Summary Report

A General Technique for Top-k Geometric Intersection Query Problems *- Saladi Rahul, Ravi Janardan*

Aman Choudhary amanc@iisc.ac.in
Shashank Singh shashanksing@iisc.ac.in
Indian Institute of Science, Bangalore, India

Geometric Intersection Query (GIQ) problems aim to report geometric objects that are intersected by a given query geometric object in d dimensional space. The top-k GIQ problem has weighted objects and its goal is to report the top k objects w.r.t weights. The authors explore this less attended area and attempt to provide a generic solution agnostic of underlying specifics of objects and query.

1 Problem Statement

In order to formulate the Top-k GIQ problem and solution, we first understand the underlying GIQ problem.

1.1 GIQ

In Geometric Intersection Query (GIQ) problem, we are given a set, $\mathcal{A}$, of n geometric objects (for e.g., points, lines, hyper-rectangles line segments, balls, etc.) in $\mathbb{R}^d$. Given a query geometric object, q, we want to report or count the objects of $\mathcal{A}$ that are intersected by q.

In the *reporting* version, the output is a list of all objects of $\mathcal{A}$ that are intersected by q; while in the *counting* version, the output is simply the number of objects of $\mathcal{A}$ intersected by q.

1.2 Top-k GIQ

In Top-k Geometric Intersection Query (Top-k GIQ) problem, we are given a set $\mathcal{A}$, of n geometric objects in $\mathbb{R}^d$. Given a query geometric object, q, and integer $k > 0$, we want to report the k largest-weight objects that are intersected by q.

2 Solution to Top-k GIQ Problem

The top-k GIQ algorithm proposed by the authors uses the solutions to the counting and reporting versions of the underlying GIQ problem as *blackboxes*. These blackboxes can be substituted with appropriate subroutines depending on the specific flavour of top-k GIQ problem which we intend to solve. This elegantly makes this solution agnostic to the geometry of the objects and query and makes it a generic algorithm. The same idea is illustrated in *Figure 1*.

Notations: For simplicity of correspondence with the paper, we try our best to use the same notations as done in the paper.

- $\mathcal{A}$: Set of n geometric objects $\{a_1, a_2, ..., a_n\}$ in $\mathbb{R}^d$ $(d \geq 1)$
- q : Query region
- k : Number of objects to return at max
- $\mathcal{A}$ (q) : Set of objects intersected by query, q
- $\mathcal{D_C}$: Data structure used for answering counting version of underlying GIQ problem
- $\mathcal{D_R}$: Data structure used for answering reporting version of underlying GIQ problem
- $\mathcal{T}$: A balanced binary tree whose leaves store the objects of $\mathcal{A}$ in non-increasing order of weight, from left to right. Each node v of $\mathcal{T}$, points to an instance, ${\mathcal{D_C}}^v$, of the structure $\mathcal{D_C}$, which is constructed on the objects stored in v's subtree.
- $\mathcal{T}'$: A balanced binary tree whose leaves store the objects of $\mathcal{A}$ in non-increasing order of weight, from left to right. Each node v of $\mathcal{T}'$, points to an instance, ${\mathcal{D_R}}^v$, of the structure $\mathcal{D_R}$, which is constructed on the objects stored in v's subtree.
- $ptr(v)$: For every object a_i stored at node v in $\mathcal{T}$, and node v' in $\mathcal{T}'$, we store a pointer ptr from v to v', i.e. $ptr(v) = v'$.
- Z : Set of canonical nodes in $\mathcal{T}'$

The main algorithm is divided into three steps, as described below.

2.1 Step 1: Perform initial check

Let r be the root of $\mathcal{T}$. The first thing we do is to handle the case where we have k or fewer objects in $\mathcal{A}$ that are intersected by q, i.e $|\mathcal{A}(q)| \leq k$. This check can be performed by querying the associated structure, ${\mathcal{D_C}}^r$. If the above condition is met, then the solution to top-k GIQ problem, will include all the elements of $\mathcal{A}(q)$. Thus, we simply report all the objects in $\mathcal{A}(q)$, by querying ${\mathcal{D_R}}^r$.

2.2 Step 2: Find a threshold object

The leaf nodes of $\mathcal{T}$ store the objects of $\mathcal{A}$ in non-increasing order. The goal of this step is to compute the k^{th}-leftmost object in this ordering, that is intersected by q. We call this object, the threshold object a_t. In order to locate this object, we will follow a **recursive binary search** procedure, wherein we would discard one half of the objects at each step, thus narrowing down our search as we progress. This is done unil we reach a_t. Let us define the following

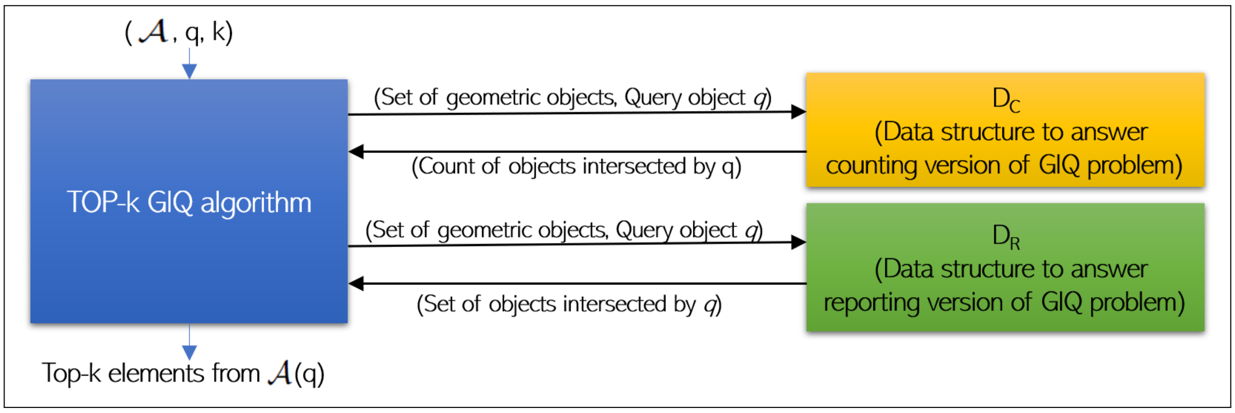
($\mathcal{A}$, q, k)
TOP-k GIQ algorithm
(Set of geometric objects, Query object q)
(Count of objects intersected by q)
(Set of geometric objects, Query object q)
(Set of objects intersected by q)
D_C (Data structure to answer counting version of GIQ problem)
D_R (Data structure to answer reporting version of GIQ problem)
Top-k elements from $\mathcal{A}$(q)

Figure 1: *A diagram illustrating the interaction between Top-k GIQ algorithm with underlying solutions to GIQ problem*

notation:

$search(T, k)$: The problem of locating the k^{th} leftmost object which is intersected by q, in balanced binary search tree T.

We start with the following problem instance: $search(\mathcal{T},k)$. Let r_L and r_R be the left and right children respectively, of the root r of T. Also, let T_L and T_R be the subtrees rooted at r_L and r_R respectively. The search begins by querying ${\mathcal{D}_C}^{r_L}$ to calculate the number of objects in T_L that are intersected by q. Let us denote the result of the query by $T_L(q)$. Now, two cases arise:

- **Case 1:** $k \le T_L(q)$ In this case, a_t would lie in T_L, and hence we discard T_R from our search. The original problem is now reduced to the subproblem $search(T_L, k)$
- **Case 2:** $k > T_L(q)$ In this case, a_t would lie in T_R, and hence we discard T_L from our search. The original problem is now reduced to the subproblem $search(T_R, k - T_L(q))$

Thus, we progressively perform binary search till we reach a_t.

2.3 Step 3: Report top-k objects

Step 2 ends with the computation of the threshold object a_t in $\mathcal{T}$. Let the object a_t be stored in node u in $\mathcal{T}$. We follow $ptr(u)$ to reach its corresponding node u' in $\mathcal{T}'$. Then, we walk from u' to r', the root of $\mathcal{T}'$, by following the parent pointers. Let the path thus traced be denoted by Π. We include all the nodes v' in $\mathcal{T}'$ into set Z, such that v' is the left child of a node lying on Π, but itself is not on Π. We also include the leaf node u' into Z. The set Z, thus obtained is a collection of *canonical nodes*.

The set of objects stored at the leaf nodes of $\mathcal{T}'$ to the left of u', together with the object a_t stored at u' form a superset, S of our solution set. Every object $a_i \in S$ satisfies the property $w(a_i) \ge w(a_t)$. All, we need to do is extract the objects from S, that intersect with the query object q. In order to accomplish this, for each $v' \in Z$, we query ${\mathcal{D}_\mathcal{R}}^{v'}$ with q, which causes all objects in v''s subtree that are intersected by q to be reported. Thus, we obtain our required top-k objects in $\mathcal{A}$.

3 Our attempt at the problem

- The initial approach that came to our mind was as follows:

  (i) Given a set of geometric objects $\mathcal{A}$ and query object q, we would first extract out the objects which are intersected by the query, i.e. compute the set $\mathcal{A}(q)$.
  (ii) In the next step, we would find the k^{th} largest object in the set $\mathcal{A}(q)$ using suitable selection algorithm. Let us denote this object by a_k.
  (iii) Finally, we would report all objects from $\mathcal{A}(q)$, which have their weights greater than that of a_k.
- Coincidentally, the paper goes on to discuss the same approach, using this naive approach as a benchmark to contrast the performance of proposed solution Top-k GIQ problem. We did try to come up with a better solution than this naive version, but unfortunately weren't able to come up with something concrete.

4 What we liked about the paper?

- The paper presents a generic algorithm, which is agnostic of the underlying geometric object. It provides a unified approach to deal with a multitude of seemingly different objects by abstracting away details of the implementations of the underlying GIQ problem. Abstraction is a cornerstone in the field of computer science, and this paper uses it beautifully.
- The paper does its best to reuse and build-upon solutions to existing problems. To illustrate a few examples, Top-k circular range search as well as Top-k circular point enclosure search are solved, via first, transforming them to a special version of the top-k halfspace range searching problem. Similarly, Top-k interval intersection search is dealt with by reducing to an instance of top-k orthogonal range search.
- It establishes the relative hardness between the GIQ and top-k GIQ problem. In doing so, it thus sets up a clear bound to what can be aimed to achieve in future work.

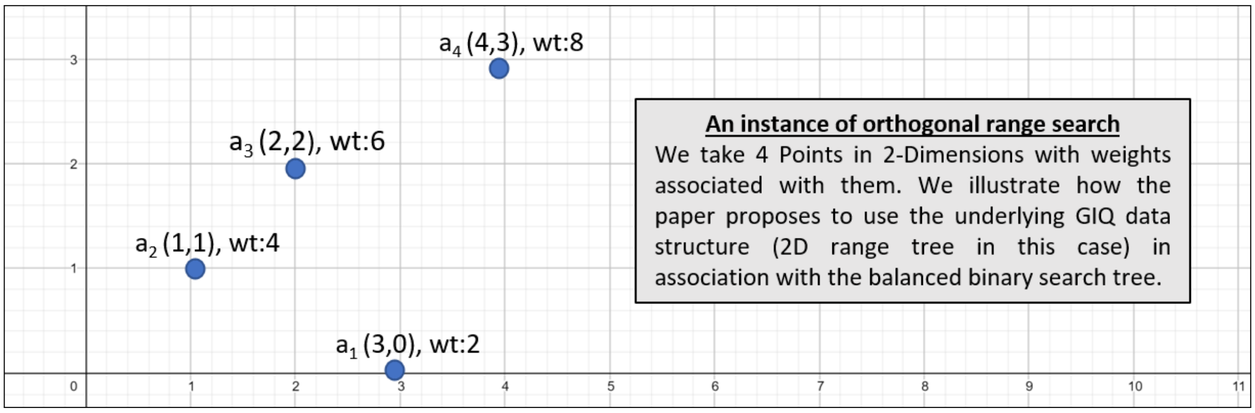

Figure 2: *A sample of* 4 *weighted point objects in* 2*D space used in* $Figure$ 3

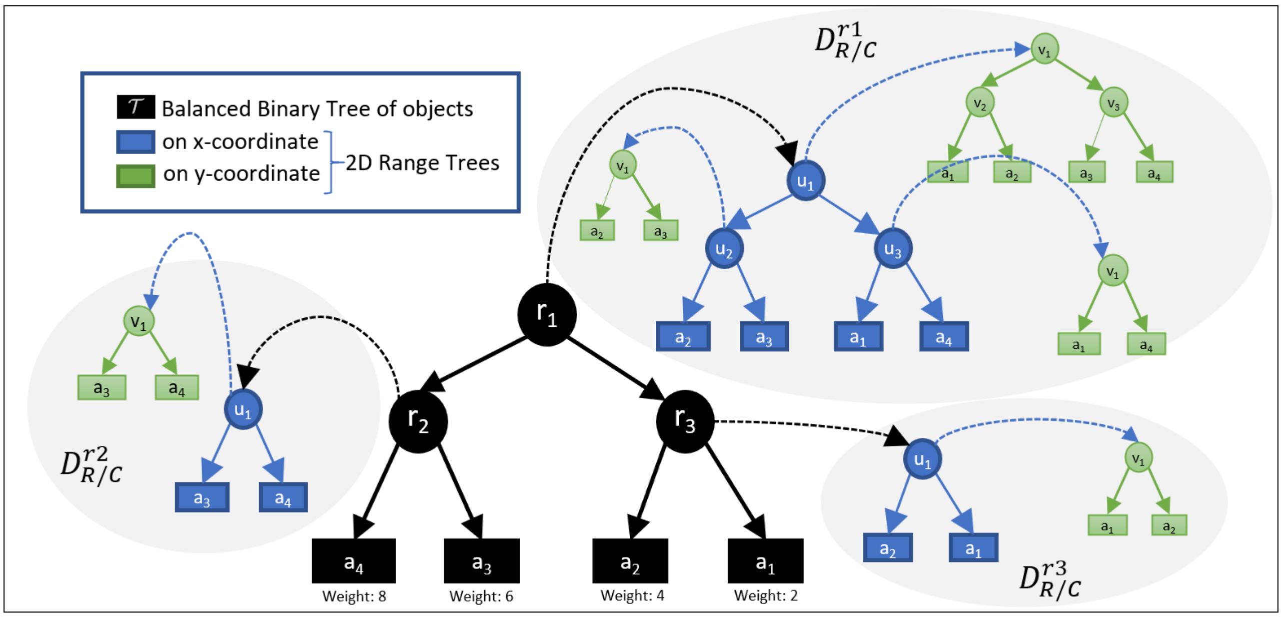

Figure 3: *Illustration of an instance of the pre-processed data structure based on objects shown in* $Figure$ 2

- It introduces us to some exotic data structures, many of which we were unaware of, such as BB(α) trees, R-trees, etc.

5 Challenges faced

- We faced difficulty in comprehending how the structures $\mathcal{D}_\mathcal{C}$ and $\mathcal{D}_\mathcal{R}$ associated with Top-k halfspace range searching were implemented. The paper refers the reader to the supporting paper about *Dynamic half-space range reporting* but we found it to be mathematically rigorous and quite challenging to follow.
- The paper leverages transformation techniques like *lifting* and *point-hyperplane duality* to reduce Top-k circular range search as well as Top-k circular point enclosure to the Top-k halfspace range searching problem. These techniques were quite involved and we were not able to completely understand them.

6 Proposals for Future work

- The complexity of the top-k GIQ problem is upper-bounded by the complexity of the underlying GIQ problem times a logarithmic factor. This is no doubt a very attractive result, but we should always strive to make our algorithms faster. The paper, in its analysis of hardness, establishes a lower bound for the algorithm in terms of the complexity of underlying GIQ problem. One possible future work could be try and reduce the logarithmic factor.
- The paper comes up with solutions to several different flavours of the top-k GIQ problem. Another aspect of applying these algorithms is to map real world problems to one of these top-k GIQ instance. Although, many applications have already been identified, we could pursue further and look for more domains and diverse use cases where the proposed algorithm could be put into practice.