Efficient Algorithms for Top-k Stabbing Queries on Weighted Interval Data

Daichi Amagata[✉], Junya Yamada, Yuchen Ji, and Takahiro Hara

Osaka University, Suita, Osaka, Japan
{amagata.daichi,yamada.junya,ji.yuchen,hara}@ist.osaka-u.ac.jp

Abstract. This paper addresses the problem of processing top-k weighted stabbing queries on interval data. A state-of-the-art algorithm for this problem incurs $O(n \log k)$ time, where n is the number of intervals, so it is not scalable to large n. We solve this inefficiency issue and propose an algorithm that runs in $O(\sqrt{n} \log n + k)$ time. Furthermore, we propose an $O(\log n + k)$ algorithm to further accelerate the search efficiency.

Keywords: Interval data · Stabbing · Top-k query

1 Introduction

Many applications deal with interval data, where an interval is a pair of left and right endpoints. For example, objects associated with time information (e.g., sales items and vehicles) are usually maintained in interval format (e.g., the left and right endpoints are activation and termination time, respectively [2,4–6]). In cryptocurrency and stock applications, the prices of cryptocurrencies and stocks vary continuously, and they record minimum and maximum prices (i.e., an interval) every certain time [10]. It is also intuitively known that each interval usually has a weight [1]. For instance, in the sales items and vehicles examples, the weights can be profits and the number of passengers, respectively.

To analyze the above weighted interval data, the following example query can be considered: Show k vehicles (e.g., trains) with the largest number of passengers at noon yesterday. This query helps consider a train operation plan and analyze train usage patterns for some events that occurred at a certain time. Motivated by such an application and usefulness, we address the problem of processing top-k weighted stabbing queries on interval data. Because a simple stabbing query does not consider weights and returns all stabbed intervals, applications cannot control the result size. That is, they may be overwhelmed by large result sizes, so the controllable result size (i.e., the top-k factor) is useful for such applications.

Given a set X of n weighted intervals and a query $q = (s, k)$ where s and k are respectively a query value and a result size, this query retrieves k intervals *stabbed by s* with the largest[1] weight among X. An interval $x \in X$ is stabbed by q iff $s \in [x.l, x.r]$, where $x.l$ and $x.r$ are the left and right endpoints, respectively. Because many applications deal with large sets of intervals (i.e., n is large), an efficient algorithm for this problem is required. However, designing such an algorithm is non-trivial and challenging.

The most straightforward algorithm is as follows. We sort the intervals $\in X$ in descending order of weight offline. Given a top-k weighted stabbing query, we run a sequential scan of X until we access k stabbed intervals. Due to the sort order, (i) this set of the k intervals is guaranteed to be the exact top-k result, and (ii) this algorithm can stop the scan before accessing n intervals. However, in the worst case, this algorithm needs to access all intervals, so it incurs $O(n \log k)$ time. (The factor of $O(\log k)$ is required to update the intermediate top-k result.) Another approach is to employ a state-of-the-art algorithm [9]. This algorithm uses an interval tree [8] to find all stabbed intervals, and the top-k intervals are found from them. Because the interval tree structure guarantees that a (non top-k weighted) stabbing query can run in $O(\log n + m)$ time, where m is the number of stabbed intervals, this algorithm can run in $O(\log n + m \log k)$ for our problem. At first glance, this algorithm seems sufficiently fast, but it is important to notice that m can be as large as n (e.g., all intervals are stabbed by a query). Therefore, this algorithm results in the same worst time as the sequential scan.

We hence have a question: For our problem, does there exist an exact algorithm with less than $O(n)$ query time (and with $\tilde{O}(n)$ space, where $\tilde{O}(\cdot)$ hides any polylog factors)? We provide a positive answer and make the following contributions:

- *An $O(\sqrt{n} \log n + k)$ time algorithm* (Sect. 3). We first propose an algorithm that exploits weight-based sorting and the interval tree structure. This technique provides a performance guarantee dominating that of the state-of-the-art algorithm [9], because our algorithm runs faster than the state-of-the-art with the same space requirement. As $\sqrt{n} \log n < n$, we have $O(\sqrt{n} \log n + k) < O(n)$.
- *An $O(\log n + k)$ time algorithm* (Sect. 4). The second algorithm improves the search efficiency by exploiting the segment tree structure [7]. A segment tree yields the same performance for simple stabbing queries, i.e., its time complexity is $O(\log n + m)$, so simply applying this structure still incurs $O(n \log k)$ time in the worst case. Nevertheless, we show that a simple modification of this structure provides an $O(k \log n)$ time algorithm for our problem. We furthermore extend the segment tree to reduce the time complexity from $O(k \log n)$ to $O(\log n + k)$.
- *Experiments on real datasets.* We conduct experiments on two real large datasets. Due to space limitation, their results appear in [3].

2 Preliminary

Problem Definition. We use X to denote a set of n intervals. Each interval $x \in X$ is a pair of its left and right endpoints, i.e., $x = [x.l, x.r]$, where $x.l \leq x.r$.

[1] Some applications may prefer smaller weights, and our algorithms can deal with this case.

In addition, each interval $x \in X$ has an application-dependent static weight $w(x)$. Given a query value s, we say that x is stabbed by s iff $x.l \leq s \leq x.r$. For ease of presentation, we first define the stabbing query:

Definition 1 (Stabbing query). *Given a stabbing query s (which is a value) and X, this query retrieves a subset X_s of X such that $X_s = \{x \mid x \in X, x.l \leq s \leq x.r\}$.*

This paper considers a variant of stabbing queries and addresses the problem defined below.

Definition 2 (Top-k weighted stabbing query). *Given a top-k weighted stabbing query $q = (s, k)$, where s and k respectively are a query value and a result size, and X, this query retrieves k intervals with the largest weights among X_s. (If $|X_s| < k$, all intervals in X_s are returned.) Ties are broken arbitrarily.*

The state-of-the-art algorithm [9] requires $O(\log n + m \log k)$ time, where $m = |X_s|$. Theoretically, m can be as large as n, so it requires $O(n \log k)$ time in the worst case.

Interval Tree and Segment Tree. We introduce the interval tree structure [8], a building block of our algorithm presented in Sect. 3. Due to space limitation, we introduce only its theoretical result.

Lemma 1. *An interval tree can be built in $O(n \log n)$ time, consumes $O(n)$ space, and processes a stabbing query in $O(\log n + m)$ time, where m is the number of stabbed intervals.*

We next introduce the segment tree structure [7], because we use it as a building block of our algorithm presented in Sect. 4. Again, we introduce only its theoretical result.

Lemma 2. *A segment tree can be built in $O(n \log n)$ time, consumes $O(n \log n)$ space, and processes a stabbing query in $O(\log n + m)$ time, where m is the number of stabbed intervals.*

3 Algorithm Based on Interval Forest

This section proves the following theorem. (Missing proofs appear in the full version of this paper [3].)

Theorem 1. *For our problem, there exists an exact algorithm that needs $O(n \log n)$ pre-processing time, $O(n)$ space, and $O(\sqrt{n} \log n + k)$ query time.*

Main Idea. The main idea of this algorithm is to combine weight-based sorting and the interval tree structure. Assume that the intervals in X are sorted in descending order of weight. Now assume that X is partitioned into two disjoint subsets X_1 and X_2, and note that $w(x) \geq w(x')$ for all $x \in X_1$ and $x' \in X_2$. Next consider that two interval trees $\mathcal{I}_1$ and $\mathcal{I}_2$ are built, i.e., $\mathcal{I}_1$ ($\mathcal{I}_2$) is built on X_1 (X_2). Given a top-k weighted stabbing query, we first use $\mathcal{I}_1$. If $\mathcal{I}_1$ returns k stabbed intervals, we do not need to use $\mathcal{I}_2$, since the weights of the intervals in $\mathcal{I}_2$ are less than those of the intervals in $\mathcal{I}_1$. Based on this observation, we reduce the $O(n \log k)$ time of [9] to $O(\sqrt{n} \log n + k)$.

3.1 Data Structure and Construction

We sort the intervals $\in X$ as above. Then, we partition X into p equal-sized disjoint subsets, i.e., $X = X_1 \cup X_2 \cup \cdots \cup X_p$ and $X_i \cap X_j = \varnothing$ $(i \neq j)$. In addition, $w(x) \geq w(x')$ for all $x \in X_i$, $x' \in X_{i+1}$ $(i \in [1, p-1])$. We later show how to specify p, which is an important factor for achieving a solid performance guarantee. Then, we build an interval tree for each subset of X, so we have p interval trees. Note that (i) this structure is general for arbitrary top-k weighted stabbing queries, meaning that this pre-processing is done only once, and (ii) Lemma 1 directly derives the following.

Corollary 1. *We can build p interval trees in $O(n \log n)$ time, and they require $O(n)$ space in total.*

3.2 Query Processing Algorithm

Our algorithm proposed in this section is denoted by IF because this algorithm employs multiple interval trees, i.e., Interval Forest. Given a top-k weighted stabbing query q, IF first uses the interval tree $\mathcal{I}_1$ on X_1 and runs q on $\mathcal{I}_1$. IF uses the stabbing query processing algorithm on the interval tree structure to find stabbed intervals. Whenever IF accesses a stabbed interval, it updates the top-k result. If the number of stabbed intervals is equal to or more than k, it is guaranteed that we can obtain the exact top-k result from $\mathcal{I}_1$, so IF returns the result. Otherwise, IF runs q on $\mathcal{I}_2$, and IF repeats this iteration until we have k stabbed intervals or all interval trees are used.

Analysis. We set $p = O(\sqrt{n})$, so we have $O(\sqrt{n})$ interval trees and $|X_i| = O(\sqrt{n})$ for each $i \in [1, p]$. Then, we have:

Lemma 3. *IF runs in $O(\sqrt{n} \log n + k)$ time.*

Proof of Theorem 1. From Corollary 1 and Lemma 3. □

4 Algorithm Based on a Variant of Segment Tree

We next consider accelerating the search efficiency further (by sacrificing pre-processing time and the space complexity a bit) and prove that

Theorem 2. *For our problem, there exists an exact algorithm that requires* $O(n \log n \log \log n)$ *pre-processing time,* $O(n \log^2 n)$ *space, and* $O(\log n + k)$ *query time.*

Main Idea. This algorithm is designed based on the segment tree structure. One may come up with the idea of sorting the intervals maintained in each node of a segment tree based on weight. This idea enables access to at most k intervals for each traversed node. As the height of the segment tree is $O(\log n)$, this idea derives an $O(k \log n)$ time algorithm. Although this algorithm can theoretically be faster than IF, its running time can be sensitive to k. We therefore do not employ this approach.

Instead, we focus on the following property: the stabbing query algorithm on the segment tree structure exploits the fact that the intervals maintained in the traversed nodes are guaranteed to be stabbed by a given query. Then, by storing all intervals existing in the path from the root to each node in a sorted array, we do not need to enumerate k intervals for each traversed node[2]. This new idea and the path-based auxiliary structure are specific to our problem, since simple stabbing queries enumerate all stabbed intervals and do not consider weights.

4.1 Variant of Segment Tree and Its Construction

We first build a segment tree on X. Then, for each node u of the segment tree, we consider the path from u_{root} to u. We collect all "distinct" intervals maintained in the nodes on the path (since duplicate intervals may exist in the path), and u stores this set of intervals in a weight-based sorted array.

After making this sorted array for each node u of the segment tree, we remove a set of intervals initially maintained in u because we do not use it anymore. Note that this structure is also general to arbitrary top-k weighted stabbing queries, so this pre-processing is done only once. We analyze this pre-processing time and the space complexity of this structure.

Lemma 4. *We can build the above variant of a segment tree in* $O(n \log n \log \log n)$ *time.*

Lemma 5. *The above variant of a segment tree needs* $O(n \log^2 n)$ *space.*

[2] This idea is not available for the interval tree structure. This is because the interval tree structure does not guarantee that all intervals maintained in a node are stabbed by a given query.

4.2 Query Processing Algorithm

Now we are ready to present our second algorithm for the top-k weighted stabbing queries. Thanks to our non-trivial extension of the segment tree structure, we can design a simple and fast algorithm. This algorithm is denoted by ST-PSA (Segment Tree with Path-based Sorted Arrays).

Let $\mathcal{S}$ be our variant of a segment tree on X. Given a top-k weighted stabbing query $q = (s, k)$, ST-PSA first runs a simple stabbing query $q.s$ on $\mathcal{S}$ and obtains the node traversed last during the stabbing. Let this node be u, and ST-PSA uses the sorted array of u. Specifically, ST-PSA returns the first k intervals in the array as the top-k result.

Correctness. The stabbing query algorithm on the segment tree structure guarantees that all intervals maintained in the traversed nodes are stabbed by a given query. In addition, the sorted array of u stores all intervals (initially) maintained in the path from u_{root} to u. From these facts, the correctness of ST-PSA is clear.

Time Complexity. We present the main result of this section below.

Lemma 6. *ST-PSA runs in $O(\log n + k)$ time.*

Proof of Theorem 2. From Lemmas 4–6. □

5 Conclusion

This paper addressed the problem of processing top-k weighted stabbing queries. A state-of-the-art algorithm for this problem incurs the same time complexity as that of a sequential scan. Motivated by this inefficiency issue, this paper proposed two sublinear time algorithms.

Acknowledgements. This work was partially supported by AIP Acceleration Research JPMJCR23U2, JST, and JSPS KAKENHI Grant Number 24K14961.

References

1. Agarwal, P.K., Arge, L., Yi, K.: An optimal dynamic interval stabbing-max data structure? In: SODA, pp. 803–812 (2005)
2. Amagata, D.: Independent range sampling on interval data. In: ICDE (2024)
3. Amagata, D., Yamada, J., Ji, Y., Hara, T.: Efficient algorithms for top-k stabbing queries on weighted interval data (full version). arXiv:2405.05601 (2024)
4. Behrend, A., et al.: Period index: a learned 2D hash index for range and duration queries. In: SSTD, pp. 100–109 (2019)
5. Christodoulou, G., Bouros, P., Mamoulis, N.: HINT: a hierarchical index for intervals in main memory. In: SIGMOD, pp. 1257–1270 (2022)

6. Christodoulou, G., Bouros, P., Mamoulis, N.: HINT: a hierarchical interval index for allen relationships. VLDB J. **33**, 73–100 (2023). https://doi.org/10.1007/s00778-023-00798-w
7. De Berg, M.: Computational Geometry: Algorithms and Applications (2000). https://doi.org/10.1007/978-3-540-77974-2
8. Edelsbrunner, H.: Dynamic Rectangle Intersection Searching (1980)
9. Xu, J., Lu, H.: Efficiently answer top-k queries on typed intervals. Inf. Syst. **71**, 164–181 (2017)
10. Zhang, Z., Gan, J., Bao, Z., Kazemi, S.M.H., Chen, G., Zhu, F.: Approximate range thresholding. In: SIGMOD, pp. 1108–1121 (2022)