scene. The set of tetrahedrons representing the visible region of a data point can be determined by performing plane sweep along principal axes, similar to the rotational line sweep in 2D scenario described in Chapter 5.1.1. To form a partition of the space using tetrahedrons, we implement the boolean intersection and subtraction of tetrahedrons, and use an incremental algorithm similar to the algorithm *visTriangulation* described in Chapter 5.1.2. Thus the region visible from the set of data points D is partitioned into disjoint tetrahedrons. To determine the volume of visible region from a set of data points, we take the sum of volumes of tetrahedrons according to their visibility status.

7.2 Handling a Discrete 3D Scene

We discuss how to solve the 3D *MVFS* problem in a discretized data space as follows. We use a 3D grid to partition the data space. Thus the cells are cube shaped. The main difference between the solution for discrete 2D and discrete 3D scenarios lies at the implementation of the routine *visibleCells* (or the routine *transitionCount*). The routine uses a projection based strategy as depicted in Figure 5.7 to determine the set of cells (or number of cells in the routine *transitionCount*) visible from a data point in the presence of obstacles. In the routine we perform line sweep and calculate projections of the obstacles as line segments on the sweep lines. In case of a 3D scenario, plane sweep is performed and projections of the obstacles are planar regions which are calculated on sweep planes. We adopt research works by Durand et al. [9] and Rabban et al. [7], which explain in details how projection of 3D obstacles can be calculated efficiently on successive sweep planes.

Chapter 8

Extensions

In this chapter, we propose farther generalizations to the discrete *MVFS* problem and discuss how our techniques can be modified to provide solutions for these variants. Thus far we have assumed that the facilities have a 360 degree field of view. In Section 8.1, we solve the *MVFS* problem assuming that the facilities have limited field of view. In Section 8.2, we address the case in which each cell in the grid partitioning has a value specifying its visibility preference and propose solutions for this case. Throughout this work, we have adopted a binary notion of visibility. A point in the data space is either visible or non-visible from a data point. In Section 8.3, we discuss how our solution can be modified to work for quantitative notion of visibility. Finally, in Section 8.4 we solve the *MVFS* variant, where the set of data points D is not provided and thus we are free to establish k facilities anywhere in the data space.

8.1 Limited FoV *MVFS*

In our work thus far, we have assumed that each facility has a field of view of 360 degree. This assumption is quite unrealistic because generally a facility (i.e., a camera) has a limited field of view. In this section, we consider a variant of the *MVFS* problem, where we are provided with the field of view of each facility, denoted by an angle A, in the problem definition and along with the set of k data points, we also determine the viewing directions of the facilities positioned at the k data points that maximizes the number of cells visible from the k data points. We call this variant

the *limitedFovMVFS* problem. In this section, first we illustrate the methodology to determine the optimum viewing direction from a given facility location separately for 2D (Section 8.1.1) and 3D (Section 8.1.2) scenarios. Next we describe how our solution to the *MVFS* problem can be adjusted to solve the above defined *limitedFovMVFS* problem (Section 8.1.3).

To solve the *limitedFovMVFS* problem, first we consider the problem where we are given a data point d, a set of cells C situated within the visibility range d, and the field of view of the facility A, and the goal is to determine the *optimum viewing direction* of the facility, i.e. the viewing direction for which the number of cells in C falling within the field of view of a facility located at d is maximized. This subproblem works as a building block to solve the *limitedFovMVFS* problem. Note that, in this subproblem, C does not necessarily contain all the cells within the viewing range of a facility positioned at d, which will be clarified in Section 8.1.3. Our proposed techniques to solve this subproblem are fundamentally different for 2D and 3D scenarios. We describe both in the following sections.

8.1.1 Optimum Viewing Direction in 2D

In this section, we describe the process of determining the optimum viewing direction from a data point in a 2D scenario. Note that, for a given data point, there are infinitely many potential viewing directions along which the facility can be oriented. Considering all possible viewing directions is not feasible. Now we develop an idea to limit the number of potential viewing directions that we consider. First we provide some necessary definitions.

For a cell c to be visible from a facility f placed at data point d, under the assumption that f has a limited field of view, A, the angle between the line joining d and the midpoint of c, and the viewing direction of f must be less than or equal to $\frac{A}{2}$. Also, we denote the most clockwise (counter-clockwise) ray within the field of view of a facility to be the *left line* (*right line*) of the facility (Figure 8.1). Now we present the following proposition.

Proposition 1: Given a facility f placed at data point d and set of cells C within the viewing range of f in a 2D scenario, there exists an optimum viewing direction of f for which at least one cell in C has its midpoint on the left line of f.

Proof: Suppose, there is an optimum viewing direction of f such that midpoint of no cell in C lies on the left line of f. Now, we rotate the viewing direction of f counter-clockwise, until its left line reaches the midpoint of some cell in C (Figure 8.1). This change in the viewing direction does not sacrifice the optimality, because no cell that was previously visible becomes non-visible during the rotation. Also, no cell becomes visible from non-visible during the process because it will contradict the assumption that the initial viewing direction was optimum. Consequently the final viewing direction of f retains the optimality and has a cell midpoint on its left line. □

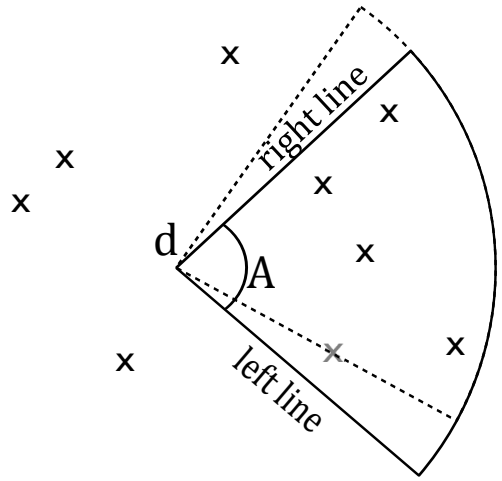


Figure 8.1: Cell midpoints are cross marked. The initial (final) position is shown in solid (dashed) arc. The viewing direction is rotated counter-clockwise until the left line reaches a cell midpoint (light gray).

Proposition 1 allows us to restrict the size of the set of potential viewing directions from infinity to the number of cells in C. Now we present the algorithm *optDirection2D* that determines the optimum viewing direction from a data point in a 2D scene. Here the routine *angle* used in Line 5 and Line 8, takes as input two vectors in 2D and returns the angle between the two vectors. In the algorithm *optDirection2D*, for cell c, the midpoint of c is denoted by $c.m$. For each cell c in C, we consider a viewing direction dir such that the midpoint of c lies on the left line of the facility (Line 5) and determine the number of cells falling inside the field of view (Lines 7-9). According to Proposition 1, the viewing direction resulting in the maximum number of cells will be an optimum direction.

Note that, the asymptotic time complexity of the algorithm *optDirection2D* is O($|C|^2$), where

Algorithm 9: optDirection2d(d,C,A)

input : d,C,A
output: optimum viewing direction

```
1  begin
2      optDir ⟵ NULL
3      maxCount ⟵ 0
4      for c ∈ C do
5          dir ⟵ angle(c.m − d, +Xaxis) + A/2
6          count ⟵ 0
7          for c ∈ C do
8              if angle(c.m − d, dir) ≤ A/2 then
9                  count ⟵ count + 1
10         if maxCount < count then
11             maxCount ⟵ count
12             optDir ⟵ dir
13     return optDir
```

$|C|$ denotes the cardinality of the set C. There are several ways to improve the performance of *optDirection2D*. Sorting the cell midpoints angularly and preprocessing the number of cells falling inside increasing intervals or using the *rotating callipers method* will yield a time complexity of O($|C|lg|C|$).

8.1.2 Optimum Viewing Direction in 3D

In this section, we present the process of determining the optimum viewing direction from a facility f in a 3D scenario, given the data point d where the facility is located, a set of cells C within the viewing range of f, and the field of view of f, A. First we describe how the 3D scene is visualized and provide some necessary definitions and notations. Then we develop a proposition that allows us to restrict the set of potential viewing directions. Finally we present the algorithm that determines the optimum viewing direction from a facility location in 3D.

We consider a sphere of unit radius centered at the d and construct rays joining d and the midpoints of the cells in C to project each cell midpoint on the unit sphere (Figure 8.2). Thus the field of view of a f can be modeled as a circle on the surface of the unit sphere, where the circle

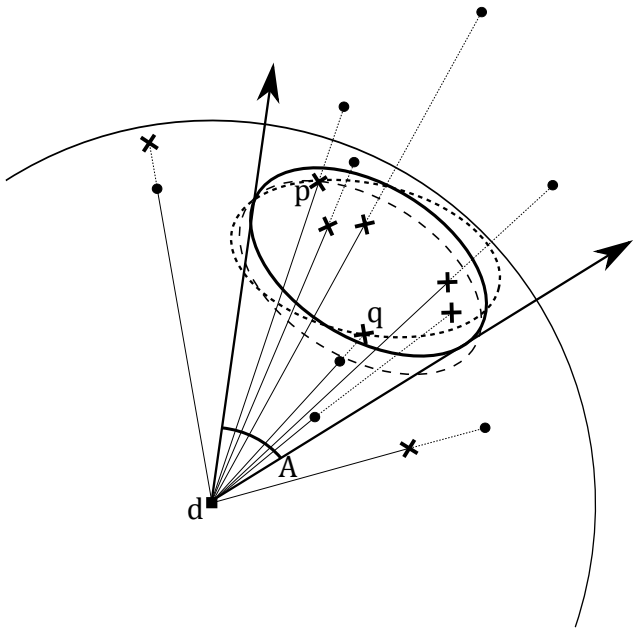


Figure 8.2: Part of the unit sphere centered at d. Cell midpoints are shown in dots and their projection on the unit sphere are cross marked.

is centered at the point of intersection of the unit sphere and the ray from d along the viewing direction of f. We call this circle the *viewing circle* of f. The projection of the midpoints of the cells which fall inside the field of view of f, will be inside the or on the perimeter of the viewing circle of f. If the projection of the midpoint of a cell c lies on the perimeter of the viewing circle of facility f, we call c to be on the *perimeter* of f. The unit vector from d to the midpoint of a cell c is denoted by $c.v$.

Similar to the 2D case, there are infinitely many potential viewing directions along which the facility can be oriented in a 3D scenario. We use the following proposition to limit the number of viewing directions that we consider while finding the optimum viewing direction.

Proposition 2: Given a facility f placed at data point d and set of cells C within the viewing range of f in a 3D scenario, there exists an optimum viewing direction of f for which at least two cells in C are on the perimeter of f.

Proof: Suppose, there is an optimum viewing direction of f such that no cell in C is on the perimeter of f (the solid circle on the sphere in Figure 8.2). First, we move the viewing circle of f in any arbitrary direction, until its perimeter reaches the projection m of the midpoint of some

cell in C (the solid circle is moved until it reaches point p in Figure 8.2; the long dashed circle is the new position of the viewing circle). Now, we rotate the viewing circle with respect to m, until a second cell midpoint is on the perimeter (the long dashed circle is rotated around p until it reaches point q in Figure 8.2; the short dashed circle is the resultant viewing circle). This change in the position of the viewing circle does not sacrifice the optimality, because projection of no cell midpoint goes out of the viewing circle during the movement and rotation. Also projection of no cell midpoint enters the viewing circle during this process, because it will contradict the assumption that the initial position of the viewing circle was optimum. Consequently the final position of the viewing circle defines a viewing direction of f through its center that retains the optimality and has at least two cells on its perimeter. □

Algorithm 10: optDirection3d(d,C,A)

input : d,C,A
output: optimum viewing direction

```
1  begin
2      optDir ⟵ NULL
3      maxCount ⟵ 0
4      for each pair (c1, c2) from C do
5          if angle(c1.v, c2.v) > A then
6              continue
7          v(t) ⟵ n(c1.v + c2.v).cost + n(c1.vXc2.v).sint
8          (t1, t2) ⟵ solve[v(t).(c1.v) = cos(A/2)]
9          count1 ⟵ 0
10         count2 ⟵ 0
11         for c ∈ C do
12             if angle(v(t1), c.v) ≤ A/2 then
13                 count1 ⟵ count1 + 1
14             if angle(v(t2), c.v) ≤ A/2 then
15                 count2 ⟵ count2 + 1
16         if maxCount < count1 then
17             (maxCount, optDir) ⟵ (count1, v(t1))
18         if maxCount < count2 then
19             (maxCount, optDir) ⟵ (count2, v(t2))
20     return optDir
```

We present the algorithm *optDirection3D* that determines the optimum viewing direction from a facility located in a 3D scene. In *optDirection3D*, it is assumed that the cell midpoints are projected on different points on the surface of the unit sphere, which might not be the case. If projections of two or more cell midpoints coincide, we can treat them as one cell whose count equals the number of cells that coincide. According to Proposition 2, we consider all possible pair of cells and construct viewing directions (Lines 4-8) such that both cells lie on the perimeter of the viewing circle. We choose the viewing direction for which the number of cells whose midpoint is projected within the viewing circle is maximized (Lines 11-19).

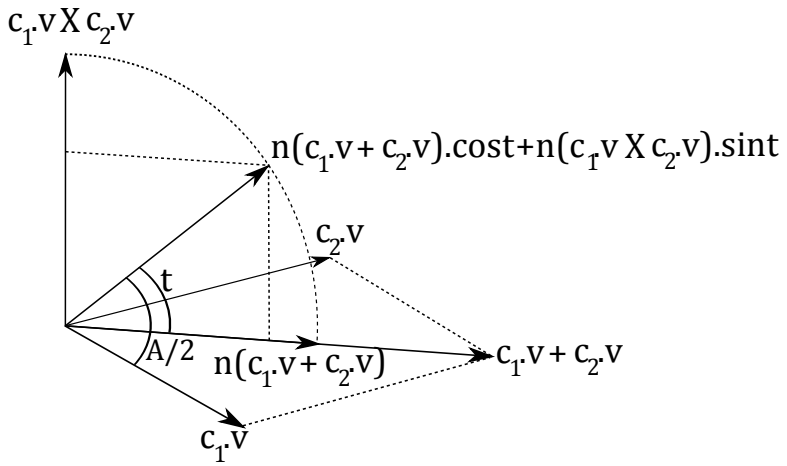


Figure 8.3: Determining the vector creating an angle $\frac{A}{2}$ with vectors $c_1.v$ and $c_2.v$.

Now we elaborate on the process of constructing a viewing direction defined by unit vector v, such that two given cells c_1 and c_2 lie on the perimeter. Such a viewing direction has the property that both $c_1.v$ and $c_2.v$ creates the same angle $\frac{A}{2}$ with v. Thus v is on a plane halfway between $c_1.v$ and $c_2.v$. This halfway plane is spanned by the vectors n($c_1.v$+$c_2.v$) and n($c_1.v$X$c_2.v$) as both create the same angle with v and perpendicular to each other. Here n(v) denote the unit vector along the vector v and the X operator indicates vector cross product. Any vector on this midway plane can be represented by the following parametric equation:

$$v(t) = n(c_1.v + c_2.v).cost + (c_1.vXc_2.v).sint, 0^\circ \leq t < 360^\circ$$

The value(s) of t for which $v(t)$ creates an angle of $\frac{A}{2}^\circ$ with $c_1.v$ (and $c_2.v$) can be determined by

solving the following equation:

$$v(t).(c_1.v) = cos(\frac{A}{2})$$

Here, the first . stands for vector dot product. Refer to Figure 8.3 for a better understanding.

Note that, the asymptotic time complexity of the algorithm *optDirection3D* is O($|C|^3$). We can improve the performance by precalculating the angles the unit vectors create with the 3 principal axes and using those angles to reduce the search space while finding the cells having projection inside the viewing circle.

8.1.3 The Algorithm

We discuss how our solution to the basic *MVFS* problem can be modified to answer the *limitedFovMVFS* query as follows. The algorithm *limitedFovMVFS* is very similar to the algorithm *bestFirst*. We can solve the *limitedFovMVFS* problem by making two adjustments in the algorithm *bestFirst*. So, instead of stating the algorithm, we mention the adjustments we make in the algorithm *bestFirst* as follows:

- In Line 7 of the algorithm *bestFirst*, we calculate the heuristic value of each data point. In the *limitedFovMVFS*, we modify the heuristic calculation as follows. To determine the heuristic value of a data point d in *limitedFovMVFS*, we pass a set of cells C to the *calculateHeuristic* routine instead of V, where C contains all cells within the viewing range of a facility placed at d that have visibility matrix value of 1, i.e., the cells whose midpoints are not inside any obstacle and which are not visible from any previously selected data point. Then we determine the optimum viewing direction using the above algorithm (*optDirection2D* or *optDirection3D*) and find the corresponding number of cells, s. s is set as the heuristic value of d, as it provides an upper bound on the transition count that a facility placed at d can achieve.

- In Line 22 of the algorithm *bestFirst*, we calculate the transition count of a data point. In the *limitedFovMVFS*, we modify the process of counting the number of transitions as follows. To determine the transition count of a data point d in *limitedFovMVFS*, we pass

a set of cells C to the *transitionCount* routine instead of V, where C contains all cells visible from d that have a visibility matrix value of 1, i.e., the cells which can undergo transition. Then we determine the optimum viewing direction using the above algorithm (*optDirection2D* or *optDirection3D*) and find the corresponding number of cells, s. s is set as the transition count of d.

8.2 Preferential *MVFS*

In our primary formulation of the discrete *MVFS* problem, we treated all cells in the grid partition in the same manner, i.e., we assigned unit weight to each cell and counted the number of cells to determine the visibility coverage. But in a practical scenario, the data space may have different visibility preferences at different locations. For example, some spots in the data space might require more rigorous surveillance than others. Consequently, in this section, we address a variant of the discrete *MVFS* problem, where each cell in the grid partitioning has a preference value specifying its priority. Higher preference value refers to higher visibility preference of the corresponding cell and vice versa. We call this version the *preferential MVFS* problem. The preferential *MVFS* problem is defined below:

The Preferential MVFS Problem: Given an m-dimensional data-space R^m (m=2 or 3), the set of obstacles in the data-space, the set of n data points where a facility can be placed, D, the viewing range of a facility, a grid partitioning of the data space G, a matrix P storing the preference values of the cells in G, and an integer k ($1 \leq k \leq n$), the problem is to determine a subset S of D, $S \subseteq D$, of size k, $|S| = k$, to establish k facilities, such that, the sum of the preference values of the cells visible from at least one data point in S is maximized. Here the notation $|.|$ stands for the cardinality of a set.

We describe how the algorithm *bestFirst* can be modified to solve the *preferentialMVFS* problem below. We make the following two modifications. In the *preferentialMVFS* problem, the heuristic value of a data point equals the sum of preference values of the cells having visibility matrix value 1. The transition count of a data point d in the *preferentialMVFS* problem returns the sum of the preference values of the cells that undergo transition because of the greedy selection

of d. Given the matrix P containing preference values, the two modifications can be incorporated in the algorithm *bestFirst*.

8.3 Quantitative *MVFS*

Throughout this work, we have assumed binary notion of visibility, i.e., a cell is either visible or non-visible. But usually the visibility of a point is inversely proportional to the distance of the point and the facility, i.e., a camera. Quantitative approach to visibility addresses this important issue and assigns a real value to each cell considering the cell's distance from the facility rather than assigning either 1 or 0 indicating visibility and non-visibility. In this section we describe how we incorporate the notion of quantitative visibility in our solution. We extend the idea of preferential *MVFS* described in the previous section to build the solution that assumes the quantitative notion of visibility.

In the binary model of visibility, as in the previous section, we treated the preference value of each cell as a whole, and added the preference values of the cells to determine the heuristic value or the transition count. To incorporate the idea of quantitative visibility, while calculating the heuristic value or the transition count, we multiply the preference value of a cell with a factor inversely proportional to the distance between the cell and the data point. Consider a cell c having midpoint m and preference value c_p. Let, the viewing range of a facility located at d be r. Then, the *quantitative preference value* of c with respect to the facility at d equals $c_p * max(0, 1 - \frac{dis(m,d)}{r})$. Thus the preference value of a cell is multiplied by a factor inversely proportional to distance to calculate its contribution to the heuristic value or the transition count. The visibility of a cell is determined by the quantitative preference value of the closest facility from which its visible. Thus, in the greedy algorithm, the selection of a new facility can increase the quantitative preference value of a cell, if it is situated closer to the cell than any other previously established facilities. This is how we incorporate the notion of quantitative visibility in our solution.

8.4 Unrestricted *MVFS*

In this section, we consider a variant of the basic *MVFS* problem, where no set of data points is provided and we are free to establish a new facility anywhere in the data-space. The goal is to find k optimum locations that maximize the visibility coverage.

In this variation, there are infinitely many candidate locations where a facility can be placed. We limit the set of potential locations where a facility can be established to the set of lattice points of the grid partitioning. Thus if the grid has $m * n$ cells, there are $(m + 1) * (n + 1)$ lattice points which constitute the set of data points. Instead of using distance based hierarchical clustering method to determine the cluster components, we treat contiguous $l * l$ blocks of lattice points as the cluster components. Thus the algorithm *batchProcessing* can be modified to solve the unrestricted *MVFS* problem.

Chapter 9

Experimental Evaluation

In this chapter, we present and analyze the experimental results. The values of the parameters used to evaluate the performance of our proposed solution vary noticeably for main memory based algorithms and disk resident algorithms. Consequently we describe the experimental results for main memory based algorithms and disk resident algorithms separately. In Section 9.1, we describe the experimental setup. In Section 9.2, we will discuss the empirical results of the three main memory based algorithms proposed in Chapter 5. In Section 9.3,we will describe the experimental results of the three disk resident algorithms described in Chapter 6.

9.1 Experimental Setup

Our experiments are based on synthetic 2D datasets. In the datasets, we generate obstacles of varying size uniformly all over the extent of the data space. The obstacles are non-overlapping axis aligned 2D rectangles. All obstacles are indexed in an R-tree, with the disk page size fixed at 1KB. We generate separate datasets with sparse and dense distribution of data points. The data points do not lie inside the obstacles. In the sparse distribution, data points are generated uniformly all over the data space. In the dense distribution, the data points are generated in a closely knitted group such that their visible regions overlap with each other. Each experiment is performed for sparse and dense distribution of data points. For each experiment, we have evaluated the results for 20 randomly generated positions of the data points and reported the

average. The algorithms are implemented in C++ and the experiments are conducted on a core i5 2.40 GHz PC with 3GB RAM, running Microsoft Windows 7.

9.2 Empirical Evaluation of Main Memory based Algorithms

We propose three main memory based solutions to the *MVFS* problem, namely, the *continuous exact algorithm* (Chapter 5.1), the *discrete exact algorithm* (Chapter 5.2), and the *greedy approximation algorithm* (Chapter 5.3). We investigate the performance of the three proposed algorithms by varying the number of data points, the number of obstacles, the range of the camera, and the cell size. The ranges and default values of the relevant parameters are listed in Table 9.1. The default values of the parameters are set to their median values.

Table 9.1: Parameters for Main Memory based Algorithms

Parameter	**Range**	**Default Value**
Number of Obstacles	500, 1000, 1500, 2000, 2500,3000	1500
Number of Data Points	8, 12, 16, 20, 24, 32	16
Distribution of Data Points	Sparse, Dense	
Camera Range	10, 15, 20, 25, 30	20
Length of Cell Edge	0.5, 1, 2, 4, 8	2
Data-Space Size		1000*1000

9.2.1 Effect of Number of Data Points

The effect of the number of data points on the total processing time of the three algorithms is shown in Figure 9.1. Here the value of k is not mentioned, because for each n, the output for all possible ks are generated, i.e., $1 \leq k \leq n$, and the total time is reported.

The results show that the greedy algorithms run much faster that the continuous and discrete algorithms. The continuous and the discrete algorithm are exact algorithms that consider all subsets of data points in the worst case, thus their time complexity is exponential to n. But the

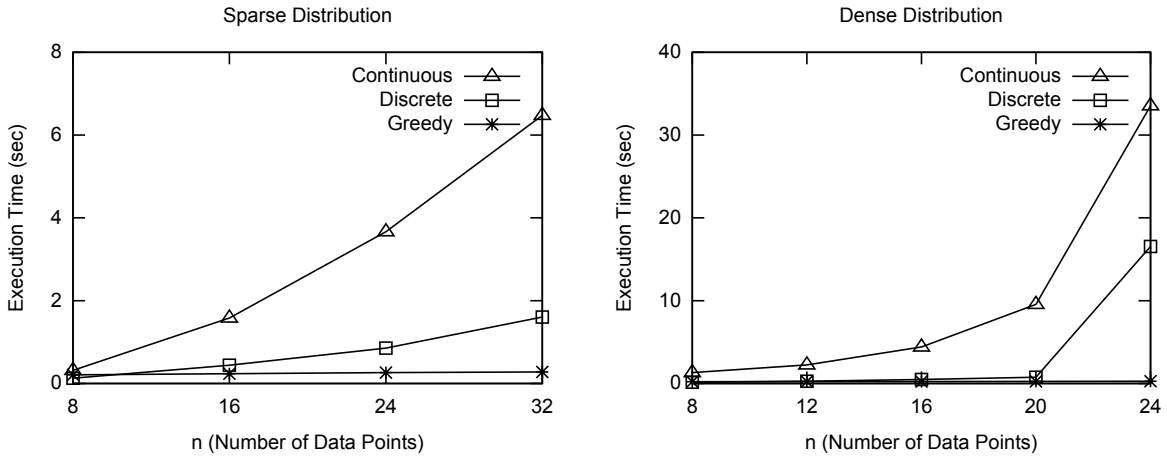


Figure 9.1: Number of Data Points vs Execution Time.

greedy algorithms takes time polynomial to n. Thus it achieves significant speedup over the two exact algorithms.

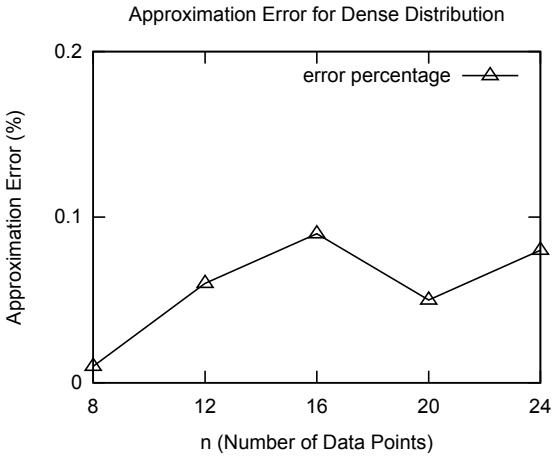


Figure 9.2: Greedy Approximation Error.

As discussed earlier, the exact algorithms perform better for sparse distribution of the data points than dense distribution of the data points. Because a dense distribution of data points results in more overlaps between the visible regions of nearby facilities. For sparse distribution there is hardly any overlap between the visible regions of the data points. Thus with an increase in the number of data points, the processing time does not increase as fast as for a dense distribution of data points. On the other hand, the processing time for dense distribution of data points increases exponentially with increasing n for the exact algorithm.

In the experiments for the sparse distribution, the greedy algorithm always finds the optimum result due to little or no overlap between visible regions of data points. Thus the greedy approximation ratio is 0. The greedy approximation ratio for dense distributions of data points with varying n is shown in Figure 9.2. Note that, the maximum average approximation error is less than 0.1%. Consequently we conclude that the greedy approximation algorithm generates near optimal solution for the *MVFS* problem and also performs much faster than the exact algorithms.

9.2.2 Effect of Number of Obstacles

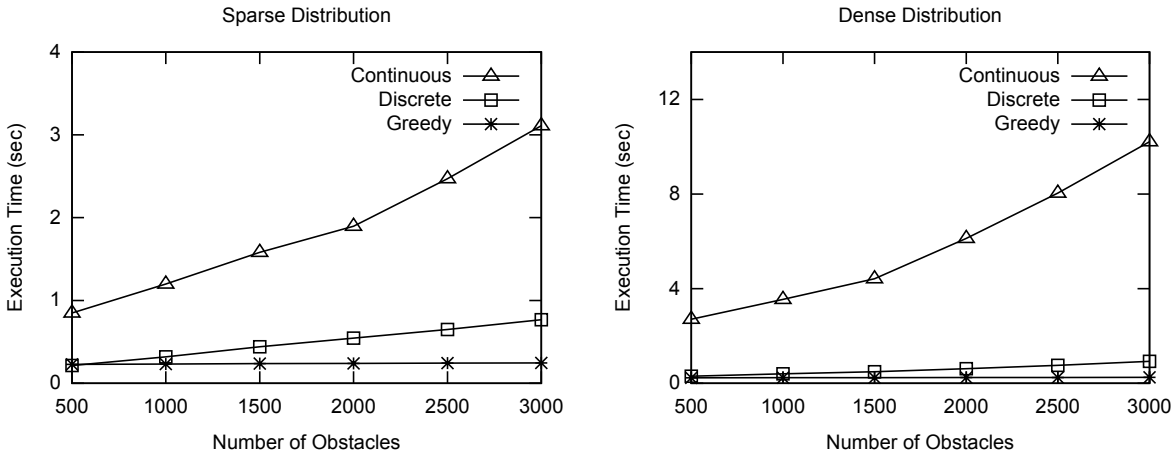


Figure 9.3: Number of Obstacles vs Execution Time.

Figure 9.3 shows the effect of number of obstacles on processing time. In general, the processing the for the continuous algorithms grows rapidly with an increase in the number of obstacles. Because if there are more obstacles in the data space, the visible region from a data point will be represented by more triangles. Performing intersection between these triangles to generate all possible bitmap areas results in a lot of small triangles. Thus it increases the processing time of the continuous algorithm.

9.2.3 Effect of Camera Range and Cell Size

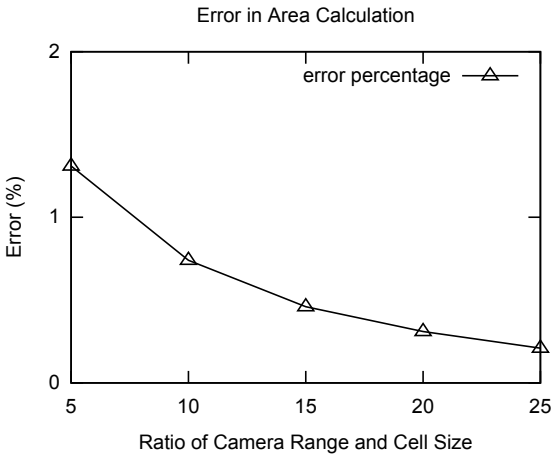


Figure 9.4: Area Calculation Error.

From Figure 5.6, note that the area of the cells visible from d_3 is a close estimation of the area of the visible region of d_3. When we use the discrete *MVFS* formulation, we avoid actual area calculation and use the number of visible cells as an estimation of the actual area. If we reduce the size of the cells, we get a closer approximation of the actual area. But using smaller cells has a drawback. If the cells grow smaller, the number of cells increases, which in turn increases the processing time of discrete and greedy algorithms. Consequently we settle for a cell size such that it is not too big to give poor approximation of the visible area and it is not too small to increase the computational overhead of the discrete and greedy algorithms. We consider the cell size with respect to the camera range. Figure 9.4 describes how the area calculation error changes with varying ratio of camera range and cell size. Note that the average error is less than 1% for a ratio of 10. As a result, we set the default value of the cell size to 2 and default value of camera range to 20. Thus the area error for the discrete *MVFS* formulation is expected to be less than 1%.

9.3 Empirical Evaluation of Disk Resident Algorithms

We propose three disk resident algorithms for the *MVFS* problem, namely, the *naive greedy algorithm* (Chapter 6.1), the *best first algorithm* (Chapter 6.2), and the *batch processing algorithm* (Chapter 6.3). We investigate the performance of the three proposed algorithms by varying the number of obstacles, number of data points, and k. The ranges and default values of the relevant parameters are listed in Table 9.2. In the disk resident algorithms, the obstacles are indexed in an R-tree. As R-tree is a persistent data structure, the range query operation on the R-tree issues IO requests on the disk. Disk accesses are usually slow and consequently incur considerable IO overhead. So we report the IO processing time along with the total processing time to compare the computational cost and the IO cost.

Table 9.2: Parameters for Disk Resident Algorithms

Parameter	**Range**	**Default Value**
Number of Obstacles	20K, 40K, 60K, 80K, 100K	50K
Number of Data Points	32, 64, 96, 128, 160, 212, 256	256
k	1, 2, 4, 8, 16, 32, 64, 128, 256	256(=*n*)
Distribution of Data Points	Sparse, Dense	
Camera Range		40
Length of Cell Edge		4
Data Space Size		10000*10000

9.3.1 Effect of Number of Obstacles

Figure 9.5 shows the effect of the number of obstacles on total processing time and IO time for both sparse and dense distribution of data points. The number of obstacles are varied from 20K to 100K, with 20K increments. The experiments conducted for $n = 256$ and $k = 256$.

The results show that the total processing time increases with increasing number of obstacles. As the number of obstacles increases, the processing time of a range query increases. Conse-

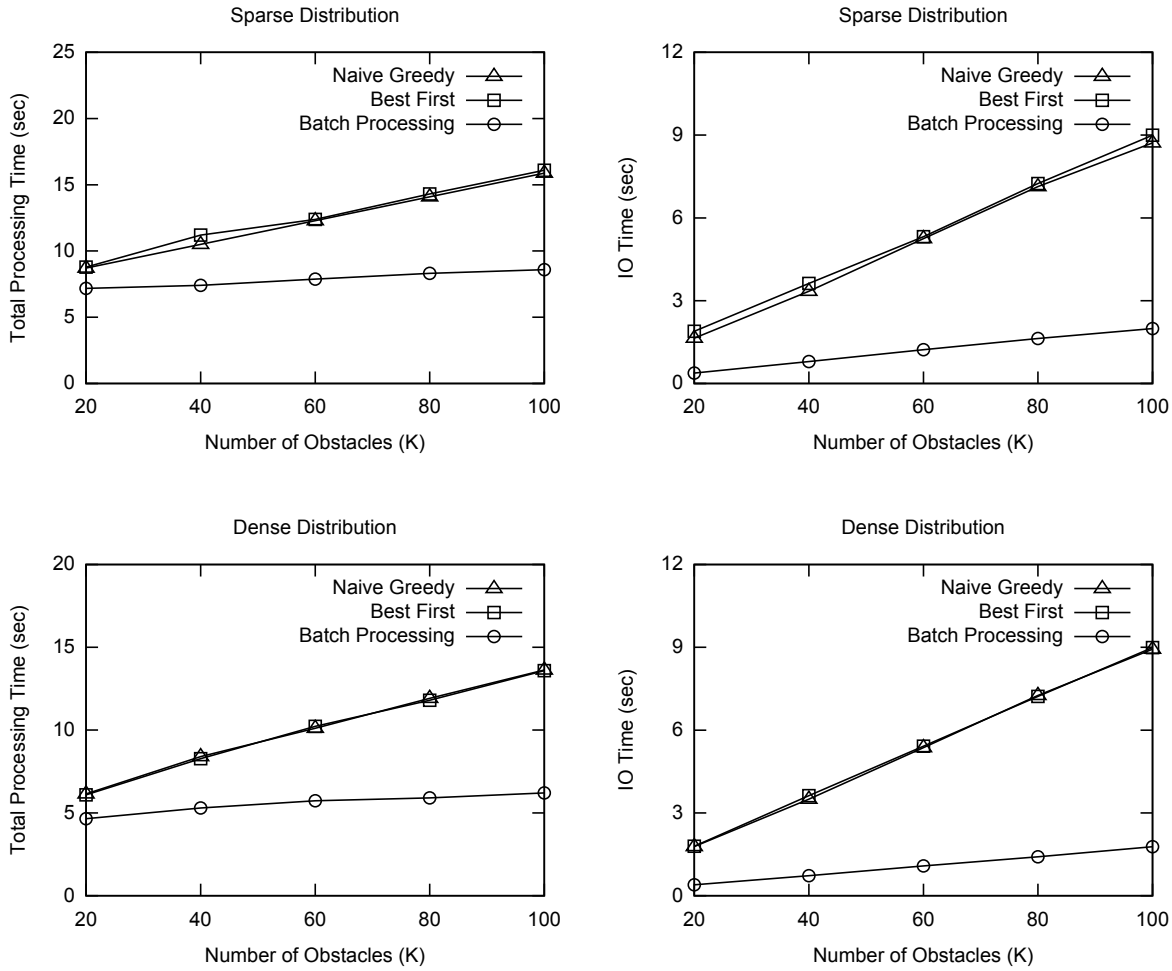


Figure 9.5: Number of Obstacles vs Total Processing Time and IO Time.

quently the IO time increases. With an increase in the number of obstacles, the complexity of the calculating the heuristic value and the transition count of a data point increases. Consequently the computational cost also increases. The figure demonstrate that the total processing time is dominated by the IO time. The computational cost does not affect the total processing time much.

The difference in total processing time in the three algorithms is largely due to the difference in IO time. The number of range queries issued by both naive greedy and best first algorithm is equal to the number of data points, i.e., 256, while the number of range queries issued by the batch processing algorithm is equal to the number of cluster components. The computational cost of the three algorithms are almost the same.

Also note that the computational cost is greater in sparse distribution of data points in comparison to dense distributions. This is because the visible region of all data points spans less number of cells in dense distribution than sparse distribution. The IO cost does not vary with the distribution of data points because the number of range queries issued remains fixed irrespective of the distribution of data points.

9.3.2 Effect of Number of Data Points

Figure 9.6 shows the effect of the number of data points on total processing time and IO time for both sparse and dense distribution of data points. The number of data points are varied from 32 to 256, with an increment of 32 data points. The value of k is set to be equal to n.

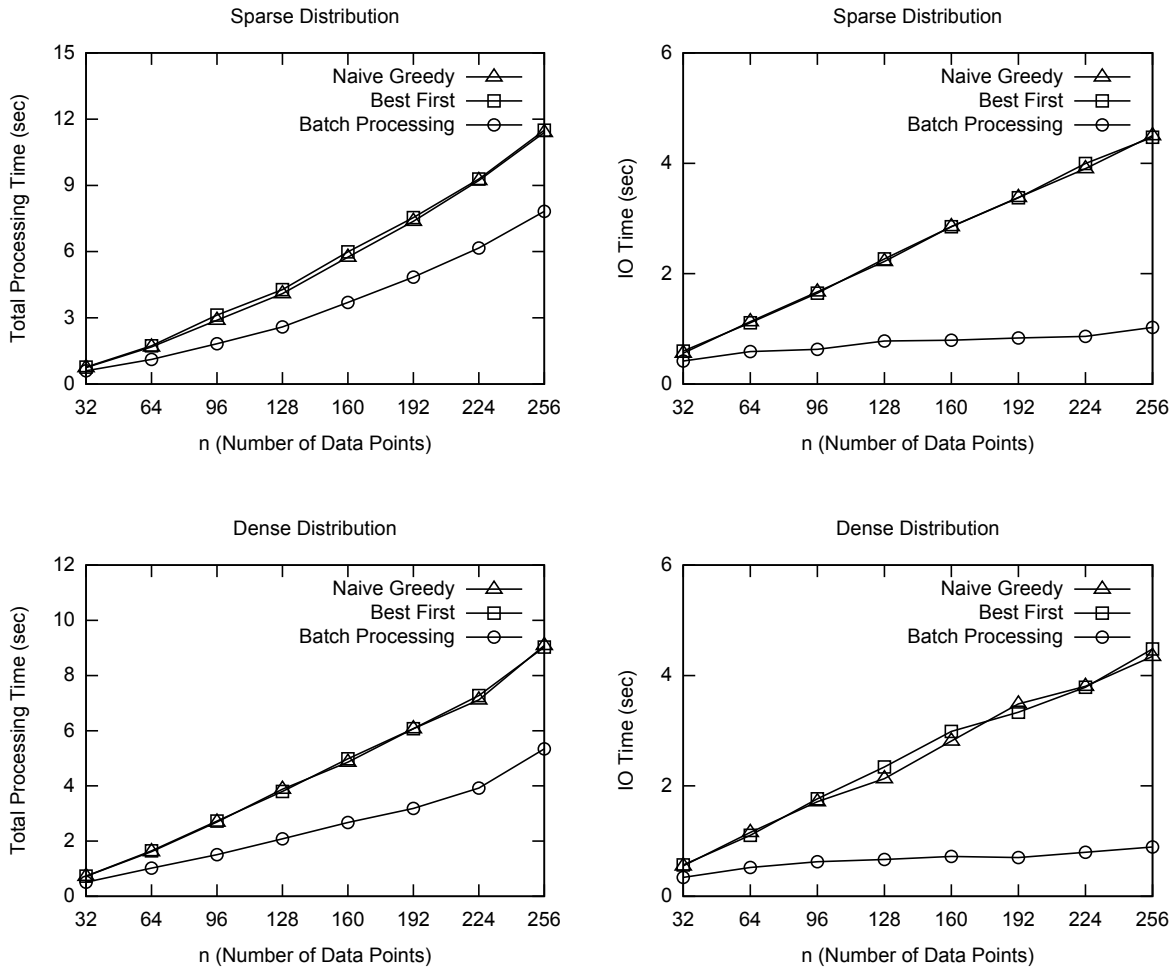


Figure 9.6: Number of Data Points vs Total Processing Time and IO Time.

The graphs show that the total processing time increases with increasing number of data points. As the number of data points n increases, the number of greedy iterations and the number of cells in the visible region of data points increase. Consequently the computational cost increases. With increasing n, the number of range query issues increases, which accounts for the increase in IO time. Note that the IO time varies linearly with the number of data points in the naive greedy and the best first algorithm, because the number of range queries issued in these algorithms is equal to the number of data points. In the batch processing algorithm, the number of range queries is proportional to the number of cluster components, which increases with increasing number of data points. Thus the total processing time is dominated by the computational cost.

The difference in the total processing time in the three algorithms is explained by the reasons stated in the previous section. Also the effect of distribution of data point on total processing time and IO time with varying n is similar to that of the previous section for similar reasons.

9.3.3 Effect of *k*

Figure 9.7 shows the effect of k on total processing time and the number of range queries issued for both sparse and dense distribution of data points. For this experiment, we set $n = 256$ and vary k exponentially from 1 to 256.

The experimental results show that the total processing time increases with the value of k. As K increases, the number of greedy iterations increases which is responsible for an increase in the computational cost. But the difference in total processing time of the three algorithms with increasing k is largely dependent on the number of range queries, i.e., IO time. In the naive greedy algorithm the number of range queries does not depend on k. It always issues n (=256) range queries. Thus the total processing time of the naive greedy algorithm is dominated by the computational cost. In the best first algorithm, the number of range queries increases with k and reaches a maximum of 256. In the batch processing algorithm, the number of range queries increases with k and reaches a maximum value equal to the number of cluster components. This is explained by the fact that the best first algorithm (the batch processing algorithm) uses

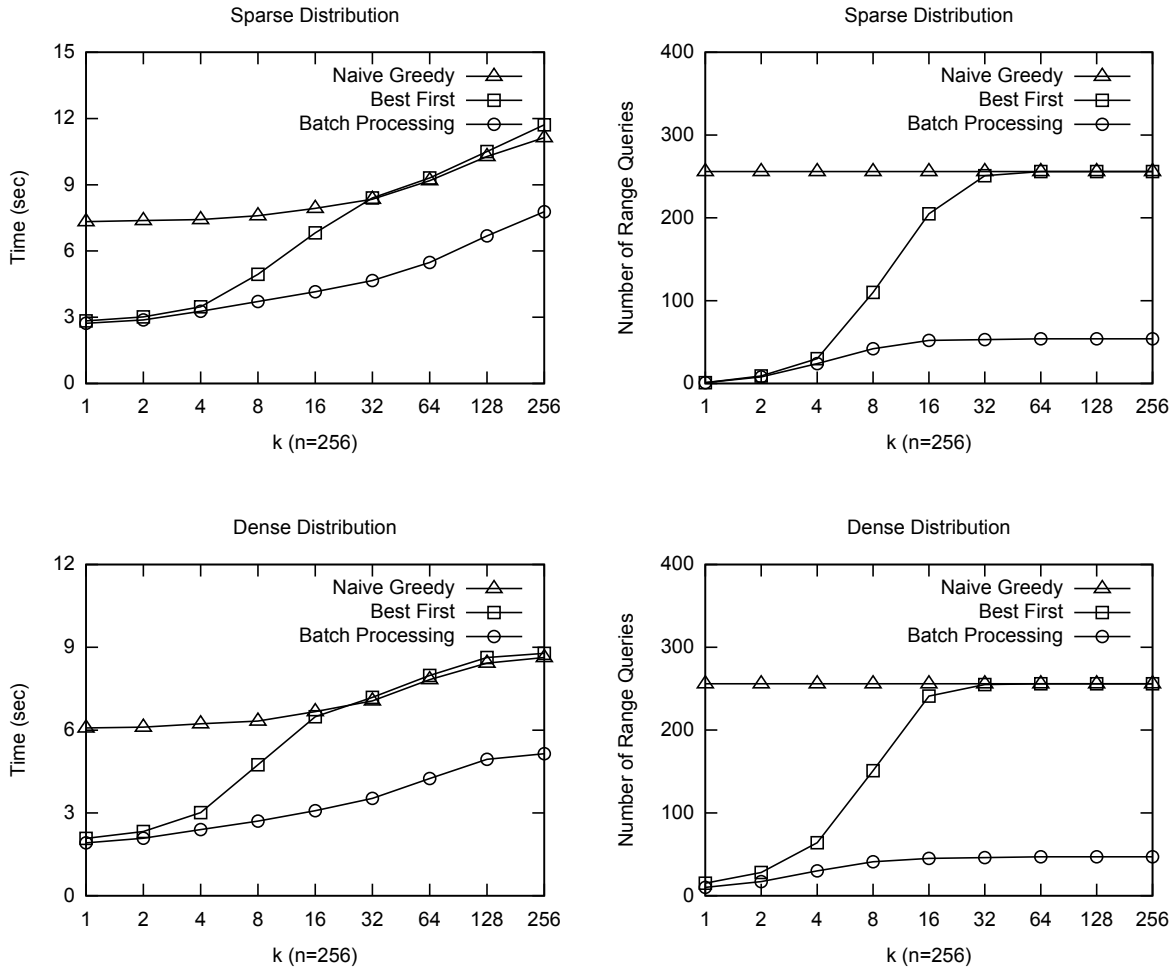


Figure 9.7: k vs Total Processing Time and Number of Range Queries.

a heuristic to avoid issuing range query for all the data points (cluster components) for small values of k.

Chapter 10

Conclusion

With the increasing availability of 3D model of cities, buildings etc., a new sector of research involving visibility queries in the presence of obstacles is on the rise. In this work, we have introduced a visibility query, named the Maximum Visibility Facility Selection *MVFS* query. In the *MVFS* query, we are given a set of obstacles, a set of n locations where a facility can be placed, the visibility range of the facility, and an integer k, we select k locations from the given n locations to establish facilities that maximizes the aggregated visibility coverage of the surrounding data space.

We have proposed exact algorithms for the *MVFS* problem for both continuous and discrete data space. For a continuous data space, we have developed a triangulation based method to compute the actual area/volume covered by any subset of the data points and determined the subset that results in maximum coverage by using some acceleration techniques. For the discrete *MVFS* problem, we have partitioned the data space into a grid and estimated the area/volume visible from any subset of data points by counting grid cells. We have also outlined a greedy approximation algorithm for the *MVFS* problem that at each greedy step selects the data point that results in maximum increase of the visible region.

To deal with datasets containing a large number of obstacles, we have developed several disk resident algorithms for the *MVFS* problem which can be applied in case the set of obstacles is too large to fit in main memory. To deal with the huge obstacle set, we have indexed the obstacles in a spatial data structure, R-tree, that can efficiently process spatial queries. We have used heuristic

driven best first search and clustering techniques to farther improve the performance of the disk resident algorithms. We have also addressed several variants of the *MVFS* problem so that our proposed algorithms can be applied to more generalized and realistic scenarios.

We have conducted extensive empirical studies to analyze the performance of our proposed algorithms. The experimental results have demonstrated that the error in area/volume in a discretized data space is less than 1% for a camera range to cell size ratio of 10, and thus can be ignored. The empirical analysis has also demonstrated that the greedy algorithm runs orders of magnitude faster that the exact algorithms and the approximation error of the greedy algorithm is on average less than 0.1%. In case of the disk resident algorithms, the experiments have shown that the acceleration techniques considerably reduce the IO overhead in comparison with a naive algorithm.

Bibliography

[1] E. Hörster and R. Lienhart, "On the optimal placement of multiple visual sensors," in *Proceedings of the 4th ACM International Workshop on Video Surveillance and Sensor Networks*, ser. VSSN '06. New York, NY, USA: ACM, 2006, pp. 111–120. [Online]. Available: http://doi.acm.org/10.1145/1178782.1178800

[2] B. Debaque, R. Jedidi, and D. Prevost, "Optimal video camera network deployment to support security monitoring," in *2009 12th International Conference on Information Fusion*, July 2009, pp. 1730–1736.

[3] S. Hanoun, A. Bhatti, D. Creighton, S. Nahavandi, P. Crothers, and C. G. Esparza, "Target coverage in camera networks for manufacturing workplaces," *J. Intell. Manuf.*, vol. 27, no. 6, pp. 1221–1235, Dec. 2016. [Online]. Available: https://doi.org/10.1007/s10845-014-0946-z

[4] S. Masud, F. M. Choudhury, M. E. Ali, and S. Nutanong, "Maximum visibility queries in spatial databases," in *29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013*, 2013, pp. 637–648. [Online]. Available: https://doi.org/10.1109/ICDE.2013.6544862

[5] C. M. R. Haider, A. Arman, M. E. Ali, and F. M. Choudhury, "Continuous maximum visibility query for a moving target," in *Databases Theory and Applications - 27th Australasian Database Conference, ADC 2016, Sydney, NSW, September 28-29, 2016, Proceedings*, 2016, pp. 82–94. [Online]. Available: https://doi.org/10.1007/978-3-319-46922-5_7

[6] F. M. Choudhury, M. E. Ali, S. Masud, S. Nath, and I. E. Rabban, "Scalable visibility color map construction in spatial databases," *Inf. Syst.*, vol. 42, pp. 89–106, 2014. [Online]. Available: https://doi.org/10.1016/j.is.2013.12.002

[7] I. E. Rabban, K. Abdullah, M. E. Ali, and M. A. Cheema, "Visibility color map for a fixed or moving target in spatial databases," in *Advances in Spatial and Temporal Databases - 14th International Symposium, SSTD 2015, Hong Kong, China, August 26-28, 2015. Proceedings*, 2015, pp. 197–215. [Online]. Available: https://doi.org/10.1007/978-3-319-22363-6_11

[8] A. Arman, K. Abdullah, I. E. Rabban, and M. E. Ali, "Indvizcmap: Visibility color map in an indoor 3d space," in *Proceedings of the Eighth ACM SIGSPATIAL International Workshop on Indoor Spatial Awareness*, ser. ISA '16. New York, NY, USA: ACM, 2016, pp. 47–50. [Online]. Available: http://doi.acm.org/10.1145/3005422.3005430

[9] F. Durand, G. Drettakis, J. Thollot, and C. Puech, "Conservative visibility preprocessing using extended projections," in *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 2000, pp. 239–248.

[10] T. Asano, "An efficient algorithm for finding the visibility polygon for a polygonal region with holes," *IEICE Transactions*, vol. 68, no. 9, pp. 557–559, 1985.

[11] F. Durand, "A multidisciplinary survey of visibility," 2000.

[12] E. Horster and R. Lienhart, "Approximating optimal visual sensor placement," in *2006 IEEE International Conference on Multimedia and Expo*, July 2006, pp. 1257–1260.

[13] U. M. Erdem and S. Sclaroff, "Automated camera layout to satisfy task-specific and floor plan-specific coverage requirements," *Comput. Vis. Image Underst.*, vol. 103, no. 3, pp. 156–169, Sep. 2006. [Online]. Available: http://dx.doi.org/10.1016/j.cviu.2006.06.005

[14] A. van den Hengel, R. Hill, B. Ward, A. Cichowski, H. Detmold, C. Madden, A. Dick, and J. Bastian, "Automatic camera placement for large scale surveillance networks," in *2009 Workshop on Applications of Computer Vision (WACV)*, Dec 2009, pp. 1–6.

[15] B. Dieber, C. Micheloni, and B. Rinner, "Resource-aware coverage and task assignment in visual sensor networks," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 21, no. 10, pp. 1424–1437, Oct 2011.

[16] Y.-G. Fu, J. Zhou, and L. Deng, "Surveillance of a 2d plane area with 3d deployed cameras," *Sensors*, vol. 14, no. 2, pp. 1988–2011, 2014. [Online]. Available: http://www.mdpi.com/1424-8220/14/2/1988

[17] Y.-C. Xu, B. Lei, and E. A. Hendriks, "Camera network coverage improving by particle swarm optimization," *EURASIP Journal on Image and Video Processing*, vol. 2011, no. 1, p. 458283, 2010. [Online]. Available: http://dx.doi.org/10.1155/2011/458283

[18] S. Nutanong, E. Tanin, and R. Zhang, "Visible nearest neighbor queries," in *Advances in Databases: Concepts, Systems and Applications, 12th International Conference on Database Systems for Advanced Applications, DASFAA 2007, Bangkok, Thailand, April 9-12, 2007, Proceedings*, 2007, pp. 876–883. [Online]. Available: https://doi.org/10.1007/978-3-540-71703-4_73

[19] ——, "Incremental evaluation of visible nearest neighbor queries," *IEEE Trans. Knowl. Data Eng.*, vol. 22, no. 5, pp. 665–681, 2010. [Online]. Available: https://doi.org/10.1109/TKDE.2009.158

[20] Y. Wang, Y. Gao, L. Chen, G. Chen, and Q. Li, "All-visible-k-nearest-neighbor queries," in *Database and Expert Systems Applications - 23rd International Conference, DEXA 2012, Vienna, Austria, September 3-6, 2012. Proceedings, Part II*, 2012, pp. 392–407. [Online]. Available: https://doi.org/10.1007/978-3-642-32597-7_34

[21] Y. Gao and B. Zheng, "Continuous obstructed nearest neighbor queries in spatial databases," in *Proceedings of the ACM SIGMOD International Conference on Management*

of Data, SIGMOD 2009, Providence, Rhode Island, USA, June 29 - July 2, 2009, 2009, pp. 577–590. [Online]. Available: http://doi.acm.org/10.1145/1559845.1559906

[22] Y. Gao, B. Zheng, G. Chen, Q. Li, and X. Guo, "Continuous visible nearest neighbor query processing in spatial databases," *VLDB J.*, vol. 20, no. 3, pp. 371–396, 2011. [Online]. Available: https://doi.org/10.1007/s00778-010-0200-z

[23] Y. Gao, B. Zheng, G. Chen, W. Lee, K. C. K. Lee, and Q. Li, "Visible reverse k-nearest neighbor queries," in *Proceedings of the 25th International Conference on Data Engineering, ICDE 2009, March 29 2009 - April 2 2009, Shanghai, China*, 2009, pp. 1203–1206. [Online]. Available: https://doi.org/10.1109/ICDE.2009.201

[24] D. Cohen-Or, Y. L. Chrysanthou, C. T. Silva, and F. Durand, "A survey of visibility for walkthrough applications," *IEEE Transactions on Visualization and Computer Graphics*, vol. 9, no. 3, pp. 412–431, July 2003.

[25] J. Bittner and P. Wonka, "Visibility in computer graphics," *Environment and Planning B: Planning and Design*, vol. 30, no. 5, pp. 729–755, 2003. [Online]. Available: http://dx.doi.org/10.1068/b2957

[26] J. M. Airey, J. H. Rohlf, and F. P. Brooks, Jr., "Towards image realism with interactive update rates in complex virtual building environments," in *Proceedings of the 1990 Symposium on Interactive 3D Graphics*, ser. I3D '90. New York, NY, USA: ACM, 1990, pp. 41–50. [Online]. Available: http://doi.acm.org/10.1145/91385.91416

[27] G. Schaufler, J. Dorsey, X. Decoret, and F. X. Sillion, "Conservative volumetric visibility with occluder fusion," in *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*, ser. SIGGRAPH '00. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 2000, pp. 229–238. [Online]. Available: http://dx.doi.org/10.1145/344779.344886

[28] P. Wonka, M. Wimmer, and D. Schmalstieg, "Visibility preprocessing with occluder fusion for urban walkthroughs," in *Proceedings of the Eurographics Workshop on Rendering*

Techniques 2000. London, UK, UK: Springer-Verlag, 2000, pp. 71–82. [Online]. Available: http://dl.acm.org/citation.cfm?id=647652.760610

[29] V. Koltun, Y. Chrysanthou, and D. Cohen-Or, "Virtual occluders: An efficient intermediate pvs representation," in *Proceedings of the Eurographics Workshop on Rendering Techniques 2000*. London, UK, UK: Springer-Verlag, 2000, pp. 59–70. [Online]. Available: http://dl.acm.org/citation.cfm?id=647652.732124

[30] ——, "Hardware-accelerated from-region visibility using a dual ray space," in *Proceedings of the 12th Eurographics Workshop on Rendering Techniques*. London, UK, UK: Springer-Verlag, 2001, pp. 205–216. [Online]. Available: http://dl.acm.org/citation.cfm?id=647653.760611

[31] S. Suri and J. O'Rourke, "Worst-case optimal algorithms for constructing visibility polygons with holes," in *Proceedings of the Second Annual Symposium on Computational Geometry*, ser. SCG '86. New York, NY, USA: ACM, 1986, pp. 14–23. [Online]. Available: http://doi.acm.org/10.1145/10515.10517

[32] A. Zarei and M. Ghodsi, "Query point visibility computation in polygons with holes," *Computational Geometry*, vol. 39, no. 2, pp. 78 – 90, 2008. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S092577210700034X

[33] ——, "Efficient computation of query point visibility in polygons with holes," in *Proceedings of the Twenty-first Annual Symposium on Computational Geometry*, ser. SCG '05. New York, NY, USA: ACM, 2005, pp. 314–320. [Online]. Available: http://doi.acm.org/10.1145/1064092.1064140

[34] B. Ben-Moshe, O. Hall-Holt, M. J. Katz, and J. S. B. Mitchell, "Computing the visibility graph of points within a polygon," in *Proceedings of the Twentieth Annual Symposium on Computational Geometry*, ser. SCG '04. New York, NY, USA: ACM, 2004, pp. 27–35. [Online]. Available: http://doi.acm.org/10.1145/997817.997825

[35] S. K. Ghosh, "On recognizing and characterizing visibility graphs of simple polygons," *Discrete & Computational Geometry*, vol. 17, no. 2, pp. 143–162, 1997. [Online]. Available: http://dx.doi.org/10.1007/BF02770871

[36] V. Chvtal, "A combinatorial theorem in plane geometry," *Journal of Combinatorial Theory, Series B*, vol. 18, no. 1, pp. 39 – 41, 1975. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0095895675900611

[37] S. Fisk, "A short proof of chvtal's watchman theorem," *Journal of Combinatorial Theory, Series B*, vol. 24, no. 3, p. 374, 1978. [Online]. Available: http://www.sciencedirect.com/science/article/pii/009589567890059X

[38] A. Guttman, "R-trees: A dynamic index structure for spatial searching," in *SIGMOD*. New York, NY, USA: ACM, 1984, pp. 47–57.

[39] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger, "The r*-tree: An efficient and robust access method for points and rectangles," in *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '90. New York, NY, USA: ACM, 1990, pp. 322–331. [Online]. Available: http://doi.acm.org/10.1145/93597.98741

[40] T. K. Sellis, N. Roussopoulos, and C. Faloutsos, "The r+-tree: A dynamic index for multi-dimensional objects," in *Proceedings of the 13th International Conference on Very Large Data Bases*, ser. VLDB '87. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1987, pp. 507–518. [Online]. Available: http://dl.acm.org/citation.cfm?id=645914.671636

[41] S. Berchtold, D. A. Keim, and H.-P. Kriegel, "The x-tree: An index structure for high-dimensional data," in *Proceedings of the 22th International Conference on Very Large Data Bases*, ser. VLDB '96. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1996, pp. 28–39. [Online]. Available: http://dl.acm.org/citation.cfm?id=645922.673502

[42] O. Maimon and L. Rokach, *Data Mining and Knowledge Discovery Handbook*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2005.

[43] D. Defays, "An efficient algorithm for a complete link method," *The Computer Journal*, vol. 20, no. 4, p. 364, 1977. [Online]. Available: +http://dx.doi.org/10.1093/comjnl/20.4.364

[44] R. Sibson, "Slink: An optimally efficient algorithm for the single-link cluster method," *The Computer Journal*, vol. 16, no. 1, p. 30, 1973. [Online]. Available: +http://dx.doi.org/10.1093/comjnl/16.1.30

[45] I. E. Sutherland and G. W. Hodgman, "Reentrant polygon clipping," *Commun. ACM*, vol. 17, no. 1, pp. 32–42, Jan. 1974. [Online]. Available: http://doi.acm.org/10.1145/360767.360802

[46] D. S. Hochbaum and A. Pathria, "Analysis of the greedy approach in problems of maximum k-coverage," *Naval Research Logistics*, vol. 45, no. 6, pp. 615–627, 1998. [Online]. Available: http://dx.doi.org/10.1002/(sici)1520-6750(199809)45:6%3C615::aid-nav5%3E3.0.co;2-5

[47] G. L. Nemhauser, L. A. Wolsey, and M. L. Fisher, "An analysis of approximations for maximizing submodular set functions—i," *Mathematical Programming*, vol. 14, no. 1, pp. 265–294, 1978. [Online]. Available: http://dx.doi.org/10.1007/BF01588971