Interval Stabbing Problems in Small Integer Ranges

Jens M. Schmidt*

Freie Universität Berlin, Germany

Enhanced version of August 2, 2010

Abstract

Given a set I of n intervals, a *stabbing query* consists of a point q and asks for all intervals in I that contain q. The *Interval Stabbing Problem* is to find a data structure that can handle stabbing queries efficiently. We propose a new, simple and optimal approach for different kinds of interval stabbing problems in a static setting where the query points and interval ends are in $\{1, \ldots, O(n)\}$.

keywords: interval stabbing, interval intersection, rank space, static, discrete, point enclosure

1 Introduction

Interval stabbing, also known as the one-dimensional *point enclosure problem* is one of the most fundamental problems in computational geometry and has been studied for decades. Let l_a be the left endpoint and r_a be the right endpoint of an interval a. We address the following static setting:

Let I be a given set of n intervals with l_a, $r_a \in Q := \{1, ..., O(n)\}$ for every $a \in I$. An interval $a \in I$ is *stabbed* by a point $q \in Q$ if $q \in a$. We want to construct simple and lightweight data structures that answer the following queries on I efficiently:

1. *Interval Stabbing Problem*: Given a query point $q \in Q$, report all intervals in I that are stabbed by q.

2. *Interval Intersection Problem*: Given a query interval $[l_q, r_q]$ with $l_q, r_q \in Q$, report all intervals $i \in I$ with $[l_i, r_i] \cap [l_q, r_q] \neq \emptyset$.

3. *Interval Cover Problem*: Given an interval $q \in I$, report all intervals in I that contain the interval q.

*This research was supported by the Deutsche Forschungsgemeinschaft within the research training group “Methods for Discrete Structures” (GRK 1408).

4. *Multiple Query Problems*: These problems extend each of the problems 1-3 by allowing multiple queries $q_1 < \ldots < q_t$, $\forall i : q_i \in Q$, at the same time. The query points have to be given as a sorted list while the output consists of the intervals that are stabbed by at least one q_i (without double occurences).

We demand in addition that the intervals in each output are reported in lexicographical order. In general, queries do not admit a worst-case running time better than $O(n)$, since the output itself can be that large. But for many inputs the output will be much smaller. Therefore, it is reasonable to consider the *output-sensitive complexity* for queries, where the running time is given with respect to the input size and the output size k. We assume the uniform cost model, thus k is the number of intervals in the output. Clearly, every data structure needs to store all intervals and, thus, needs at least $\Omega(n)$ space and preprocessing time to be built. The query time is at least $\Omega(1+k)$ (or $\Omega(t+k)$ for problems of type 4), the 1 (or t) coming from queries in Q that are not covered by any interval.

We will only focus on solutions that reach that bounds, i. e., that solve problems 1-4 in asymptotic optimal space and time. Therefore common interval data structures like *interval trees* [6, 8], *segment trees* [4] and *priority trees* [9] cease to apply, as each of them needs a preprocessing time of $\Omega(n \log n)$ and query times of at least $\Omega(\log n + k)$. Alstrup, Brodal and Rauhe [1] describe a data structure, based on results in [7], that can be used for solving the problems optimally. The idea is to interpret every interval $a \in I$ as a point (l_a, r_a) in the integer grid $n \times n$ and then model the given problem by *three-sided range queries* in this grid, i. e., by rectangular range queries with one side going to ∞ or $-\infty$. Each three-sided range query can be performed in time $O(1+k)$ by computing iteratively *nearest common ancestors* in a cartesian tree as shown by Gabow, Bentley and Tarjan [7]. However, this procedure seems far too involved for the type of problems we look at and comes with a significant implementation overhead.

The essence of this paper is a direct, new approach that solves all problems optimally and does not rely on computing nearest common ancestors, thus has considerably less overhead. Problems 1 and 2 can as well be solved by the *filtering search* data structure due to Chazelle [5] in the same asymptotic time and space requirements. However, filtering search does not solve Problems 3 and 4 and experiments show that our data structure performs faster than filtering search in practice. That can be explained with the lower number of comparisons needed for one query in the theoretical worst case: Our data structure needs $3k$ point-to-q comparisons instead of $8k$ comparisons for Chazelle's data structure.

We assume all given intervals a to be closed, but, if necessary, open and half open intervals may be easily modeled by increasing l_a and/or decreasing r_a by one in advance. Let l_a and r_a be the *endpoints* (or shorter *ends*) of an interval $a \in I$ and let l_a be the *left endpoint* and r_a be the *right endpoint*.

If the query range Q is not $\{1, ..., O(n)\}$ there are techniques that reduce problems to work within a small integer range [1, 7]. E. g., any universe can be reduced to the integer range $\{2, ..., 4n\}$ by first sorting the $\leq 2n$ interval ends and then assigning to each one two times its rank. This leaves a gap between every pair of consecutive interval ends. Then a binary search transforms any stabbing query $q \in Q$ to a query in $\{2, ..., 4n\}$, reflecting its relative position in Q, either at an interval end or a gap. This goes along with a blow-up of the preprocessing time to $O(n \log n)$ and query time to $O(\log(n) + k)$ for problems 1-3 and to $O(\min(t \log(n), n) + k)$ for problems of type 4. If the model of computation is the unit-cost word RAM and each query point fits in a constant number of words, much faster algorithms for sorting and predecessor searching of query points can be applied (Andersson et al. [2], Beame and Fitch [3]), although these results are not needed for the restricted universe we consider here.

2 The Data Structure

We identify intervals with their left and right endpoints and sort all intervals according to the *lexicographic order* $< \subseteq N \times N$, i. e., for two intervals a and b holds $a < b$ if $l_a < l_b$ or $(l_a = l_b \wedge r_a \leq r_b)$. The computation time of this lexicographic list is $O(n)$ by using (stable) bucket sort for the right endpoints followed by a bucket sort for the left endpoints, since all interval ends are by definition in Q.

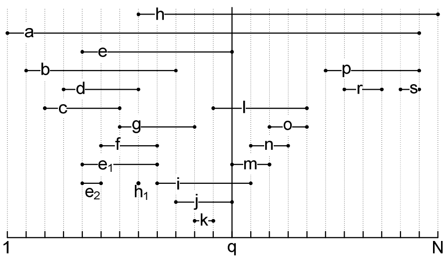


Figure 1: The intervals e_1, e_2 and h_1 are removed from I in advance, because *Smaller*$(e) = \{e_1, e_2\}$ and *Smaller*$(h) = \{h_1\}$. Only intervals a and b cover d and *Parent*$(d) = b$. Moreover, c overlaps d, e, f, g and e_1 but d does not overlap c.

Intervals that share right endpoints will integrate well in our data structure, thus the frequently used input transformation to intervals with completely distinct ends is not necessary. To get rid of intervals sharing their left endpoint l (for every l), we apply the following preprocessing: All the intervals with left endpoint l, except one such longest interval a, are stored in a list called *Smaller*(a) (see Figure 1). These lists are sorted by length in descending order, get a link to a, and every element in them is removed from I (i. e., from now on I does not contain intervals in *Smaller*(a) and n is redefined to $|I|$ afterwards). This establishes $<$ to be a *strict total order* on I relying only on left endpoints. Later, a simple trick will deal with the omitted intervals *Smaller*(a).

Two intervals $a, b \in I$ *intersect* if $a \cap b \neq \emptyset$. Otherwise, they are called *disjoint*. We say that interval a *overlaps* interval b if $l_a < l_b \leq r_a < r_b$. Moreover, let a be *covered* by b (and b *cover* a) if $a \subseteq b$. Let the *rightmost* interval in a non-empty subset of I be the interval with the maximal left endpoint. Note that this is well-defined as the left endpoint is unique in I. Then *Parent*(a) is defined as the rightmost among all intervals that cover a (see Figure 1). If a is not covered by any interval, *Parent*$(a) := \emptyset$.

Proposition 1. *For two intervals $a, b \in I$ with $a < b$ exactly one of the following statements holds:*

- *a and b are disjoint*
- *a is covering b*

- *a overlaps b.*

We attach each interval a to *Parent*(a), yielding a forest F with intervals as nodes and the *Parent*-function as edges. Let $root_i$ denote the root of a maximal tree T_i in F. We construct a spanning tree $S = (V, E)$ by augmenting the forest with a special dummy node *root* (representing the interval Q) and attaching the roots of all trees T_i to it (see Figure 2).

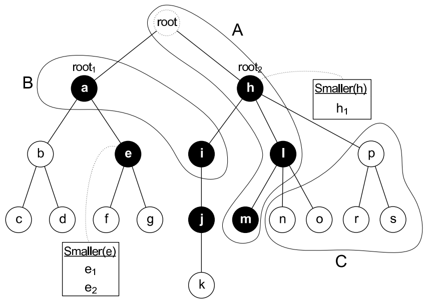


Figure 2: The spanning tree S of the example in Figure 1. Black nodes indicate intervals stabbed by q.

Let S be ordered by sorting the children of each node according to their left endpoints. The children of a node $v \in V$ are stored in a doubly linked list, denoted by *Children*(v). Every entry in *Children*(v) is a *sibling* of each other entry. We call the sibling immediately to the left (right) of an entry the *left sibling* (*right sibling*). In a tree, a node w is an *ancestor* of a node v, if w is contained in the path from v to the root (including the node v).

3 The Interval Stabbing Problem

We show how to solve Problem 1 using the spanning tree S and extend this result later to problems 2-4. First, imagine that all pairs of intervals in I would either be disjoint or cover each other. In this restricted case it suffices to precompute the rightmost stabbed interval *Start*(q) for every $q \in Q$, if it exists. If a query q arises, let T_s be the tree in F that contains *Start*(q) and P be the path from *Start*(q) to $root_s$ in T_s. Then we can get all k stabbed intervals by traversing P in time $O(1 + k)$, since *Start*(q) must be the smallest stabbed interval and all other stabbed intervals have to be ancestors of it in T_s.

However, in general intervals may overlap and stabbed ones can even be contained in different trees of F. We partition $V(S)$ into four classes subject to P (see Figure 2). A node $v \in V(S)$ is in class

- A, if v is in P or the dummy node
- B, if v has a sibling w in P with $l_w > l_v$
- C, if $l_v > q$
- D, otherwise

Lemma 2. *For every $v \in V(S)$ the stabbed children of v are adjacent in* Children(v).

Proof. We assume to the contrary that there is at least one child $b \in I$ that is not stabbed between two stabbed children a and c. Since siblings cannot cover each other and a and c cannot be disjoint a must overlap c. Then $a \cap c$ contains the query point and b is stabbed as well, since $l_b < l_c$ and $r_b > r_a$. ☐

Given a query point q, we first show how to obtain all stabbed intervals in the sets A, B and C efficiently with a traversal starting at $\mathit{Start}(q)$. If $\mathit{Start}(q)$ is not stabbed, no interval can be stabbed and the query time is $O(1)$. Otherwise, all intervals in A must contain q and we traverse them. The stabbed intervals in B can then be easily computed with Lemma 2 by iteratively traversing to the left sibling for each node in P until the list ends or a node was not stabbed. No interval in C can be stabbed because their left endpoints are greater than q by definition, so only class D remains.

Lemma 3. *Every stabbed node $v \in D$ has a stabbed ancestor in B.*

Proof. With v all ancestors of v are stabbed and at least one of them is contained in A, since the dummy node is in A. Let w be the ancestor that is not in A but has a parent z in A. If $z \neq \mathit{Start}(q)$ then w is a stabbed sibling left of a node in P and therefore in B and the claim follows. Otherwise, $z = \mathit{Start}(q)$ contradicts $v \in D$, since all intervals in the subtree of z have a greater left endpoint than z has. ☐

Lemma 4. *If $v \in D$ has a right sibling w and is stabbed, w is stabbed as well.*

Proof. According to Lemma 3, there is a stabbed ancestor of v and w in B. Then the right sibling z of this ancestor exists, is either in A or B and is stabbed. By construction of the spanning tree $l_v < l_w < l_z$ must hold and the query point q is in $v \cap z$. Since $v \cap z \subseteq w$, the point q has to stab w as well. ☐

Lemmas 3 and 4 lead immediately to a recursive characterization of all stabbed nodes in D. Let $U(v)$ for a node $v \in D$ be the sequence of nodes from v to the first node in B where each successor is the right sibling, if it exists, and otherwise the parent.

Corollary 5. The node $v \in D$ is stabbed if and only if all nodes of $U(v)$ are stabbed.

Corollary 5 allows us to compute all stabbed nodes in D by traversing paths back from stabbed nodes in B.

Definition 6. The *rightmost path* $R(v)$ of a node $v \in V(S)$ is empty if v has no left sibling or its left sibling w is not stabbed. Otherwise, $R(v)$ is the path from w to the rightmost stabbed node in the subtree of w in S.

Note that $R(v)$ contains only stabbed intervals and can be constructed by iteratively taking the last child, starting with w. We are now in a position to compute all stabbed nodes by traversing P from the bottom up and recursively computing and traversing $R(v)$ from the bottom-up for each visited node v (see Algorithm 1). All stabbed nodes in A and B are found, since the computation of rightmost paths considers left children and continues with them at some point, if they are stabbed. The same holds for stabbed nodes in D, since Corollary 5 ensures that all stabbed nodes in C are reachable by a sequence of rightmost paths that start with a stabbed node in B.

We can find all k stabbed intervals in $O(1 + k)$ time, because checking an interval to be stabbed by q, computing $\mathit{Start}(q)$ and traversing to the parent, left sibling or last child can be done in constant time.

Algorithm 1 Traverse $(v \in V(S), \text{stack } O, q \in Q)$

Push v to stack O ▷ for output purposes
while next interval w in *Smaller*(v) exists and $q \in w$ **do**
    Push w to stack O
Compute the rightmost path $R(v)$
for all nodes w in $R(v)$ (from the bottom up) **do**
    Traverse(w, O, q)

Algorithm 2 Stabbing query $(q \in Q)$

Stack $O = \emptyset$ ▷ O for output purposes
if *Start*$(q) = \emptyset$ **then** STOP ▷ q stabs no interval
Compute the path P from *Start*(q) to $root_s$
for all nodes v in $V(P)$ (from the bottom up) **do**
    Traverse(v, O, q)
while $O \neq \emptyset$ **do** ▷ reverse list of stabbed intervals
    Append *pop*(O) to output

It only remains to show how to deal with the intervals in the *Smaller*-lists and ensure that the output is sorted in lexicographic order. Each time we reach a node v with *Smaller*$(v) \neq \emptyset$, we traverse that list until the end or the first non-stabbed node was found. This way we do not visit intervals that are not stabbed and, thus, preserve the running time of $O(1+k)$ for each query. We use the following lemma to verify that the output is sorted in lexicographic order.

Lemma 7. *A preorder traversal on root returns all intervals of S sorted by their left endpoints.*

Proof. All children of a node $v \in V(S)$ are sorted and have left endpoints strictly greater than l_v for $v \neq root$. Let w be the right sibling of v. Then, due to the definition of *Parent*, every interval in the subtree on v has a left endpoint of strictly less than l_w. Recursively collecting the actual node and traversing the children from left to right returns the intervals sorted by their left endpoints. □

The traversal of S starts with the stabbed interval *Start*(q) that has the maximal left endpoint and visits subsequent intervals containing q in a postorder traversal that prefers right children to left children. As this postorder reverses the preorder traversal and the output of the preorder traversal is sorted in inverse lexicographic order with Lemma 7, we need to reverse the order of intervals found. This is done by using a stack (see Algorithm 2).

All preprocessing steps, i. e., computing the *Parent* and *Start* pointers can be done with one sweep line procedure in time $O(n)$ by maintaining a list of stabbed intervals for each $q \in Q$ (see the pseudocode description in Algorithm 3). For each stabbed interval v of a query, we check at most three subsequent intervals on containing q, the left sibling of v, the last child of v and the successor in *Smaller*(v). However, we need only to compare the right endpoints of those intervals with q, since Lemma 7 ensures that $l_v \leq q$ holds.

Algorithm 3 Preprocessing

```
List L = ∅; create dummy node root
∀q ∈ Q : compute lists Smaller(q); update I
∀a ∈ I : create a pointer to the interval Parent(a)
for all a ∈ I in lexicographic order do                ▷ build event structure
    Append a to Event(r_a)                             ▷ event list of intervals on r_a
    Append a to Event(l_a)
for q = 1 to N do                                      ▷ sweep line
    If L ≠ ∅, store the last element in L as Start(q), else Start(q) := NULL
    for all intervals a ∈ Event(q) in reverse order do
        if l_a = q then
            Start(q) = a
            Append a to L and save a link to its position in L
        else                                           ▷ r_a = q
            if a has a predecessor b in L then
                Store b as Parent(a) and append a to Children(b)
            else
                Store root as Parent(a) and append a to Children(root)
            Remove a from L
```

Theorem 8. *All k intervals stabbed by a query point q can be found sorted in lexicographic order in query time $O(1+k)$ and with at most $3k$ comparisons with q ($2k$ comparisons if all left endpoints in I are pairwise distinct). The preprocessing time and space requirement is $O(n)$.*

4 Variants of the Problem

We discuss the problems 2-4. The *Interval Intersection Problem* differs from the *Interval Stabbing Problem* only in having a query interval $[l_q, r_q]$ instead of a query point. Let an interval be stabbed if its intersection with $[l_q, r_q]$ is non-empty. Then Lemmas 2, 3, 4 and Corollary 5 remain valid and we can still use the data structure of the *Interval Stabbing Problem* to get all stabbed intervals. The traversal starts at the rightmost interval t intersecting $[l_q, r_q]$, if exists, and recurses to its ancestors and rightmost paths as described before, finally stopping at intervals that are not stabbed. These lie with Lemma 7 completely left of l_q and are not part of the output. Testing a visited interval a on being stabbed can be done with one comparison by checking $r_a \geq l_q$, leading to a $O(1+k)$ query time in total. It remains to show how t can be precomputed. During the preprocessing, we store additional values $\mathit{Start}_2(p)$ for every $p \in Q$, which point to the rightmost interval a with $l_a \leq p$ (this does not increase the asymptotic running time). When querying $[l_q, r_q]$, $l_q \in t$ implies that $t = \mathit{Start}(l_q)$. Otherwise, $t = \mathit{Start}_2(r_q)$ must hold and t can be obtained by computing the interval $\max(\mathit{Start}(l_q), \mathit{Start}_2(r_q))$ in time $O(1)$, if exists, and checking whether the right endpoint of this interval is at least l_q.

For the *Interval Cover Problem* an interval $q \in I$ is given. We set $\mathit{Start}(q) = q$, because there is no interval with a higher left endpoint covering q. Since ancestors cover q if one of their children does, we can build the path P and partition $V(S)$ subject to P as in the *Interval Stabbing Problem*. When we

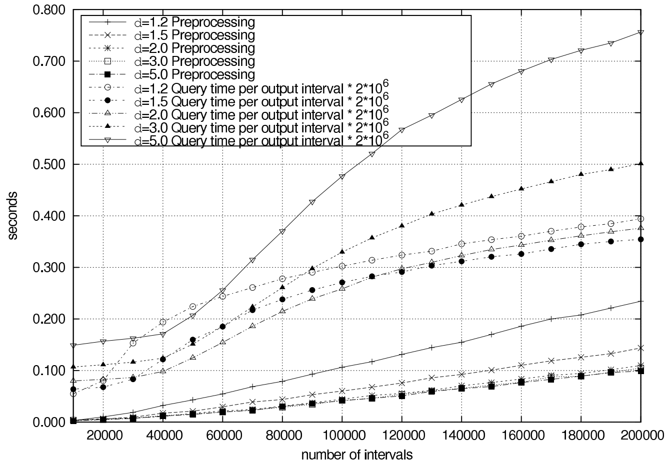


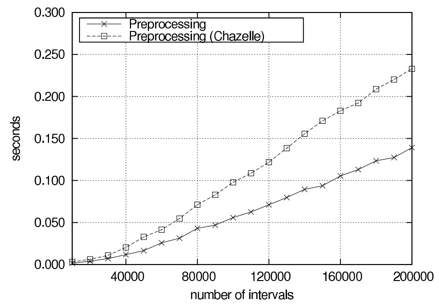
Figure 3: Different values for the window size δ in Chazelle's data structure (short random intervals).

replace the property *stabbed* with *covering* q on intervals, the Lemmas 2, 3, 4 and Corollary 5 still hold. Every visited interval a can be tested on covering q by checking $r_a \geq r_q$, which gives a query time of $O(1+k)$.

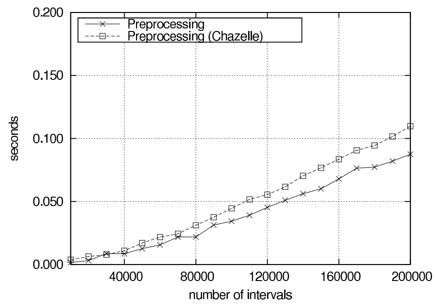
We show that the *Multiple Interval Stabbing Problem* in 4 allows for a query time of $O(t+k)$, the other problems in 4 can then be solved using the same technique. If every interval in the output would be stabbed by only one value q_i, $1 \leq i \leq t$, the problem could be solved in time $O(t+k)$ by applying the queries $q_1, \ldots, q_t$ subsequently. In general that is not the case and we have to ensure that the traversals of different query points do not both visit a node.

Assume that we start with the traversal of the rightmost query point q_t and compute recursively rightmost paths. Then with Lemma 7 the sequence of left endpoints of visited intervals is strictly monotone decreasing. For every visited interval a we check in advance if $l_a \leq q_{t-1}$ holds and if so, replace the current query point q_t with the minimal query point $q_j \geq l_a$, $j < t$. If now a or any subsequent interval is stabbed by q_t, it will also be stabbed by q_j and we can perform all comparisons with q_j instead of q_t. If a traversal of q_i, $i > 1$, ends without switching to q_{i-1} we invoke the traversal on the next query point q_{i-1}. Since the list $q_1, \ldots, q_t$ is ordered, the additional expense to update the query point is bounded by t constant time comparisons, which gives a total query time of $O(t+k)$.

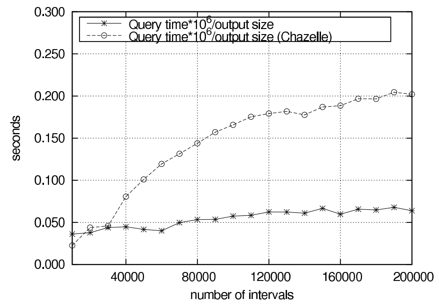
Corollary 9. The k intervals in the output of problems 2 and 3 can be found sorted in lexicographic order in query time $O(1+k)$ and with at most $3k$ comparisons with q ($2k$ comparisons if all left endpoints in I are pairwise distinct). For problems of type 4 a query can be done in time $O(t+k)$ with at most $4k$ comparisons with values in $\{q_1, \ldots, q_t\}$. For all problems the preprocessing time and space requirement is $O(n)$.



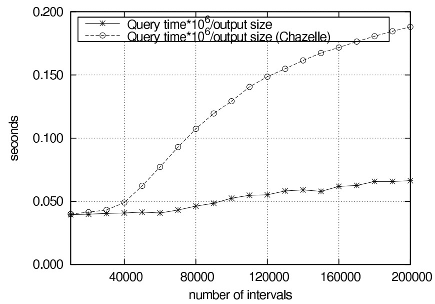
(a) Preprocessing (random intervals)



(b) Preprocessing (short random intervals)



(c) Query times (random intervals)



(d) Query times (short random intervals)

Figure 4: Comparison of preprocessing and query times.

5 Experimental Analysis

We implemented Chazelle's data structure [5] with various window sizes ($\delta = 1.2,\ 1.5,\ 2,\ 3,\ 5$) for the *Interval Stabbing Problem* and compared the running times to our approach on Problem 1. However, $\delta = 2$ gave the best results in both preprocessing and query times and we will focus on that parameter, since $\delta < 2$ did not lead to observable better query times but to a considerably worse preprocessing time instead (see Figure 3). Both data structures use identical representations for intervals, lists and stacks and work under the same conditions as much as possible. All tests are performed on a 1.86 GHz CPU and 2GB RAM using the MS compiler 9.0 with optimization level O2. The source code is available online: http://page.mi.fu-berlin.de/jeschmid.

The input consists of various n from 10000 to 1000000, $Q := \{1, \ldots, 5n\}$ (other constants than 5 led to similar results) and either *random intervals* with uniformly distributed interval ends in Q or *short random intervals*. *Short random intervals* have an exponentially distributed length with expected value 1000, while their left endpoints and all query points are uniformly distributed on Q. The exponential dis-

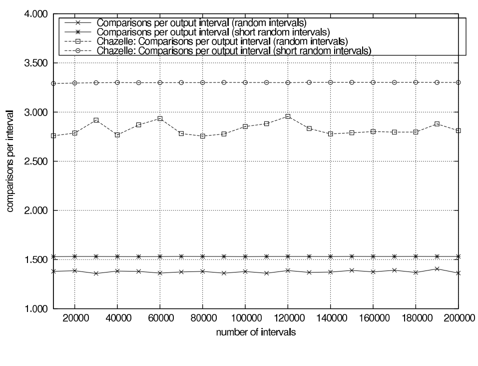


Figure 5: Point-to-q comparisons

tribution is generated with inverse transform sampling on a uniform distribution. Both query and preprocessing times are averaged over 20 instances for each n with up to 10000 queries per instance.

The preprocessing times of both data structures in practice reflect the theoretical linear bound of $\Theta(n)$, except for small n (see Figures 4(a) and 4(b)). In both figures, our approach performs faster, although on short random intervals the advantage is marginal. Since query times are primarily dependent on the output length, we measure the average computation time needed for one interval in the output. Theoretically, each query time should be constant, although the memory hierarchy can increase the time in practice when n grows. For large n, the query times of our data structure are significantly faster than Chazelle's for both input types (see Figures 4(c) and 4(d)).

Figure 5 shows how many point comparisons with q are made on average for each interval in the output. Both data structures need about half of the comparisons of the theoretical worst case (3 point-to-q comparisons for our data structure, 8 point-to-q comparisons for Chazelle's data structure).

References

[1] S. Alstrup, G. S. Brodal, and T. Rauhe. New data structures for orthogonal range searching. In *Proceedings of the 41st Annual Symposium on Foundations of Computer Science (FOCS'00)*, pages 198–207, 2000.

[2] A. Andersson, T. Hagerup, S. Nilsson, and R. Raman. Sorting in linear time? In *Proceedings of the 27th Annual ACM Symposium on Theory of Computing (STOC'95)*, pages 427–436, 1995.

[3] P. Beame and F. E. Fich. Optimal bounds for the predecessor problem. In *Proceedings of the 31st Annual ACM Symposium on Theory of Computing (STOC'99)*, pages 295–304, New York, NY, USA, 1999. ACM.

[4] J. L. Bentley. Solutions to Klee's rectangle problems. Tech. report, Carnegie-Mellon Univ., Pittsburgh, PA, 1977.

[5] B. Chazelle. Filtering search: A new approach to query answering. *SIAM J. Comput.*, 15(3):703–724, 1986.

[6] H. Edelsbrunner. Dynamic data structures for orthogonal intersection queries. Tech. Report F59, Inst. Informationsverarb., Tech. Univ. Graz, 1980.

[7] H. N. Gabow, J. L. Bentley, and R. E. Tarjan. Scaling and related techniques for geometry problems. In *Proceedings of the 16th Annual ACM Symposium on Theory of Computing (STOC'84)*, pages 135–143, 1984.

[8] E. M. McCreight. Efficient algorithms for enumerating intersecting intervals and rectangles. Tech. Report CSL-80-9, Xerox Palo Alto Res. Center, CA, 1980.

[9] E. M. McCreight. Priority search trees. *SIAM J. Comput.*, 14(2):257–276, 1985.