Continuous Maximum Visibility Query for a Moving Target

Ch. Md. Rakin Haider[1], Arif Arman[1]
Mohammed Eunus Ali[1], and Farhana Murtaza Choudhury[2]

[1] Bangladesh University of Engineering and Technology, Bangladesh
{eunus,rakinhaider}@cse.buet.ac.bd
{arman}@cse.uiu.ac.bd
[2] RMIT University, Melbourne, Australia
{farhana.choudhury}@rmit.edu.au

Abstract. Opportunities to answer many real life queries such as *"which surveillance camera has the best view of a moving car in the presence of obstacles?"* have become a reality due to the development of location based services and recent advances in 3D modeling of urban environments. In this paper, we investigate the problem of continuously finding the best viewpoint from a set of candidate viewpoints that provides the best view of a moving target in presence of visual obstacles in 2D or 3D space. We propose a query type called *k Continuous Maximum Visibility (kCMV)* query that ranks *k* query viewpoints (or locations) from a set of candidate viewpoints in the increasing order of the visibility measure of the target from these viewpoints. We propose two approaches that reduce the set of query locations and obstacles to consider during visibility computation and efficiently update the results as target moves. We conduct extensive experiments to demonstrate the effectiveness and efficiency of our solutions for a moving target in presence of obstacles.

Keywords: visibility query, moving object, spatial database

1 Introduction

Recent development in 3D modeling of urban environments and popularity of location based services have increased the opportunity to answer different real life queries involving the visibility of objects in the presence of 3D obstacles [1,5]. For example, in a smart city, surveillance cameras may need to track a moving car continuously, and then a security officer may want to find the cameras that produce the best view of the car at different timestamps in the presence of obstacles. Consider another scenario, where a motor racing is telecasted, and the director may want to continuously track the leading car and display the best view of the leading racer continuously. In each of these cases, the objective is to continuously find the viewpoint or camera location from a set of candidate locations that has the maximum visibility of the moving target at particular time instance. We call this new type of query *Continuous Maximum Visibility Query*

(*CMVQ*). In this paper, we investigate efficient techniques to answer the CMVQ query. At each timestamp and for the current position of the target, the *CMVQ* returns the query point that provides the maximum visibility. A generalization of *CMVQ* is *k-Continuous Maximum Visibility Query* or *kCMVQ* that finds k best query points (or cameras) in increasing order of their visibility of T.

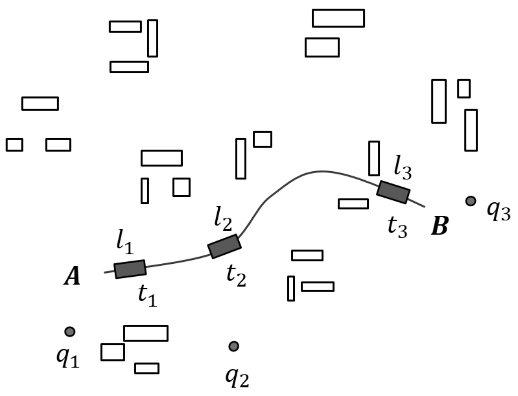


Fig. 1: A target object T moving along the trajectory AB

Figure 1 shows an example of a CMVQ query, where $Q = \{q_1, q_2, q_3\}$ are the locations of three security cameras, and O is a set of obstacles in a city. A security officer wants to track a moving car, T, by using the installed security cameras. At time t_1, target T is at location l_1 and q_1 provides the maximum visibility (or the best view) of T. But as the target changes its location and moves from l_1 to l_2 and then from l_2 to l_3, the camera that provides maximum visibility changes from q_1 to q_2 and then from q_2 to q_3, respectively.

The visibility of an object from a viewpoint depends on the distance, angle and existence of obstacles between the target and the viewpoint. Thus the main challenge of solving a CMVQ query is the computation of visibility of a target object for a given set of viewpoints by considering a large set of obstacles. Visibility computation has been extensively studied in computational geometry and computer graphics, and different algorithms have been proposed to solve the visibility problem. However, they rely on accessing all the obstacles [3, 14, 15], and propose in-memory algorithms. Moreover, they only consider the visibility in terms of binary measure, i.e., visible or non-visibile. An efficient approach was suggested in [10] that overcame these drawbacks for visibility queries for a static object. They compute visibility of a static target object from a set of query points and find relative ranking of these query points. This approach needs to update visibility of all the query points based on the obstacles that are inside the view of those query points. Applying this technique to solve the *kCMVQ* requires considering all the query points and obstacles within a certain range for each position of the target repetitively, which will incur high query processing and I/O overhead.

We have proposed techniques to solve *kCMVQ* queries with reduced processing time and I/O. The novelty of our approach is that we do not need to access all the obstacles and also do not require to update the visibility of the target with respect to all the query points. Moreover, our solution relies on some pre-computed data structure based on obstacles and query datasets to enhance the query processing time. At each position of the target object we only use those query points from which the target might be actually visible and consider only those obstacles that can affect the visibility of the query points under consideration. Also we have proposed an incremental approach that will avoid retrieving the same obstacle multiple times for consecutive positions of the target.

In summary, the contribution of the paper can be stated as follows.

- We introduce the novel problem of *Continuous Maximum Visibility (CMV)* query in a d-dimensional space.
- To solve CMV queries, we propose two different approaches. In the first approach, we resort to pre-computation to answer the query fast at the expense of high memory cost. In the second approach, we make a tradeoff between pre-computation and memory usage.
- We conduct an extensive experimental study on two real datasets that demonstrate efficiency of our algorithms.

2 Related Works

Visibility in Computational Geometry and Computer Graphics In computational geometry, several approaches have been proposed to construct visibility graph and visibility polygon [3,4,14,15]. The problem of computing visbility of input polygon P from a query point q, $V(q)$ was first addressed for simple polygons in [6]. A visibility graph is defined by a set P of n points inside a polygon Q where two points $p, q \in P$ are joined by an edge if the segment $pq \subset Q$ [4]. They have given a nearly optimal algorithm for simple polygons and introduced a notion of *robust* visibility for polygons with holes (non-simple polygons). Different works [2,15] showed that query time for simple polygons can be reduced to logarithmic bound using polynomial time preprocessing. In [3], Asano et al. presented an algorithm with $O(n)$ query time with $O(n^2)$ preprocessing time and space. These algorithms efficiently constructs the visibility polygon with the expense of heavy preprocessing and/or accessing all obstacles present in dataset, which makes this technique inappropriate for many spatial database applications that handles a large number of obstacles.

Although computation of visibility is also an active study topic [13] in computer graphics, their main focus is rendering a scene while we focus on calculating the visibility of a specific target object from various query points.

Visibility in Spatial Queries *kNN* queries have been extensively studied in spatial databases. Variants of *kNN* query consider the effect of obstacles while measuring the distance between two objects. Visible NN (VNN) query [11] finds the NN that is visible to a query point. Continuous Obstructed NN (CONN) query [7] retrieves the nearest neighbor of each query point according to the obstructed distance. Continuous Visible NN (CVNN) query [8] retrieves visible nearest neighbor along a query line segment in the presence of obstacles.

A new type of query called the k Maximum Visibility (kMV) query was introduced in [10], where for a given fixed target T, a set of obstacles O, and a set of query points Q, the kMV query returns k query points in the increasing order of their visibilities to T. Another study [5] quantified the visibility of a target object from the surrounding area with a visibility color map (VCM). But their approach has no notion of partial visibility and has not considered the case of a moving target. A method is presented in [12] that considers partial visibility of the target and support incremental updates of the *VCM* if the target moves to near-by positions.

In this paper, we focus on answering kMV query for a moving target, that is, we need to continuously find k query points based on the visibility of the target T for each time instance.

3 Problem Formulation

Let O be a set of obstacles stored in *R*-Tree*, Q be a set of query points or candidate locations and T be a moving target object in a dataspace. In this scenario *Continuous Maximum Visibility Query (CMVQ)* can be defined as follows:

Definition 1 (*CMVQ*). *Given a set Q of n query points $\{q_1, q_2, ..., q_n\}$, a set O of m obstacles $\{o_1, o_2, ..., o_m\}$ and a moving target object T that is at location l_i at time instance t_i in a d-dimensional space R^d, the Continuous Maximum Visibility Query (CMVQ) for T continuously returns the query point $q \in Q$ at each time instance where visibility$(q) \geq$ visibility(q_j) where $q_j \in Q \setminus q$.*

The generalization of *CMVQ* is *kCMVQ*, where k query points $Q'\{q'_1, q'_2, ..., q'_k\}$ are returned at each time instance, where *visibility*$(q'_i) \geq$ *visibility*(q'_j), $1 \leq i < j \leq k$, and *visibility*$(q') \geq$ *visibility*(q''), for each $q' \in Q'$, $q'' \in Q \setminus Q'$.

3.1 Preliminaries

The visibility of a target from a viewpoint varies with the distance and the angle between the viewpoint (or camera) and the target. Similar to [5], in this paper, by considering both of the above factors, we quantify the visibility as visual angle that varies with both the angle and distance between the target and the viewpoint. We assume each camera q has a *field of view* (FOV), a region beyond which nothing is visible to that camera. If we use the visual angle measure, a FOV produces a circular sector in 2D and a spherical sector in 3D which we call the *visible region* (VR). Any target object lying in this region is *point to point visible* from q in the absence of obstacles. For simplicity, we assume that an FOV is triangular in 2D and conical in 3D.

Aggregated Visible Region and Potentially Visible Query Point Set An *AVR* is a region formed by overlapping *VRs* of a set of query points and is disjoint from all other *AVRs*. Any point within an *AVR* is visible only from a specific set query points. We call this set of query points *potentially visible query point set* (PVQS). While computing the visibility of a target object T inside an *AVR*, we only need to consider query points in the corresponding *PVQS*. The maximum number of *AVRs* generated by *VRs* of n query points in 2D is $\frac{3n(3n+1)}{2} + 1$ since each *VR* is bounded by 3 lines [9].

Blocking Set and Aggregated Blocking Set The visibility of a target object T from a query point q_i is affected by the obstacles that lie in the *visible region* V_i of q_i. All other obstacles can be safely pruned while computing the visibility of T from q_i. We call this set of obstacles a *blocking set* (BS) of that query point. Each *AVR* A_i has an associated *PVQS* P_i. Since the target will be visible only from the query point $q \in P_i$, the only obstacles to consider are the *blocking sets* of each q. Thus, by combining *BS*s of each query point $q \in P_i$, we get an *aggregated blocking set (ABS)* for the *AVR*.

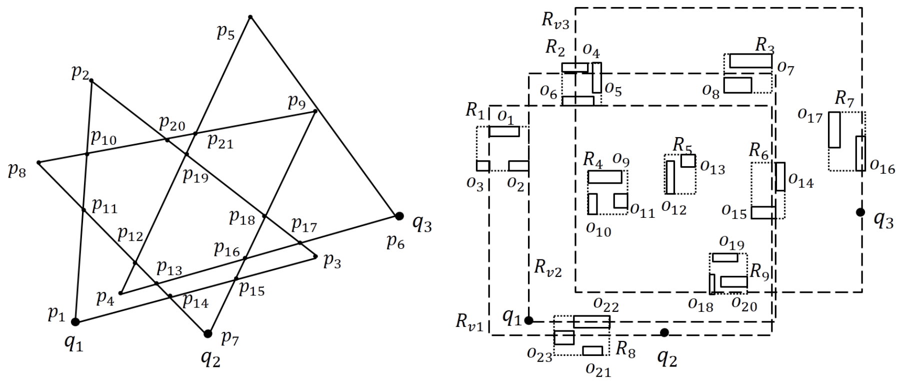
4 Our Approach

In this section we present an algorithm to efficiently solve *kCMVQ*. The key challenge of solving *kCMVQ* is that since the target object is moving, we need to re-compute *k*MV query and update the ranking of query points for every position change of the target. A straightforward approach is to run the *kMVQ* [10] every time when T moves. This approach result in high processing and I/O overhead as for each position change of the target we need to consider all obstacles and query points that fall within the visible range. Since there can be many common obstacles that fall within the range of two consecutive target locations, the retrieval of same object occur multiple times.

To overcome the above limitation, we introduce an approach with reduced I/O and processing overhead. To answer a *kCMVQ* we rely on pre-computed data structure based on our obstacle and query datasets.

Preprocessing A moving target may change its position fast. For each position update, finding the query points from which T becomes visible and retrieving the required obstacles from the database by traversing the *R*-Tree* is costly and time consuming. Since in our problem setting, we assume query points and obstacles are static, we can precompute certain steps of our approach. Specifically, we precompute (i) the set of *AVR* and their corresponding *PVQS*, as when T is in an *AVR*, we only need to consider the query points of its corresponding *PVQS*, and (ii) the *BS* of each query point $q \in Q$, so that we do not need to consider the obstacles that has no affect on the visibility of T while processing a query.

As the first step, we construct the *AVR* set for the dataspace from the *VRs* of the queries in Q. The query points whose overlapping *VRs* form an *AVR* are recorded as its *PVQS*. We do not consider obstacles while computing *AVRs*. Second, we generate the set of *BS* for every query. To reduce the complexity, we maintain the *minimum bounding rectangle (MBR)* of VR, and use it to compute the set of obstacles that are inside the VR. Note that, for each obstacle in a *BS*, we store block number and entry number of the *R*-Tree*, and thus we can avoid the *R*-Tree* traversal.



p_1 p_2 p_3 p_4 p_5 p_6 p_7 p_8 p_9 p_{10} p_{11} p_{12} p_{13} p_{14} p_{15} p_{16} p_{17} p_{18} p_{19} p_{20} p_{21}
q_1 q_2 q_3
R_{v1} R_{v2} R_{v3}
R_1 R_2 R_3 R_4 R_5 R_6 R_7 R_8 R_9
o_1 o_2 o_3 o_4 o_5 o_6 o_7 o_8 o_9 o_{10} o_{11} o_{12} o_{13} o_{14} o_{15} o_{16} o_{17} o_{18} o_{19} o_{20} o_{21} o_{22} o_{23}

(a) Aggregated Visible Regions (b) Computing Blocking Sets

Fig. 2: Building Aggregated Visible Regions

Example 1. We will next show the construction of *AVR*, *PVQS*, and *BS* using an example. Figure 2(a) shows locations of a set Q of three query points $\{q_1, q_2, q_3\}$. First we take q_1 and its *VR* ,V_1. Since *AVR* set A is empty at this point, V_1 (p_1, p_2, p_3) itself is added as an *AVR*. Next q_2 is considered and its *VR* V_2 (p_7, p_8, p_9) intersects with existing *AVR*s and creates new *AVR*s. For example, V_2 intersects with (p_1, p_2, p_3) and divides it into four regions $(p_1, p_{11}, p_{14}), (p_2, p_{10}, p_{20}), (p_3, p_{15}, p_{18})$ and $(p_{10}, p_{11}, p_{14}, p_{15}, p_{18}, p_{20})$. For each *AVR* its *PVQS* is updated, e.g., the region bounded by $(p_{10}, p_{11}, p_{14}, p_{15}, p_{18}, p_{20})$ has *PVQS* $\{q_1, q_2\}$ since overlapping *VRs* of these query points form the *AVR*. This process is repeated until all query points are considered. To compute *BS*, we start with q_1 and compute *MBR* of *VR* V_1. Let this *MBR* be R_{V1}. We perform a range query in the *R*-tree* to find out the obstacles inside R_{V1}. We can see from Fig. 2(b) that *BS* of q_1 is $B_1 = \{o_1, o_2, o_9, o_{10}, o_{11}, o_{12}, o_{13}, o_{15}, o_{18}, o_{19}, o_{20}, o_{22}, o_{23}\}$. We compute *BS* of q_2 and q_3 in a similar manner.

4.1 AVR-BS Incremental Approach

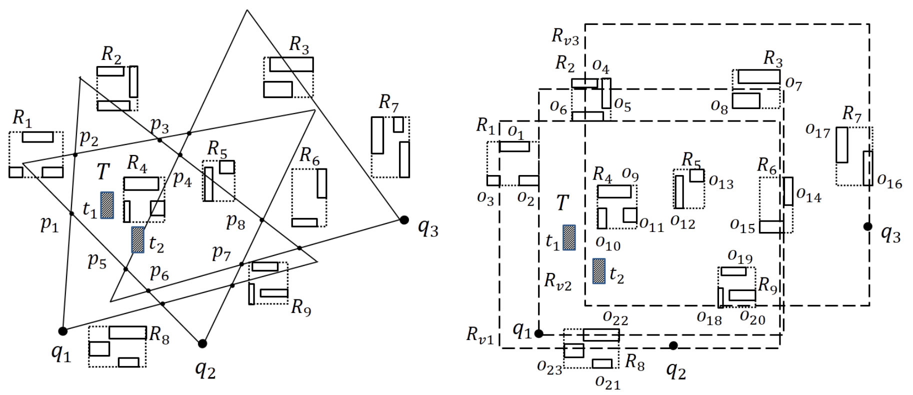
We divide the dataspace into *AVRs* such that when T is inside an *AVR*, the set of query points from which T is visible and set of obstacles that affect the visibility of T remain same. Thus we do not need any extra I/O as long as the target does not change its current *AVR*. Since neighboring AVRs are likely to have many queries in common in their PVQS, we re-use the already retrieved obstacles for those common queries when T moves to a neighboring AVR.

Algorithm 1: kCMVQ-Incremental(T,k)

```
Input: $T$ a target object, $k$
Output: $L$ a set of $k$ query points ordered by visibility
1  $A_c \leftarrow getCurrentAVR(T)$; $Q_c \leftarrow getCurrentQs(A_c)$;
   $B_a \leftarrow getBlockingSets(Q_c)$
2  while $true$ do
3      if $getTargetMovement(T) < th$ then $continue$
4      if $isAVRChanged(T,A_c)$ then
5          $A_c \leftarrow updateAVR(T)$; $Q_n \leftarrow getCurrentQs(A_c, T)$
6          $Q_{com} \leftarrow Q_n \cap Q_c$
7          $Q_{new} \leftarrow Q_n \setminus Q_{com}$
8          for $q \in Q_{new}$ do $B_a \leftarrow B_a \cup q.B$
9          $Q_{obs} \leftarrow Q_c \setminus Q_{com}$
10         for $q \in Q_{obs}$ do $UpdateABS(B_a, q, Q_n)$
11         $Q_c \leftarrow Q_n$
12     $L \leftarrow ComputeVisibility(T, Q_c, B_a, k)$
```

Algorithm 1 shows our incremental approach for answering a *kCMV* query. It takes target object T and positive integer k as input and reports k best query points in set L ordered by the visibility of T. We update L only when T has moved at least a threshold amount th. Algorithm starts with retrieving current *AVR* A_c where T resides, PVQS Q_c of A_c and *ABS* B_a for all query points in Q_c (Line 1). If T resides on the border of multiple *AVRs*, A_c stores the set of *AVRs*. Q_c is constructed by union of *PVQS* of each *AVR* in A_c. The *while* loop iterates as long as the target continues querying in the dataspace. If target changes its position within the current *AVR*, we compute visibility in Line 12. If T moves to another *AVR*, we update current *AVR* A_c and compute a new query set Q_n by combining *PVQS* of *AVRs* that overlap with T. Line 6 calculates Q_{com} whose corresponding *blocking set*s are already retrieved in previous iteration. At Line 7 Q_{new} holds the new query points from which T is now visible. Line 8 updates *ABS* B_a. With the region change, T may also become invisible to some query points from which it was visible earlier. Set of such query points Q_{obs} is computed in Line 9. For each such query point *ABS* is updated, i.e., we remove obstacles from B_a that do not affect the visibility anymore. This is not a set minus operation since an obstacle can be in multiple *blocking sets*. So removing *BS* of a query point may remove some obstacles that are part of *BS* of some other query points from which T is still visible. Line 12 then uses Q_c and A_b to compute visibility.

Example 2. We explain the incremental approach using *AVRs* A_1 $(p_1, p_2, p_3, p_4, p_5)$ and A_2 $(p_4, p_5, p_6, p_7, p_8)$. In Fig. 3(a) at time t_1 target T is completely inside A_1. The corresponding *PVQS* is $P_1 = \{q_1, q_2\}$ and *ABS* is $B_{a(1)} = \{B_1 \cup B_2\}$. We retrieve all the objects of $B_{a(1)}$ and compute visibility of the target from the query points in P_1. Then at t_2, T moves to the region boundary between the *AVRs* A_1 and A_2. At this point we need to consider query point sets $P_{1\cup2} = P_1 \cup P_2$ and set of obstacles $B_{a(1\cup2)} = B_{a(1)} \cup B_{a(2)}$. T is now visible from a new query point q_3 that was not in P_1 and thus the obstacles in its *BS*



(a) AVR-BS (b) VR-BS

Fig. 3: An Example Scenario of Approaches

B_3 are not in the *ABS* that was computed in the previous step. We can now update ABS as $B_{a(1\cup 2)} = B_{a(1)} \cup B_3$. We do not need to retrieve obstacles that are already in B_1 and B_2.

4.2 VR-BS Approach

The incremental approach computes AVRs based on the given set of query locations. If a large number of query points are densely placed in the dataspace, the number of AVRs can be huge. In some scenario, it may happen that the number of AVRs may outnumber the number of obstacles in the dataset. For such cases, the previous approach may not perform well. Thus, in this section, we introduce another approach that preprocesses obstacle set to compute *BS* of each query point and avoids computation of AVRs.

In this approach, we maintain an *R*-Tree* that holds *MBRs* of *VRs* (VRMBR) of query points. We search the *R*-Tree* to find *VRMBR*s with which target T overlap, and then we add the corresponding query point to *potentially visible query point set* (PVQS).

Algorithm 2: κCMVQ-VRBS(T,k)

```
  Input: $T$ a target object, $k$
  Output: $L$ a set of $k$ query points ordered by visibility
1 while true do
2     if $getTargetMovement(T) < th$ then continue
3     $Q_c \leftarrow findActiveQ(T, VRMBRTree); B_a \leftarrow \emptyset$
4     for $q \in Q_c$ do $B_a \leftarrow B_a \cup q.B$
5     $L \leftarrow ComputeVisibility(T, Q_c, B_a, k)$
```

Algorithm 2 shows the steps of *VR-BS* approach. Lines 1 - 5 iterate as long as the target continues querying. If T does not move a threshold distance, we do not update existing ranking of query points in L. Line 3 computes Q_c by traversing the *R*-Tree* of VRMBRs and finding query points whose *VRMBRs* have non-empty intersecting region with T. The *ABS* B_a is reconstructed in Line 4. With the reduced set of query points Q_c and the reduced set of obstacle B_a, we compute visibility of target in Line 5.

Example 3. Figure 3(b) shows *VRMBRs* of query point set $Q = \{q_1, q_2, q_3\}$. For target position at time t_1 we traverse the tree to find that T lies inside *VRMBR* of all three query points. Therefore *PVQS* $P = \{q_1, q_2, q_3\}$. Since $P \subset Q$, *ABS* $B_a = B_1 \cup B_2 \cup ... \cup B_{|P|}$. Hence at t_1 $B_a = B_1 \cup B_2 \cup B_3$. When target moves to a new location depicted at time t_2, we again traverse *R*-Tree* to find that T lies inside *VRMBR* of q_3 only. Hence *PVQS* $P = \{q_3\}$ and *ABS* $B_a = B_1$.

5 Experimental Evaluation

We evaluate performance of our proposed algorithms for answering the *kCMV* query with two real datasets, for both *Uniform*(U) and *Zipf*(Z) distribution of query point locations. Two real datasets are, the British[3] dataset, representing 5985 data objects obtained from British ordnance survey[4] and Boston dataset represents 130,043 data objects in Boston downtown[5]. Random paths in search space are generated to simulate movement of a target. We assume a total of 300 queries requested by target while it is moving in the dataspace. All obstacles are stored in a *R*-Tree* with the disk page size fixed at 1KB and block size fixed at 256B. The target size is kept fixed at 2 units. The algorithms are implemented in C++, and the experiments are conducted on a core i5 2.67GHz PC with 4GB RAM, running 64 bit Microsoft Windows 8.1.

Parameter	**Range**	**Default**
dataset size	British (6K), Boston(130K)	
number of query points n_q	100, 200, 300, 400, 500	400
field of view fov	30, 45, 60, 75, 90	30
maximum distance of visibility D	$300, 400, 500, 600, 700$	500

Table 1: Parameters

5.1 Performance Evaluation

We conduct four sets of experiments to evaluate the performance of AVR-BS and VR-BS, which are referred as A and V respectively, in the following figures. In each set of experiments, one parameter is varied while all other parameters are set to their default values. In all cases AVR-BS outperforms VR-BS in terms of execution time and I/O cost with the expense of high pre-computation cost. For each experiment we have evaluated the results of 10 iterations and reported average performance.

[3] http://www.citygml.org/index.php?id=1539
[4] http://www.ordnancesurvey.co.uk/oswebsite/indexA.html
[5] bostonredevelopmentauthority.org/BRA_3D_Models/3D-download

Straightforward Approach We have compared our proposed approaches with the straightforward approach, which finds *kMVQ* at each time instance. The straightforward approach is not scalable for a moving target and is outperformed by both *AVR-BS* and *VR-BS* algorithms. *AVR-BS* is about 250 times (British) and 1500 times (Boston) faster than the straightforward approach, whereas *VR-BS* is about 125 times (British) and 128 times (Boston) faster than the straightforward approach. *AVR-BS* has 48 times (British) and 28 times (Boston) less I/O cost than the straightforward approach. *VR-BS* has similar I/O cost to straightforward approach. *VR-BS* first traverses a *R*-Tree* to find out the query points whose *VRs* intersect T and then retrieves obstacles that belong to *BS* of these query points. This results in high I/O cost. But *VR-BS* uses reduced set of obstacles and set of query points during visibility computation, which explains its lower processing time than the straightforward approach.

	British		Boston	
	Time (ms)	I/O (avg)	Time (ms)	I/O (avg)
Straightforward	778.78	24.92	21946.42	365.82
AVR-BS	3.31	0.54	13.93	13.91
VR-BS	6.36	47.05	170.31	354.72

Table 2: Comparison with Straightforward Approach

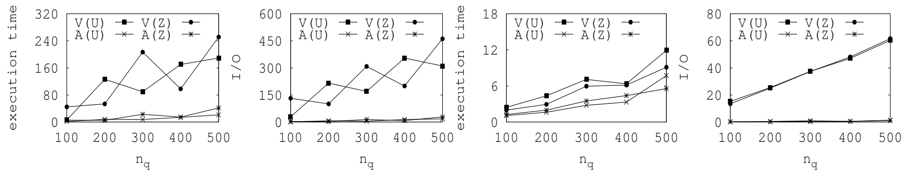
Preprocessing. Table 3 shows that precomputation time in milliseconds (ms) for both *AVR* and *BS* for default values of parameters. Both costs increase with the increase in n_q, fov or D. For more query points, more polygon intersections take place and more *AVRs* are generated. For *AVR* both datasets show that the cost is more for Z as query points are clustered nearby and their *VRs* are more likely to overlap and is similar for both datasets.

	British		Boston	
	Uniform	Zipf	Uniform	Zipf
AVR	56135.66	85149.0	58718.67	92671.67
BS	1438.0	1734.33	12320.33	14827.67

Table 3: Preprocessing time for AVR and BS

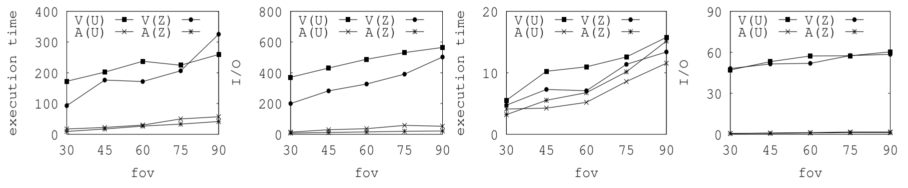
Varying No. of Query Points. Figure 4 shows that with the increase of number of query points, execution time and I/O cost both increase for both *AVR-BS* and *VR-BS*. *VR-BS* approach has higher execution time than *AVR-BS* as *PVQS* and *ABS* needs to be recomputed at each position change of the target. On average *AVR-BS* is 10 times (Boston) and 2 times (Bristish) faster than *VR-BS*. *VR-BS* incurs higher I/O cost than *AVR-BS* as building *PVQS* each time requires the *R*-Tree* traversal. Execution time of *VR-BS* is similar to that of *AVR-BS*, for British dataset. This is because British dataset has a smaller search space and with the increase in number of query points in this smaller space, *VR*s overlap more frequently and hence number of *AVRs* increases. Therefore, a moving target changes *AVR* more often, which requires more computation in AVR-BS.

Varying Field of View. Figure 5 shows that execution time increases for both approaches with increase in *fov*. More obstacles come into consideration for each



(a) Time (b) I/O (c) Time (d) I/O

Fig. 4: Effect of varying n_q for Boston (a-b) and British (c-d) datasets



(a) Time (b) I/O (c) Time (d) I/O

Fig. 5: Effect of varying fov for Boston (a-b) and British (c-d) datasets

query point as *fov* increases. Effect of varying *fov* is similar to the effect of varying n_q. On an average *AVR-BS* performs approximately 10 times and 1.5 timesfaster than $f$or Boston and British datasets, respectively.

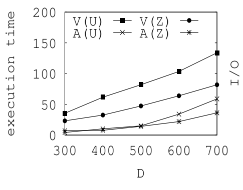
Varying Maximum Distance of Visibility. Figure 6 shows that execution time increases for both approaches in both Boston and British datasets. *AVR-BS* outperforms *VR-BS* in all cases. Execution times for both approaches are similar in British dataset; the underlying reason can be high density of query points.

Varying Paths. Figure 7 shows the result of average execution time and I/O cost of 10 different paths. In all cases I/O cost for VR-BS approach is higher than that of AVR-BS as expected. On average *AVR-BS* performs 7 times and 2 times faster than *VR-BS* for Boston and British datasets, respectively.

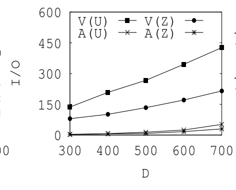
6 Conclusion

In this paper, we have introduced a new type of query, namely *k Continuous Maximum Visibility Query* that continuously finds the best viewpoint for a moving target in the presence of obstacles. To efficiently answer a *kCMV* query, we have proposed two approaches. The first approach, *AVR-BS*, relies on pre-computation to provide fast answer to queries. On the other hand, the second approach, *VR-BS*, makes a tradeoff between pre-computation and memory usage while processing queries. Our experimental results show that *AVR-BS* is two to three orders of magnitude faster than the straightforward approach, and *VR-BS* is at least one order of magnitude faster than the straightforward approach.

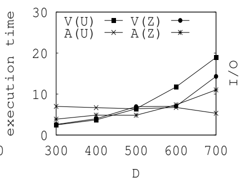
Acknowledgements. This research is conducted at the department of Computer Science and Engineering, Bangladesh University of Engineering and Technology (BUET). This research is supported by the ICT Division - Government of the People's Republic of Bangladesh.



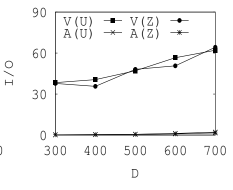
(a) Time



(b) I/O

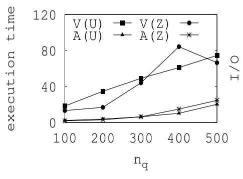


(c) Time

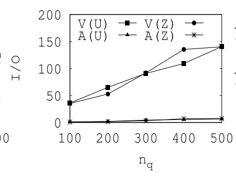


(d) I/O

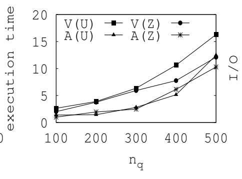
Fig. 6: Effect of varying D for Boston (a-b) and British (c-d) datasets



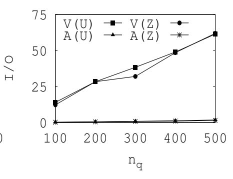
(a) Time



(b) I/O



(c) Time



(d) I/O

Fig. 7: Effect of varying paths for Boston (a-b) and British (c-d) datasets

References

1. M. E. Ali, E. Tanin, R. Zhang, and L. Kulik. A motion-aware approach for efficient evaluation of continuous queries on 3d object databases. *VLDB J.*, 19(5):603–632, 2010.
2. B. Aronov, L. J. Guibas, M. Teichmann, and L. Zhang. Visibility queries and maintenance in simple polygons. *DCG*, 27(4):461–483, 2002.
3. T. Asano, L. J. Guibas, J. Hershberger, and H. Imai. Visibility of disjoint polygons. In *Algorithmica*, pages 49–63, 1986.
4. B. Ben-Moshe, O. Hall-Holt, M. J. Katz, and J. S. B. Mitchell. Computing the visibility graph of points within a polygon. In *SCG*, pages 27–35, 2004.
5. F. M. Choudhury, M. E. Ali, S. Masud, S. Nath, and I. E. Rabban. Scalable visibility color map construction in spatial databases. *Inf. Syst.*, 42:89–106, 2014.
6. L. S. Davis and M. L. Benedikt. Computational models of space: Isovists and isovist fields. In *Computer Graphics and Image Processing*, page 4972, 1979.
7. Y. Gao and B. Zheng. Continuous obstructed nearest neighbor queries in spatial databases. In *SIGMOD*, pages 577–590, 2009.
8. Y. Gao, B. Zheng, W. Lee, and G. Chen. Continuous visible nearest neighbor queries. In *EDBT*, pages 144–155, 2009.
9. R. Graham, D. Knuth, and O. Patashnik. *Concrete Mathematics*. Addison-Wesley.
10. S. Masud, F. M. Choudhury, M. E. Ali, and S. Nutanong. Maximum visibility queries in spatial databases keys. In *ICDE*, pages 637–648, 2013.
11. S. Nutanong, E. Tanin, and R. Zhang. Visible nearest neighbor queries. In *DASFAA*, pages 876–883, 2007.
12. 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 *SSTD*, pages 197–215, 2015.
13. L. Shou, Z. Huang, and K.-L. Tan. Hdov-tree: The structure, the storage, the speed. *In ICDE*, 2003.
14. S. Suri and J. ORourke. Worst-case optimal algorithms for constructing visibility polygons with holes. In *SCG*, pages 14–23, 1986.
15. A. R. Zarei and M. Ghodsi. Efficient computation of query point visibility in polygons with holes. In *SCG*, pages 6–8, 2005.