The Maximum Visibility Facility Selection Query in Spatial Databases

[1]Ishat E Rabban, [2]Mohammed E. Ali, [3]Muhammad A. Cheema, [4]Tanzima Hashem
[1,2,4]BUET, Dhaka-1000, Bangladesh,[3]Monash University, Australia
[1,2,4]ieranik,eunus, tanzimahashem@cse.buet.ac.bd,[3] aamir.cheema@monash.edu

ABSTRACT

Given a set of obstacles in 2D or 3D space, a set of n candidate locations where facilities can be established, the Maximum Visibility Facility Selection (*MVFS*) query finds k out of the n locations, that yield the maximum visibility coverage of the data space. Though the *MVFS* problem has been extensively studied in visual sensor networks, computational geometry, and computer vision in the form of optimal camera placement problem, existing solutions are designed for discretized space and only work for *MVFS* instances having a few hundred facilities. In this paper, we revisit the *MVFS* problem to support new spatial database applications like *"where to place security cameras to ensure better surveillance of a building complex?"* or *"where to place billboards in the city to maximize visibility from the surrounding space?"*. We introduce the concept of *equivisibility triangulation* to devise the first approach to accurately determine the visibility coverage of continuous data space from a subset of the facility locations, which avoids the limitations of discretizing the data space. Then, we propose an efficient graph-theoretic approach that exploits the idea of vertex separators for efficient exact in-memory solution of the *MVFS* problem. Finally, we propose the first external-memory based approximation algorithm (with a guaranteed approximation ratio of $1 - \frac{1}{e}$) that is scalable for a large number of obstacles and facility locations. We conduct extensive experimental study to show the effectiveness and efficiency of our proposed algorithms.

CCS CONCEPTS

• **Information systems → Spatial-temporal systems**;

KEYWORDS

Visibility, Maximum Coverage Problem, Triangulation, Graph Partitioning, Greedy Approximation

ACM Reference format:
[1]Ishat E Rabban, [2]Mohammed E. Ali, [3]Muhammad A. Cheema, [4]Tanzima Hashem. 2019. The Maximum Visibility Facility Selection Query in Spatial Databases. In *Proceedings of 27th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems, Chicago, IL, USA, November 5–8, 2019 (SIGSPATIAL '19),* 10 pages.
https://doi.org/10.1145/3347146.3359091

SIGSPATIAL '19, November 5–8, 2019, Chicago, IL, USA

ACM ISBN 978-1-4503-6909-1/19/11…$15.00
https://doi.org/10.1145/3347146.3359091

1 INTRODUCTION

The 3D models of real-life urban structures such as buildings and infrastructures are becoming increasingly available through popular mapping services such as Google Maps and OpenStreetMap. These map based services also allow users to upload 3D models. Modern mobile devices enable users to build the 3D models of indoor spaces by a simple scan. The availability of 3D models of both outdoor and indoor spaces enables us to address many practical applications that require visibility computation in the presence of 3D obstacles. For example, an advertisement company may need to find a set of locations to build new billboards in a city to attract maximum number of people. A security company may need to find a set of locations to place cameras to ensure maximum surveillance of a building complex. A fire security expert may want to determine the locations of water sprinklers to be installed in a building to ensure maximum coverage of the sprinkler system in case of fire.

To answer the above queries efficiently, in this paper we investigate the Maximum Visibility Facility Selection (*MVFS*) query in spatial databases for optimal facility placement in the presence of obstacles. Given a set of obstacles, a set D of n candidate locations where new facilities can be placed, and the visibility range of the facilities, the *MVFS* query finds k locations from D for placing new facilities such that the combined visibility coverage is maximized.

The *MVFS* problem (also known as the Optimum Camera Placement (*OCP*) problem [9] [10]) is an important visibility based optimization problem, and has been studied extensively in visual sensor networks, computational geometry, robotics, and vision. This problem is NP-hard and the existing exact solutions of this problem rely on Binary Integer Programming (BIP) techniques. Commercial MILP (Mixed Integer Linear Programming) solvers take thousands of seconds to solve this problem with a maximum of 50 facility locations [10]. Since an exact solution of the *MVFS* problem is infeasible for a practical large scale scenario, a number of works focus on finding approximate solution using greedy algorithms and local search techniques [19] [2] [13] [10].

Though many existing works address the *MVFS* problem, they suffer from the following limitations: (i) Most of the existing scalable solutions assume a discretized data space, i.e., the data-space is sampled to form a set of control points, and the visibility coverage is measured in terms of the number of visible control points instead of the actual area/volume of the visible region. (ii) Since the existing exact solutions involve solving BIP formulations, and the number of binary variables in this formulation is high, the performance of these exact algorithms is not satisfactory even for small instances. (iii) No existing approaches offer disk-based solution and thus cannot handle the scenarios where the obstacle set is stored in secondary memory (as is typically the case for many real world

applications). Thus existing solution are not applicable in the context of big data. (iv) The existing approximation algorithms do not provide any theoretical bound on the approximation ratio.

In this paper, we revisit the *MVFS* problem in the context of big spatial data and provide both in-memory and disk-based efficient query solutions. In particular, we propose two exact in-memory algorithms (*BasicExact* and *EfficientExact*), that do not require the data-space to be discretized into control points. The key idea of our approaches is to construct a novel visibility based triangulation of the data-space, namely *equivisibility triangulation*, using which we can accurately determine the area of the region visible from one or more facility locations. *Thus we achieve a continuous notion of visibility, which is critical in applications where no spot in the data-space can be left unattended, i.e., in a highly secured environment.*

In our algorithms, we model the *MVFS* problem as a graph problem, and employ a divide-and-conquer strategy to obtain the optimum solution. We determine a vertex separator (a subset of the vertices, removal of which along with the associated edges renders the graph disconnected) of the graph, recursively solve the smaller subgraphs split by the vertex separator, and finally merge the solutions obtained from the subgraphs to form the overall solution of the *MVFS* problem. Our proposed algorithm can solve larger instances of the *MVFS* problem in comparison with the existing exact solutions and thus overcomes limitations (i) and (ii) of the existing *MVFS* solutions. *To the best of our knowledge, our solution is the first approach to solve MVFS problem in continuous data-space.*

To handle a large number of obstacles, we propose the first external-memory algorithm (*ScalableGreedy*), that addresses the limitations (iii) and (iv) of the existing approaches. To attain scalability, we apply a greedy approximate technique to reduce computational cost, use a disk-resident spatial data structure to accelerate obstacle retrieval, and employ a heuristic driven best-first search technique to reduce the I/O overhead. We prove that our approximate greedy method has an approximation ratio of $1 - \frac{1}{e}$. We empirically show that the error induced by the greedy approximation is less than 0.1%, and hence can be safely ignored. In summary we have made the following major contributions:

- We develop the first solution to the *MVFS* problem that does not simplify the problem by discretizing the space into control points and instead works on the continuous data space. We introduce a novel concept, namely equivisibility triangulation, using which we accurately compute the area/volume of the region visible from one or more facility locations.
- We develop an efficient divide-and-conquer algorithm (*EfficientExact*) that outperforms the existing exact solutions.
- We propose the first disk-resident algorithm (a greedy approximation algorithm called *ScalableGreedy*) which can handle very large datasets. Our proposed algorithm has theoretically proven approximation ratio and generates near optimal results in practice.
- We conduct extensive set of experiments to show the effectiveness and efficiency of our proposed algorithms.

2 LITERATURE REVIEW

The *MVFS* problem is a highly studied visibility based optimization problem. This problem is a variant of the well-known Optimum Camera Placement (*OCP*) problem. There are two major versions of the *OCP* problem, namely, the *set cover* and the *maximum k coverage* formulation. The *MVFS* problem is similar to the maximum k coverage formulation. In both problems, a region of interest (ROI) is provided that is to be observed by the visual sensors, i.e., cameras. The ROI is represented discretely by a set of control points. We are given a set D of n facility locations where a camera can be placed, the viewing range, the field of view of the camera etc. In the set cover version, we determine the minimum number of facility locations from D to place cameras such that a complete visibility coverage of the ROI is achieved. In the maximum k coverage version, given an integer k, we select k locations from D to place cameras such that the visual coverage of the ROI is maximized.

Both versions of the *OCP* problem discussed above are proved to be NP-hard. 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 [9] [10] [4] [3] [6] [12]. In the set cover version, the number of binary variables in the BIP formulation equals the number of facility locations. In the maximum k coverage version, the number of binary variables equals the sum of the number of facility locations and the number of control points. Thus finding the optimum solution of the maximum k coverage version of the *OCP* problem is computationally more expensive than solving the set cover version. In earlier works, commercial MILP (Mixed Integer Linear Programming) solvers were used to generate the optimum solution of the BIP formulations [10] [12]. The largest instance of the set cover version (maximum k coverage version, respectively) that has been optimally solved consists of 164 facility locations [4] (50 facility locations [10], respectively) and it takes thousands of seconds to solve these instances. Thus, 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 a number of works focus on finding approximate solution of the *OCP* problem. The approximate techniques include greedy algorithms and local search techniques [19] [2] [13] [10]. In the greedy algorithms, choice of facility locations is made at each iteration according to different greedy heuristics. In the local search technique, an initial solution is constructed and it is incrementally improved by jumping to a neighboring solution until it reaches a local optima. Approximate solutions based on AI techniques are also studied, which include evolutionary algorithm [18] [17], particle swarm optimization [5], simulated annealing [7] etc.

3 PROBLEM FORMULATION

In this section, first we define some terminologies and then formally state our *MVFS* query problem. For simplicity, we assume that each facility has equal viewing range. The candidate locations where facilities can be established are termed as *data points*. Two points in the data-space are *visible* to each other if the line segment joining the two points does not intersect any obstacles and the distance between the two points is less than or equal to the visibility range of a facility. Otherwise the two points are *non-visible*. The set of all points visible from a data point defines the *visible region* of the data point. We define the *kMVFS* and *MVFS* problems below.

Problem Definition: Let R^m be an m-dimensional data-space (m=2 or 3), O be the set of obstacles in the data-space, D be the set of n data points where facilities can be established, r be the viewing range of a facility, and k be an integer where $0 < k < n$. Then, the *kMVFS* query finds S_k, the subset of D with k data points ($S_k \subset D$ and $|S_k| = k$) that yields the visible region of maximum area (or volume, in 3D). Here the notation $|.|$ stands for the cardinality of a set. The *MVFS* query finds S_l, for each $0 < l < n$. Thus, the solution of the *MVFS* problem is the ordered list $[S_1, S_2, \ldots, S_{n-1}]$.

Note that, for simplicity while computing the area of visible regions, we consider the *supremum distance* metric. Under this metric, a facility with visibility range r covers an axis aligned square (in 2D) or cube (in 3D) of the data space with side length $2r$ centered at the facility location. However, we remark that our techniques can be applied on other distance metrics such as Euclidean distance.

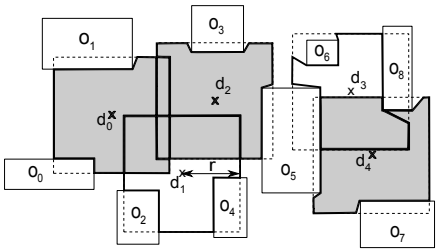

Figure 1: An instance of *kMVFS* problem for k=3.

Figure 1 shows a 2D instance of the *kMVFS* query for $k = 3$. In this example, there are 5 data points (d_0 to d_4, cross marked), and 9 obstacles (o_0 to o_8). Under the supremum distance metric, the viewing range of a facility is a square of side length $2r$ (shown as dotted squares). The visible regions of each data point is shown separately using 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. Thus, {d_0,d_2,d_4} is the solution of the *kMVFS* query where $k = 3$.

4 BASIC EXACT ALGORITHM

We first discuss some basic constructs for developing our in-memory exact algorithm for continuous space (Section 4.1). Then we present one of our key ideas of constructing visibility based triangulation, which facilitates the computation of visible area from multiple data points (Section 4.2). Based on our developed constructs, we present the basic exact algorithm in details (Section 4.3).

4.1 Preliminaries

4.1.1 Visible Region of a Data Point. The problem of constructing the visible region of a data point is a well studied problem in computational geometry. We adopt the work of Asano [1] to construct the visible region of a data point, which involves performing rotational plane sweep around the query point.

The visible region of a data point is a simple polygon. For example, in Figure 1, the shaded polygon on the right is the visible region of d_4. In this paper, we represent the visible region of a data point by its triangulation, i.e., a set of triangles. Connecting the data point with each pair of adjacent vertices of the visibility polygon creates a triangulation of the visible region.

If we use Euclidean distance metric instead of supremum distance, the visible region of a data point may contain circular patches. In that case, we can construct arbitrarily close approximation of such regions by increasing the number of triangles, as proposed in existing literature [11].

4.1.2 Intersection Graph. To solve the *MVFS* problem, we use a concept from graph theory, namely, the *Intersection Graph*, which is denoted by G^I. Given the set of data points, D, the set of obstacles, O, and the viewing range of a facility, r, the intersection graph contains nodes corresponding to the data points in D, and an undirected edge between two nodes if the visible regions of the corresponding data points overlap each other. In Figure 2, the visible regions of the data points are shown in grey and the intersection graph of the scenario is shown on the right. Here, the visible regions between the data points d_0, d_1, and d_2 overlap each other, hence there are edges between each pair of the corresponding nodes in the intersection graph. Similar is the case between d_3 and d_4.

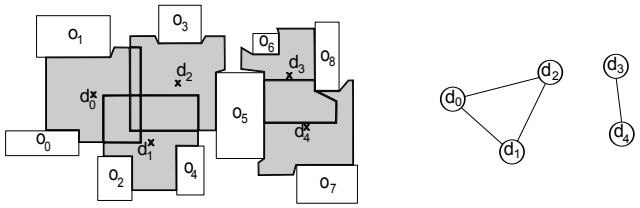

Figure 2: Intersection graph of an *MVFS* instance.

We construct the intersection graph as follows. We construct the visible region for all the data points in D. Then, for each pair of data points $(d_i, d_j) \in D$, we place an edge between the corresponding nodes of d_i and d_j in the intersection graph if and only if there exists a triangle $t_i \in V_{di}$, and a triangle $t_j \in V_{dj}$, and the intersection of t_i and t_j is not null. The process of determining the intersection between two triangles are described later in this section.

Note that, if two data points are located in separate connected components in the intersection graph, their visible regions do not overlap. This characteristics allows us to process the data points pertaining to each connected component independently.

4.2 Constructing Equivisibility Triangulation

The equivisibility triangulation is a partitioning of the data-space into disjoint triangles such that all points within a triangle are visible from the same subset of data points in D. For each triangle t in the partitioning, we store an additional piece of information (a bitmap value), which indicates the subset of D from which all the points inside t are visible. We can calculate the exact area of the region visible from any subset of D using the bitmap values. Thus the equivisibility triangulation enables us to achieve a continuous notion of the data-space unlike the previous approaches, which involved discretizing the dataspace into control points.

Definition 4.1. **Visibility Status**: 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 a bitmap of length n, where, for $0 \leq i < n$, the i^{th} bit is 1 if p is visible from the i^{th} data point in D, and 0 otherwise. If all points inside a triangle t have the same visibility status, the *visibility status of* t equals the visibility status of a point inside t.

Definition 4.2. **Equivisibility Triangulation**: Given a set of data points D, and a set of obstacles O, the *equivisibility triangulation*, T, is a partitioning of the visible region of D into disjoint triangles, such that for each triangle $t \in T$, all points inside t have the same visibility status.

To construct the equivisibility triangulation, we use two boolean operation on triangles, namely, *boolean intersection* and *boolean subtraction*. 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, which we denote by $t_a \cap t_b$. The boolean subtraction operation on t_a and t_b returns the region of t_a not inside t_b, which we denote by t_a/t_b. In Figure 3, 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. We represent the resultant regions by their triangulation.

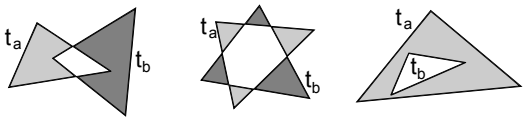

Figure 3: Boolean intersection and subtraction.

We present the algorithm *visTriangulation* that constructs the equivisibility triangulation of the data points pertaining to one connected component. In this algorithm, the routine *rangeQuery* is used to determine the subset of obstacles that intersect or fall within the visibility range of a the data point. The *visRegion* routine is used to determine the visible region of a data point in the form of a set of triangles as discussed in Section 4.1.1.

The **intersectRegion** and **subtractRegion** routines take as input two regions (A and B) and return their boolean intersection and subtraction respectively. The input and output regions are represented by their triangulation. To implement the *intersectRegion* routine, we iterate over all pair of triangles (t_A,t_B), where $t_A \in A$ and $t_B \in B$, and we accumulate the triangulation of the boolean intersection of all such (t_A,t_B) pairs in a list, which is returned at termination. To implement the *subtractRegion* routine, we consider each triangle of A, t_A, and perform boolean subtraction of all triangles of B from t_A. As we keep subtracting triangles of B from t_A, t_A may get split into multiple triangles. We keep track of the split triangles by maintaining a list, which is returned at termination.

In the algorithm *visTriangulation*, c_i denotes the i^{th} data point in C, $t.bitmap$ denotes the visibility status of triangle t, the $\ll$ operator denotes bitwise left shift, and subscripted L's denote lists of triangles. The data points in D are numbered from 0 to $n-1$. Now we describe the algorithm *visTriangulation* in details.

The algorithm *visTriangulation* constructs the equivisibility triangulation of a connected component C incrementally, including one data point at a time. Initially, in Lines 2-5, we construct the equivisibility triangulation for the first data point of C. Then, in the for loop spanning from Lines 6-18, we add one data point of C at each iteration and incrementally reconstruct the equivisibility triangulation. At the beginning of each iteration of the for loop, the list L_{old} holds the equivisibility triangulation of the first i data points of C. In Lines 7-8, we construct the equivisibility triangulation for the $(i+1)^{st}$ data point of C and put it in the list L_{new}. In Lines 9-11, we create three lists of triangles $L_{old'}$, $L_{new'}$, and L_{inter}. Here, $L_{old'}$ contains the triangulation of the region obtained from subtraction of region L_{new} from L_{old}, i.e., $L_{old'}$ is the region visible from one or more of the first i data points of C, but not from the $(i+1)^{st}$ data point of C. Similarly, $L_{new'}$ represents the region visible from the $(i+1)^{st}$ data point of C, but visible from no data point of the first i data points in C. L_{inter} represents the region visible from the $(i+1)^{st}$ data point of C and one or more data points from the first i data points of C. Note that, $L_{old'}$, $L_{new'}$, and L_{inter} collectively represent the region visible from one or more data points of the first $i+1$ data points of C. Finally we determine the bitmap values for the triangles in these three lists and combine the lists to construct the updated L_{old}, which represent the equivisibility triangulation of the first $i+1$ data points in C (Lines 12-18).

Algorithm 1: visTriangulation(O,C,r)

```
   input  : O,C,r
   output : Equivisibility triangulation of data points in C
1  begin
2      O_rq ⟵ rangeQuery(c_0, O, r)
3      L_old ⟵ visRegion(c_0, O_rq, r)
4      for each triangle t in L_old do
5          t.bitmap ⟵ (1 ≪ c_0)
6      for i ⟵ 1 to C.size() − 1 do
7          O_rq ⟵ rangeQuery(c_i, O, r)
8          L_new ⟵ visRegion(c_i, O_rq, r)
9          L_old' ⟵ subtractRegion(L_old, L_new)
10         L_new' ⟵ subtractRegion(L_new, L_old)
11         L_inter ⟵ intersectRegion(L_old, L_new)
12         for each triangle t in L_new' do
13             t.bitmap ⟵ (1 ≪ c_i)
14             L_old'.insert(t)
15         for each triangle t in L_inter do
16             t.bitmap ⟵ t.bitmap|(1 ≪ c_i)
17             L_old'.insert(t)
18         L_old ⟵ L_old'
19     return L_old
```

Figure 4 shows the construction of equivisibility triangulation for the first connected component of the scenario depicted in Figure 2. Initially the equivisibility triangulation holds the triangulation of the visible region of d_0 (Figure 4(a)). In the next two iteration, data points d_1 and d_2 are added and the equivisibility triangulation is constructed incrementally as shown in Figure 4(b) and Figure 4(c).

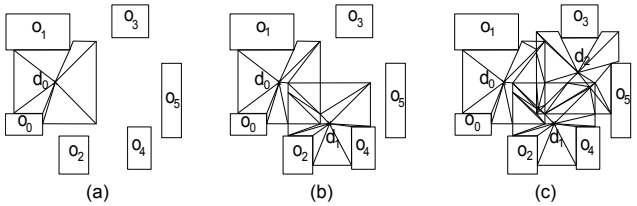

Figure 4: Constructing equivisibility triangulation.

To handle 3D *MVFS* instances, we modify our methodology as follows. To represent the visible region of a data point in 3D space, we use a set of tetrahedrons, instead of triangles. 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 is determined by performing plane sweep along principal axes. 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 *visTriangulation*.

Reduced List: After constructing the equivisibility triangulation, we employ an acceleration technique, in which, we *reduce* the equivisibility triangulation into a list of *element*s. Reducing the triangulation creates a compact representation of the triangulation without losing necessary information, which helps accelerate the subsequent algorithms. The *reduce* routine is described below.

In the equivisibility triangulation of a component, the number of unique bitmap values is significantly less than the number of triangles. For example, in Figure 4(c), there are 68 triangles but only 7 distinct bitmap values. Consequently, in the *reduce* routine, for each unique bitmap value, we create one *element*. Each element has a key (bitmap) and a value (area). The value of an element with key b equals the sum of areas of all triangles in the equivisibility triangulation having the bitmap value of b. By using *reduce*, we obtain a smaller list of elements, which we call the *reduced list* and denote by L^R. For the example depicted in Figure 4(c), there are 7 elements in the reduced list, because there are 7 distinct bitmap values (001 to 111). The reduced list is used to calculate the area visible from a subset of data points. Instead of searching all triangles in the triangulation, we traverse a smaller reduced list.

4.3 The Basic Exact Algorithm

In this section, first we describe how to solve the *MVFS* instance for each component independently (Section 4.3.1). Then, we discuss the process of merging the results obtained from different components (Section 4.3.2). Finally, we provide the pseudocode of the basic exact algorithm (Section 4.3.3).

4.3.1 Solving MVFS for a Component. We use the routine *basicSolveComponent* to solve the *MVFS* problem for a set of data points pertaining to one connected component. This routine takes as input a connected component, C, and the reduced list for the data points of C, L^R. The solution vector of the *MVFS* instance is an ordered list, which we denote by X. The process of generating X from C and L^R is called *solving the component* C.

The entries of X are (bitmap,area) pairs. Each bitmap has a length of n, which represents a subset of C (or, equivalently, D). The i^{th} entry of X, which we denote by $X[i]$, stores the (bitmap,area) pair corresponding to the solution of the *kMVFS* problem with $k = i$. To determine $X[i]$, we consider all possible subsets of C where i bits are set. For each such subset S, we determine the visible area of the data points in S by adding up the area of the appropriate elements from the reduced list, L^R, and select the subset with highest area. We repeat the procedure for each i, where $0 \leq i \leq n$, to populate the solution vector X.

4.3.2 Merging Results of Multiple Components. The routine *mergeComponents* employs an incremental approach to merge the solution of multiple components and form the overall solution. The overall solution vector is X^*, which initially contains the solution of the first component. We update X^* by incorporating the solutions of the rest of the components iteratively as follows.

Let, the set of all components be $\{C_0, C_1, C_2, \ldots, C_m\}$. Each component C_i, $0 \leq i \leq m$, contains the set of data points in the i^{th} component. The solution vector of component C_i is denoted by X_i, which can be computed using the *basicSolveComponent* routine. At the beginning of the *mergeComponents* routine, X^* is set to X_0. On the i^{th} iteration, the result of the $(i+1)^{st}$ component, X_i, is merged with the result of the first i components, which is available in X^*. The merging process at each iteration is as follows. We consider all possible index values of X^*, from 0 to $X^*.size() + X_i.size()$. For each index, k, we consider all possible choices (j data points selected from the first i components, whose result is stored at $X^*[j]$, and $k - j$ data points selected from the $(i+1)^{st}$ component, whose result is stored at $X_i[k-j]$; j ranges from 0 to $min(X^*.size(), k)$) and determine the optimum choice. After the m^{th} iteration, we return X^* as the optimum solution.

4.3.3 The Algorithm. In the basic exact algorithm, first we divide the data points of D into a set of connected components, C, using the routine *connectedComponents* (Line 2). This routine takes a set of data points, a set of obstacles, and the visibility range as input parameters and returns the set of connected components. The routine determines the connected components by first constructing the intersection graph G^I and then performing Depth First Search (DFS) on G^I. Next, for each component c, we construct the equivisibility triangulation, T (Line 4), reduce T to obtain the reduced list, L^R (Line 5), and use the *basicSolveComponent* routine to obtain the solution of the *MVFS* problem for c (Line 6). The solutions of all the components is accumulated in a 2D data structure, U (Line 1, 6). Finally we merge the results of all components using the algorithm *mergeComponents* to solve the *MVFS* problem (Line 7).

We solve the *kMVFS* problem by making the following modification to the above solution. In *basicSolveComponent*, instead of considering all possible subsets, we consider only the subsets of size less than or equal to k, and in the routine *mergeComponents*, we return $X[k]$ instead of X. In the basic exact algorithm, the time

Algorithm 2: BasicExact(O,D,r)

```
   input  : O,D,r
   output : Output of the MVFS Problem
1  begin
2      U ⟵ ∅, C ⟵ connectedComponents(O, D, r)
3      for each connected component c ∈ C do
4          T ⟵ visTriangulation(O, c, r)
5          L^R ⟵ reduce(T)
6          U.insert(basicSolveComponent(c, L^R))
7      return mergeComponents(U)
```

complexity of the routines *visTriangulation* and *reduce* is exponential on the size of the component, because for a component c, up to $2^{c.size()}$ distinct bitmap values may arise from equivisibility triangulation. The time complexity of the routine *basicSolveComponent* is also exponential on the component size, because for a component c, we consider all $2^{c.size()}$ subsets of c to find the optimum solution.

According to the above discussion, the performance of the basic exact algorithm is dependent on the size of the largest connected component, instead of the total number of data points n. Thus by treating the connected components independently, we achieve a significant performance speedup.

5 EFFICIENT EXACT ALGORITHM

The performance of the basic exact algorithm described in the previous section depends exponentially on the cardinality of the largest component, and consequently degrades rapidly when the cardinality of the largest component increases. In this section, we propose an improved methodology, the *efficient exact algorithm*, which solves a component faster by using *vertex separators*.

5.1 The Key Insight: Vertex Separator

The key insight of our approach to solve a component faster is based on the idea of vertex separators. A vertex separator of a connected graph G is a set of vertices S, $S \subset V(G)$, such that the deletion of S from G renders G disconnected. Here, $V(G)$ denotes the set of vertices of the graph G. To solve a component faster, first we determine a vertex separator of the component and branch on the vertices of the vertex separator. If the cardinality of the vertex separator is κ, we create 2^κ branches, one branch for each selection choice of the vertices in the vertex separator. A vertex in the vertex separator can either be selected or be rejected and thus 2^κ branches are created. The vertex separator splits the graph into two or more smaller connected components, which we call *inner components*. For each branch, we solve the inner components and merge the results of the inner components to form the overall solution.

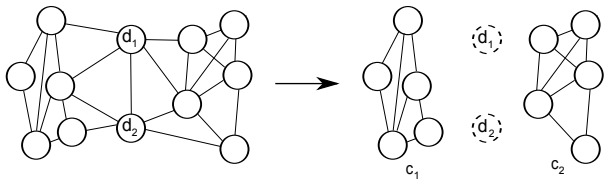

Figure 5: Splitting a component by a separator.

Figure 5 illustrates the above idea. Here the connected component shown in the left has 12 vertices. In the basic exact algorithm, solving this component requires calculating visible regions for 4096 (=2^{12}) subsets. But we can use the idea described above to avoid considering all 4096 subsets. First we determine a vertex separator, which consists of two vertices, d_1 and d_2, as shown in the right of Figure 5. Thus the value of κ is 2. The graph is split into two inner components, c_1 and c_2, both having 5 vertices. We create 4 (= 2^κ) branches, one for each selection choice of d_1 and d_2. Then, for each branch, we solve c_1 and c_2 separately by considering all possible subsets (2^5 subsets for each of c_1 and c_2) within each inner component. In total, we consider 256 (= $2^2 * (2^5 + 2^5)$) subsets. Thus we reduce the number of subsets to be considered and achieve performance speedup over the basic exact algorithm.

In the above method, the number of subsets we consider while solving a component depends exponentially on the sum of the cardinality of the vertex separator and the cardinality of the largest inner component. We name this sum as the *critical number* of a component. According to the idea developed above, the running time of solving a component depends exponentially on the critical number of the component. Thus a vertex separator leads to better performance if the separator has a small size and the size of the largest inner component split by the separator is as small as possible. The problem of finding a vertex separator of small size that splits a graph into inner components of roughly equal size has been extensively studied in literature [15] [16]. We adopt an idea based on articulation points [16] to find the vertex separator.

5.2 The Efficient Exact Algorithm

In the improved algorithm, we use the idea of vertex separators, outlined in Section 5.1, to provide faster methods for solving a component (Section 5.2.1) and constructing the equivisibility triangulation of a component (Section 5.2.2). We obtain the improved solution for the *MVFS* problem, the algorithm *EfficientExact*, by replacing the methods for constructing equivisibility triangulation and solving a component in the basic exact algorithm (Line 4 and 6 of algorithm *BasicExact*) with the procedures mentioned below.

5.2.1 Solving a Component. In the improved algorithm, to solve a component faster, we consider each subset, S', of the vertex separator, S, independently. For each such subset S', we remove the visible region of S' from the reduced list, we solve all the inner components, merge the solution of the inner components, and re-include the vertices of S' into the merged solution. Finally, for each cardinality, we return the choice having the maximum area over all subsets of S. Here the *setBits* routine returns a bitmap of length n representing the input subset S of data points. The *countSetBits* routine returns the number of set bits in the input bitmap b.

First, a list of (bitmap,area) pairs, X, which would hold the solution of the component, is initialized (Line 2). We determine the vertex separator, S, for the connected component C (Line 3) and determine the set of inner components, I, obtained by splitting C by the vertex separator S (Line 4). The routine *split* is implemented using the DFS algorithm. We consider each subset of the vertices in S (Line 5). Similar to the basic algorithm, a subset of the vertices in S is represented by the bitmap, *bitmapS* (Line 6). For each such subset, S', we construct a reduced list of elements, *tmpL*. To obtain *tmpL*, we remove from L^R the elements containing one or more vertices of S' (Lines 7-8) using *remove*. The *remove* method updates the list of elements provided in the first parameter (*tmpL*) by removing from the list the elements having set bits in the bitmap provided in the second parameter (*bitmapS*) and returns the sum of the area of the removed elements. Next, we solve all the inner components separately and merge the results of all the inner components to form W (Lines 9-12). For each subset S', W holds the solution that excludes the vertices of S'. Hence we include the vertices of S' by considering the number of vertices of S' (l) and the area covered by the vertices of S' (*initArea*) and update X accordingly (Lines 13-17).

5.2.2 Constructing the Triangulation. We outline below an improved procedure to construct the equivisibility triangulation of a component using the idea of vertex separator. We do not describe the procedure in details for brevity. Instead, we enumerate the key steps of the procedure below. We also thematically illustrate the process using Venn diagram as shown in Figure 6.

- First we find the vertex separator of the component C and the inner components split by the vertex separator. Let S be the set of data points in the vertex separator of C, S^* be the set of data points in C that are not in S, and I be the set of inner components. In Figure 6, $I = \{I_1, I_2\}$, and $S^* = I_1 \cup I_2$. The visible region of S is oval shaped, and the visible regions of I_1 and I_2 are circular.

Algorithm 3: improvedSolveComponent(C,L^R)

```
input  : C, L^R
output : Solution of MVFS instance for component C
1  begin
2      X.init(C.size() + 1, −1.0)
3      S ⟵ findSeparator(C)
4      I ⟵ split(C, S)
5      for each subset S' of S do
6          bitmapS ⟵ setBits(S')
7          tmpL ⟵ L^R
8          initArea ⟵ remove(tmpL, bitmapS)
9          U ⟵ ∅
10         for each component c ∈ I do
11             U.insert(basicSolveComponent(c, tmpL))
12         W ⟵ mergeComponents(U)
13         l ⟵ countSetBits(bitmapS)
14         for j ⟵ 0 to W.size() − 1 do
15             if X[l + j].area < W[j].area + initArea then
16                 X[l + j].area ⟵ W[j].area + initArea
17                 X[l + j].bitmap ⟵ W[j].bitmap|bitmapS
18     return X
```

- Next we construct triangulation for the region of the data space visible from one or more data points in S, but visible from no data point in S^*. Let this triangulation be denoted by T. We construct T by using the routine *subtractRegion*. In Figure 6, T corresponds to the dark grey region.
- Then we construct equivisibility triangulation of all inner components in I separately using the algorithm *visTriangulation*. We denote the triangulation by T^*. In Figure 6, T^* corresponds to the circles I_1 and I_2.
- Note that, the triangles in T^* that are visible from one or more data point in S (the white overlapping region in Figure 6) have incorrect bitmap value, because their visibility from S will not be reflected in their bitmaps. We correct such bitmap values by finding the triangles in T^* that fall within the visible region of one or more data points in S and updating their bitmaps accordingly. Thus, the equivisibility triangulation of C is $T \cup T^*$.

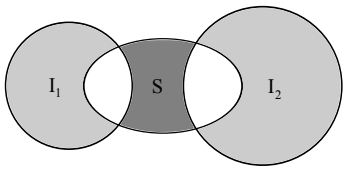

Figure 6: Constructing equivisibility triangulation using vertex separator.

The performance of the efficient exact algorithm depends on the maximum critical number among the components of the intersection graph. The efficient exact algorithm empirically performs far better than the basic version as shown in the experimental section. Because, for a randomly built graph, the largest component size is usually much bigger than the maximum critical number.

5.3 Baseline Method

The existing solutions of the *MVFS* problem use Binary Integer Programming (BIP) technique to find the exact solution. BIP technique is a special class of linear programming. Linear programming is a method to solve an optimization problem, where the problem is formulated as a mathematical model, the goal of the problem (which is to maximize/minimize something) is represented as a linear objective function, and the requirements of the problem are represented using linear equality of inequality constraints. BIP is a sub-class of linear programming, where the variables (to be determined) are restricted to assume only binary values (0 or 1). After constructing the equivisibility triangulation and finding the reduced list, we can solve the *kMVFS* problem using BIP techniques. The BIP formulation of the *kMVFS* can be constructed as follows.

In the reduced list, each element corresponds to the region visible from a unique subset of the data points. The bitmap of an element represents the subset of data points from which the region is visible. The area of each element represents the area of the corresponding region. Let, for $0 \leq i < n$, S_i denote the set of elements of the reduced list, whose i^{th} bit is set. In other words, S_i contains the elements/regions visible from the i^{th} data point, d_i. In the BIP formulation, for each data point, we create a binary variable x_i to indicate whether or not the i^{th} data point is in the optimum selection of k data points. Also for each element e in the reduced list, we declare a binary variable y_e to indicate whether or not the element e is covered by any of the k selected data points.

To enforce that no more than k data points are selected, we add the following constraint: $\sum_{0 \leq i < n} x_i \leq k$. To ensure that an element e is covered if and only if at least one of the data points from which e is visible is selected, we include the following constraint for each element e in the reduced list: $\sum_{e \in S_i} x_i \geq y_e$. Note that if $y_e > 0$, at least one data point from which e is visible must be selected. Our goal is to maximize the sum of the areas of the covered elements, hence the objective function is: $\sum_{e \in L^R} y_e * e.area$.

In the baseline method, we first construct the equivisibility triangulation, then find the reduced list, and finally use a commercial MILP solver, Gurobi, to solve the above mentioned BIP formulation.

6 SCALABLE ALGORITHM

The exact algorithms proposed in Section 4 and 5 are not scalable because they have exponential time complexity and they can not handle large disk-resident datasets. In this section, we present an algorithm that achieves scalability by employing a greedy approximation technique (Section 6.1), and using a heuristic-driven algorithm to guide the greedy search (Section 6.2).

6.1 The Greedy Approximation Technique

In this section, we outline a polynomial-time greedy approximation algorithm for the *MVFS* problem and prove that it's approximation ratio is $1 - \frac{1}{e}$. We obtain such results by reducing the *kMVFS* problem to the well known Weighted Maximum k Coverage (*WMkC*) problem. The formulation of the *WMkC* problem is given below.

The Weighted Maximum k Coverage Problem: An integer k and a collection of n sets S=S_0, S_1, ...S_{n-1} are given. Each ground element of the sets 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.

The *WMkC* problem is known to be NP-hard. The best known approximation algorithm for the *WMkC* 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 as the best-possible polynomial time approximation algorithm for the *WMkC* problem and has an approximation ratio of $1 - \frac{1}{e}$ [8] [14].

The process of reducing the *kMVFS* problem to the *WMkC* problem is as follows. Consider an instance of the *kMVFS* problem. All the triangles/elements in the equivisibility triangulation/reduced list is the set of ground elements. The weight of each triangle/element is its area. The sets of triangles/elements visible from the n data points are denoted by $S_0, S_1, ...S_{n-1}$. The objective is to maximize the weighted coverage of k sets selected from the n sets. Thus the *kMVFS* problem can be reduced to the *WMkC* problem.

A greedy approach for the *kMVFS* problem, similar to the one for the *WMkC* problem, is to choose, at each step, the data point that covers the highest non-visible area. According to the above reduction, this greedy approach for the *kMVFS* problem has an approximation ratio of $1 - \frac{1}{e}$.

6.2 The Scalable Algorithm

In the scalable algorithm, we partition the data space homogeneously into equal sized *cell*s as in a grid. We use a matrix data structure to store the visibility status of the cells. A cell c is considered as *covered* if c is situated completely inside any obstacle or c is completely visible from any greedily chosen data point; otherwise, c is considered as *non-covered*. In the greedy algorithm, we iteratively make k greedy choices. At each iteration, we select the data point that maximizes the number of cells that turns from non-covered to covered. We index the obstacles in an R-tree to quickly retrieve the obstacles located within the viewing range of a data point.

To make a greedy choice, instead of expanding all the data points at random, we use a heuristic to establish an order, according to which the data points are expanded. The heuristic value of a data point is the number of non-covered cells within its viewing range.

In the scalable algorithm, we maintain a max priority queue for the data points where the key of a data point is its heuristic value. Initially all the data points are marked as *OPEN*. We retrieve data points from the queue according to the heuristic value. If the retrieved data point is marked *OPEN*, we calculate the number of cells that turns covered from non-covered if a camera is placed at the data point, mark the data point *CLOSED*, and insert the data point back into the queue. Otherwise, if the retrieved data point is marked *CLOSED*, we select the data point as the k^{th} greedy choice and update the heuristic values of nearby data points.

Note that, the heuristic value of a data point provides an upper bound on the visibility of the data points, and consequently allows us to make k greedy choices without necessarily expanding all the data points. We do not present the algorithm in details for brevity.

7 EXPERIMENTAL EVALUATION

Our experiments are based on real (Boston Dataset[1]) and synthetic 2D obstacle datasets. To construct synthetic obstacle dataset, we generated obstacles of varying size uniformly all over the extent of the data-space. The obstacles are non-overlapping axis aligned 2D rectangles. The set of data points is synthetic and generated uniformly such that the data points do not lie inside any obstacle. The algorithms are implemented in C++ and the experiments are conducted on a core i7 3.40 GHz PC with 8GB RAM, running Microsoft Windows 10.

We evaluate the efficiency and effectiveness of our algorithms by varying the following parameters: (i) the number of obstacles, (ii) the number of data points, (iii) visibility range, and (iv) type of dataset. To evaluate the performance of the exact algorithms, we use total execution time as the metric. In case of the scalable greedy algorithm, the evaluation metric are total execution time, IO time, approximation error, and the number of range queries issued. For each experiment, we generate 20 random input instances with the same parameter setting and report the average performance.

7.1 Evaluation of Exact Algorithms

Table 1: Parameters for Exact Algorithms

Parameter	Range	Default
Number of Obstacles (K)	25, 50, 75, 100	50K
Number of Data Points	8, 16, 32, 64	32
Visibility Range	10, 15, 20, 25	20
Dataset	Synthetic, Real	Synthetic
Data-Space Size		1000*1000

The ranges and default values of the parameters for the exact algorithms are listed in Table 1. The experimental results, presented in Figure 7, show that the execution time of the exact algorithms do not vary much with the size of the obstacle set (Figure 7(a)). The execution time increases when the visibility range of a facility increases. Because increase in visibility range causes more overlap between the visible regions of the data points, and consequently increases the largest component size and maximum critical number (Figure 7(b)). The execution time increases when n increases. Because increase in the number of data points increases the largest component size and maximum critical number (Figure 7(c)). To evaluate the performance of the exact algorithms with real dataset and compare the results with synthetic datasets, we conducted experiments using the Boston dataset, which contains 11,437 obstacles. We compared the results with a synthetic dataset containing the same number of obstacle, as shown in Figure 7(d). The results shows no significant difference between the running time of the exact algorithms for real and synthetic datasets. Consequently, the rest of the experimental results shown in this section are based on synthetic datasets.

All the experimental results show that the efficient exact algorithm runs orders of magnitude faster than the basic exact algorithm. Because, in the intersection graph of an *MVFS* instance, the maximum critical number is usually significantly lower than the largest component size, as described in the next section.

7.2 Comparison with Existing Solutions

In this section we introduce a new parameter, the edge density of the intersection graph and use it as a parameter to compare the performance of the exact algorithms.

[1]http://www.bostonplans.org/3d-data-maps

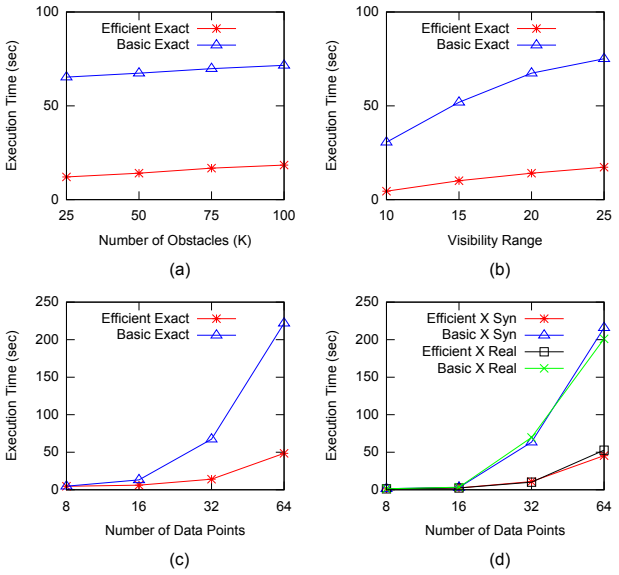

Figure 7: Comparison of the exact algorithms.

7.2.1 Edge Density as a Parameter. The performance of the exact algorithms depends on the structure of the intersection graph, i.e., largest component size in case of the basic exact algorithm (Section 4.3), and maximum critical number in case of the efficient exact algorithm (Section 5.2.2). To compare the performance of the exact algorithms, we use a parameter that is positively correlated with the largest component size and maximum critical number, namely, the *edge density* of the intersection graph. In a graph with v nodes and e edges, the edge density is the average degree of a node, i.e., $\frac{2e}{v}$. The positive correlation of largest component size and maximum critical number with edge density is demonstrated in Figure 8(a), which also shows that the first quantity is significantly larger than the later. In this experiment, we generate *MVFS* instances with 64 data points with edge density values between 2.0 to 5.0. For each of the following ranges of edge density values, [2.0-2.5], [2.5-3.0], ..., and [4.5-5.0], we plot the average of the largest component size and the maximum critical number.

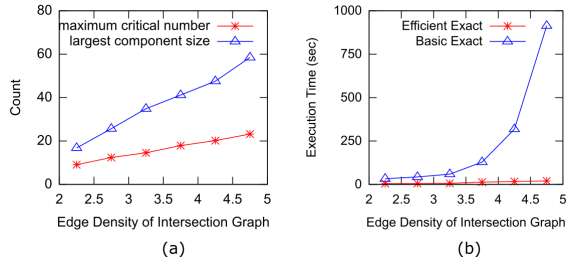

Figure 8: Edge density as a parameter.

The edge density of the intersection graph incorporates the level of overlap between the visible regions of the data points. An *MVFS* instance with edge density x corresponds to a scenario where the visible region of one data point overlaps with the visible regions of x other data points on average. Figure 8(b) shows that the efficient exact algorithm significantly outperforms the basic exact version.

7.2.2 Comparison with Existing Exact Solutions. We present the effect of the number of data points on the total processing time of the efficient exact algorithm in Figure 9(a). In this experiment, for each n, we generate random instances of the *MVFS* problem, and plot the average total processing time of the efficient exact algorithm against the edge density ranges. Note that, the total processing time increases with increasing n. Because as the number of nodes in a graph increases, the size of the largest component, and consequently the maximum critical number increases. The exact algorithms proposed in literature [10] could solve *MVFS* instances with at most 50 data points. The efficient exact algorithm proposed in this paper can solve *MVFS* instances with 256 data points and edge density value of 4.75 within 500 seconds on average.

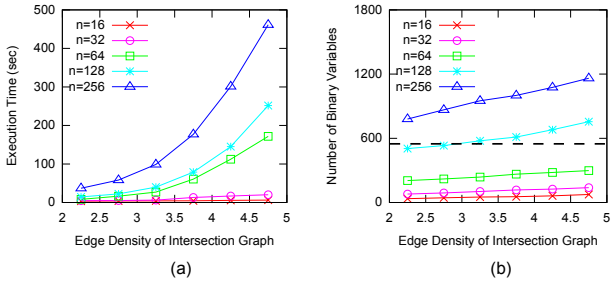

Figure 9: (a) Performance of efficient exact algorithm. (b) Comparison with baseline method.

The earlier exact solutions formulated the *MVFS* problem as a binary integer program (BIP) and used commercial linear program solvers to find the optimum solution of the (BIP). The number of binary variables in the BIP form of the *MVFS* problem equals the sum of the number of control points (i.e., the size of the reduced list in our formulation) and the number of data points. We denote the number of binary variables in the BIP formulation by b. We conduct an experiment, where we use a state-of-the-art MILP solver, Gurobi, to solve the BIP formulation of the *MVFS* problem. The solver was able to generate the optimum solution when $b < 530$, but failed to generate any solution for $b \geq 530$.

Figure 9(b) shows the results of an experiment, where for each n, we generate input instances, and plot the number of binary variables in the BIP formulation against the edge density ranges. The graph shows that *MVFS* instances having 128 data points and edge density over 3.0, or having 256 data points have more than 530 binary variables, and hence can not be solved by the state-of-the-art LP solvers. But, using the efficient exact algorithm, we solved *MVFS* instances with $b > 1000$ within approximately 500 seconds. Thus our solution outperforms the earlier exact approaches.

7.3 Evaluation of Greedy Algorithm

To assess the performance of the scalable greedy algorithm, we index the obstacles in an R-tree, with disk page size fixed at 1KB. We report the total time and I/O time to compare the computational and the I/O cost. The number of obstacles, number of data points, and the data-space size are set to 50K, 256, and 10000*10000 respectively by default. The default value of k is set to n.

7.3.1 Approximation Error. In the *ScalableGreedy* algorithm, we achieve scalability by employing a greedy approximation technique. The greedy approximation algorithm has a theoretically proven

approximation ratio of $1 - \frac{1}{e}$. But in our experiments, the average approximation error was found to be less than 0.1%. Thus we conclude that the greedy approximation algorithm generates near optimal solution of the *MVFS* problem.

7.3.2 Comparison of Exact and Greedy Algorithms. In this section, we compare the performance of the *EfficientExact* and *ScalableGreedy* algorithms by varying the number of data points (n) between 16 to 256. For each value of n, we generate 20 random *MVFS* instances having edge density value between 3 to 4, and report the average total execution time. We found that the scalable algorithm runs almost 15 times faster than the efficient exact algorithm.

7.3.3 Effect of Number of Obstacles and Data Points. Figure 10 shows the effect of the number of obstacles (a) and number of data points (b) on total processing time and I/O time of the scalable greedy algorithm. In the first experiment, we vary the number of obstacles from 25K to 100K, with 25K increments. 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. Consequently the I/O time increases. The figure demonstrates that the total processing time is dominated by the I/O time. The computational cost does not vary much with increase in number of obstacles.

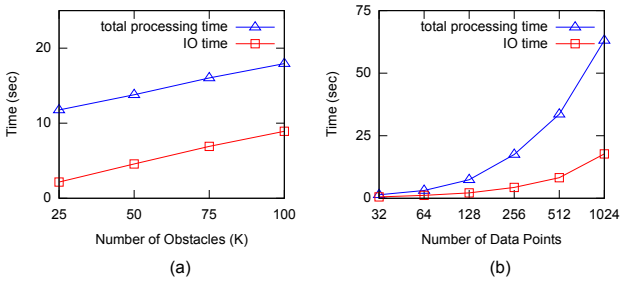

Figure 10: Performance of scalable algorithm.

In the second experiment, the number of data points are varied from 32 to 1024. The results show that the total processing time increases with increasing number of data points. As n increases, so does the number of greedy iterations. Consequently the computational cost increases. With increasing n, the number of range queries increases, which explains the increase in I/O time.

7.3.4 Effect of k. In this experiment, which is summarized in Table 2, we set n to 256, vary k exponentially from 1 to 256, and report the number of range queries issued by the scalable greedy algorithm, I/O time, and total processing time in seconds. The number of range queries increases with k and reaches a maximum of 256, because the greedy algorithm uses a heuristic to avoid issuing range query for all the data points for small values of k. Consequently, the I/O time and total processing time increases with increasing k.

Table 2: Effect of k

k	**1**	**2**	**4**	**8**	**16**	**32**	**64**	**128**	**256**
#RQ	17	31	65	154	243	255	256	256	256
IO time	0.25	0.44	1.12	2.65	4.32	4.56	4.57	4.57	4.59
Total time	2.91	3.51	6.08	11.8	16.9	18.0	18.3	18.4	18.4

8 CONCLUSION

We have developed a novel triangulation based technique to solve the *MVFS* problem in a continuous data-space, unlike all previous solutions which worked for discrete settings only. We have proposed an exact algorithm (*EfficientExact*) for the *MVFS* problem based on graph theory, which has outperformed the existing exact approaches. We have formulated the first scalable solution (*ScalableGreedy*) for the *MVFS* problem with guaranteed approximation ratio, which can handle large disk-resident obstacle sets.

Our proposed exact algorithm is able to solve larger instances of the *MVFS* problem in comparison with the existing exact methods. Previous exact solutions could solve *MVFS* instances having at most 530 binary variables, while our method can solve instances having more than 1000 binary variables. Our proposed scalable solution runs orders of magnitude faster than our exact algorithm. In future, we hope to develop novel algorithms to solve the set cover version of the *OCP* problem in a continuous data-space.

REFERENCES

[1] T. Asano. An efficient algorithm for finding the visibility polygon for a polygonal region with holes. *IEICE Transactions*, 68(9):557–559, 1985.
[2] W. Cheng, S. Li, X. Liao, S. Changxiang, and H. Chen. Maximal coverage scheduling in randomly deployed directional sensor networks. In *Parallel Processing Workshops, 2007. ICPPW 2007. International Conference on*, pages 68–68. IEEE, 2007.
[3] B. Debaque, R. Jedidi, and D. Prevost. Optimal video camera network deployment to support security monitoring. In *2009 12th International Conference on Information Fusion*, pages 1730–1736, July 2009.
[4] U. M. Erdem and S. Sclaroff. Automated camera layout to satisfy task-specific and floor plan-specific coverage requirements. *Comput. Vis. Image Underst.*, 103(3):156–169, Sept. 2006.
[5] Y.-G. Fu, J. Zhou, and L. Deng. Surveillance of a 2d plane area with 3d deployed cameras. *Sensors*, 14(2):1988–2011, 2014.
[6] J.-J. Gonzalez-Barbosa, T. García-Ramírez, J. Salas, J.-B. Hurtado-Ramos, et al. Optimal camera placement for total coverage. In *Robotics and Automation, 2009. ICRA'09. IEEE International Conference on*, pages 844–848. IEEE, 2009.
[7] S. Hanoun, A. Bhatti, D. Creighton, S. Nahavandi, P. Crothers, and C. G. Esparza. Target coverage in camera networks for manufacturing workplaces. *Journal of intelligent manufacturing*, 27(6):1221–1235, 2016.
[8] D. S. Hochbaum and A. Pathria. Analysis of the greedy approach in problems of maximum k-coverage. *Naval Research Logistics*, 45(6):615–627, 1998.
[9] E. Horster and R. Lienhart. Approximating optimal visual sensor placement. In *2006 IEEE International Conference on Multimedia and Expo*, pages 1257–1260, July 2006.
[10] 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*, VSSN '06, pages 111–120, New York, NY, USA, 2006. ACM.
[11] S. H. Lo. A new mesh generation scheme for arbitrary planar domains. *International Journal for Numerical Methods in Engineering*, 21:1403–1426, 1981.
[12] A. T. Murray, K. Kim, J. W. Davis, R. Machiraju, and R. Parent. Coverage optimization to support security monitoring. *Computers, Environment and Urban Systems*, 31(2):133–147, 2007.
[13] A. Neishaboori, A. Saeed, K. A. Harras, and A. Mohamed. On target coverage in mobile visual sensor networks. In *Proceedings of the 12th ACM international symposium on Mobility management and wireless access*, pages 39–46. ACM, 2014.
[14] G. L. Nemhauser, L. A. Wolsey, and M. L. Fisher. An analysis of approximations for maximizing submodular set functions—i. *Mathematical Programming*, 14(1):265–294, 1978.
[15] A. Pothen. Graph partitioning algorithms with applications to scientific computing. Technical report, Norfolk, VA, USA, 1997.
[16] H. L. S. Rosenberg, Arnold L. *Graph Separators, with Applications*. Kluwer Academic Publishers, Norwell, MA, USA, 2001.
[17] I. Sreedevi, N. R. Mittal, S. Chaudhury, and A. Bhattacharyya. Camera placement for surveillance applications. In *Video Surveillance*. InTech, 2011.
[18] C. Wang, F. Qi, and G.-M. Shi. Nodes placement for optimizing coverage of visual sensor networks. *Advances in Multimedia Information Processing-PCM 2009*, pages 1144–1149, 2009.
[19] J. Zhao, R. Yoshida, S.-c. S. Cheung, and D. Haws. Approximate techniques in solving optimal camera placement problems. *International Journal of Distributed Sensor Networks*, 9(11):241913, 2013.