# Advanced Data Structures

Assignment - Interval Tree

Manuel Haag

June 15th 2024

## 1 Introduction

In this project we want to evaluate Interval Trees on the Line Stabbing problem. In the later problem, we are given a set of intervals $I_1 = [a_1, b_1], I_2 = [a_2, b_2], \ldots, I_n = [a_n, b_n] \subseteq \mathbb{R}$ and are asked to report all intervals that intersect a query $s \in \mathbb{R}$. The Brute Force solution is to scan all intervals and check $s \in I_j$. The worst case run time of $O(n)$ is optimal in the case that we have to report all intervals. However, in this problem it is natural to consider output-sensitive algorithms, in which the run time is parameterized in the input size $n$ and the output size $k$. Interval Trees on the Line Stabbing problem achieve a query complexity of $O(\log n + k)$, being more efficient than the Brute Force method for $k \in o(n)$. They require $O(n \log n)$ time for construction and $O(n)$ additional space. We implemented both approaches and experimentally evaluate them on random instances with fixed interval size. In our experiments we want to address the question, whether for instances with many number of intersections the Brute Force method might be faster in practice than the Interval Tree.

Our work is structured as follows. In Section 2 we will briefly introduce the Interval Tree data structure. Then, we conduct a experimental evaluation of our methods in Section 3 and finish our work with a conclusion in Section 4.

## 2 Interval Trees

Interval Trees reduce the number of intervals we have to check, by storing the intervals in a balanced search tree as follows. First, we select a split point $m \in \mathbb{R}$ and divide the intervals into three sets. The intervals $S_l$ completely to the left of $m$, the intervals $S_r$ completely to the right of $m$ and $S_c$, the intervals intersecting with $m$. We create a node storing the set $S_c$ by two sorted lists, one containing the intervals sorted by the start point and one sorted by the end point. Then, we continue recursively with $S_l$ and $S_r$. Ideally, the split point $m$ divides the intervals such that $|S_l| = |S_l|$ to guarantee that the binary search tree is balanced.

One way to select a value of $m$, is to select the median of $a_1, b_1, a_2, b_2 \ldots, a_n, b_n$. This method may not yield a perfectly balanced tree, but still yields $O(\log n)$ tree depth. The median can be computed in $O(n)$ using the quickselect algorithm. We have to perform linear work on each level of the tree to compute the median and scan the intervals. Additionally, we have to sort the start and end points of intervals whenever we create a new node. In the worst case we have to sort all intervals on the first level. So the overall construction time is $O(n \log n)$. The number of nodes can not exceed $n$ so the additional space is in $O(n)$.

In our implementation, we stop the recursion as soon as the number of intervals is smaller than 100 and we switch to the Brute Force method. This way we can save some memory on the lower levels of the tree and need less levels. However, the impact on query and construction time in our experiments is insignificant.

Now we describe how to perform a Line Stab query given a value $s \in \mathbb{R}$. Let $m$ be the split value at the current node, $x_1 \leq x_2 \leq \cdots \leq x_l$ the sorted starting points in the first list and $y_1 \leq y_2 \leq \cdots \leq y_l$ the sorted end points in the second list. We consider three cases.

a) $s = m$
The node stores exactly the intersected intervals, so we report all intervals in the current node.

b) $s < m$
Each interval in the node contains $m$, so we only have to consider intervals with $x_j \leq s$. Since the start points are sorted, we scan the list from left to right and report all intervals until $x_j > s$. Then, we continue recursively in the left sub-tree.

c) $s > m$
This case is symmetric to b), we have to consider the intervals with $s \leq y_j$. We scan the second list in reverse order until $y_j < s$ and continue recursively in the right sub-tree.

Reporting a single intersection takes constant time and in the worst case we have to traverse the longest root to leaf path in the tree, which is $O(\log n)$. So the query complexity of the Interval Tree is $O(\log n + k)$. In our implementation, we assign each interval a unique id and in a query we append the ids of intersected intervals to a given list. This allows the user in practice to access meta-data associated with intersected intervals.

# 3 Experiments

In this section we present an experimental evaluation our methods. We implemented the Brute Force method and an Interval Tree using dynamically allocated nodes with pointers to children in C++20 and compiled with optimization flags -O3 -march=native. We ran our experiments on a laptop with Ubuntu 20.04.6 LTS, a AMD Ryzen 5 5600H CPU clocked at 3.3 GHz, 16 GB RAM and L1, L2, L3 cache of 192 KiB, 3 MiB and 16 MiB respectively.

As a benchmark set we use random uniformly generated intervals of fixed length 100 in the range $[0, 100/p]$ . The parameter $p$ controls the maximal range of the intervals. Notice that $p$ is the probability that an interval intersects a random query $s \in [0, 100/p]$ and $pn$ is the expected number of intersections. We do this to analyze the behavior of our algorithms for fixed number of intersections $k$.

In one run we construct the Interval Tree and then run $10^3$ random Line Stab queries. For the Brute Force approach there is no construction phase and we simply query the list of intervals directly with $10^2$ random queries. We use less queries than in the Interval Tree, since for large $n$ the Brute Force approach slows down the experiments significantly and since each query scans the hole array, the variance in a query is smaller than in the Interval Tree. We use 10 runs with different seeds and report average values in our plots.

Figure 1 shows the average tree depth with various values of $p$. For $p$ larger than 0.5, all the interval intersect at 100, so they are all stored at the root node. Decreasing $p$ increase the depth to 7 for $p = 10^{-2}$ and depth 15 for $p = 10^{-5}$. This is because a node in the higher levels contains on average $np$ intervals. With smaller values $p$, the depth increases, since less nodes are stored in one level.

Next we will look at construction times with varying number of intervals $n$ in Figure 2. For $n$ less than $2^{16}$ the construction takes less than 20 milliseconds, while for large values ($n = 2^{20}$) it takes between to 0.2 and 0.4 seconds. We observe for larger $n$ that the construction time increases with lower value of $p$. This is due to the higher tree depth we showed in Figure 1. More tree nodes have to be allocated, for each level the median algorithm is called and the remaining intervals are scanned, which increases the construction time.

Looking at the query time of Brute Force and Interval Tree in Figure 3 and Figure 4, we see that in the Brute Force approach a single query takes between 1 and 3.5 milliseconds for large

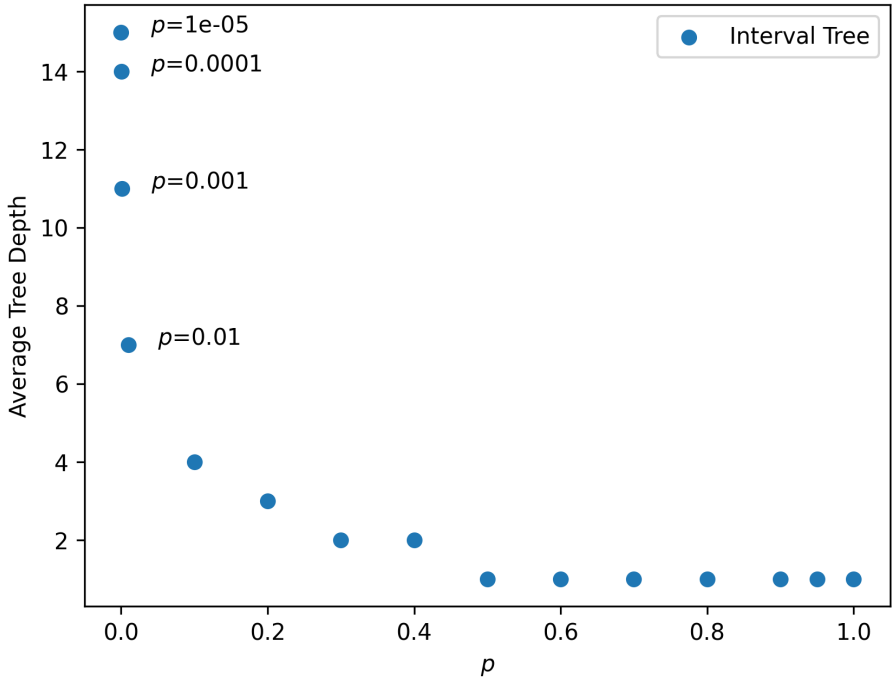


Figure 1: Average depth for Interval Tree by $p$ for $n = 2^{20}$ intervals.

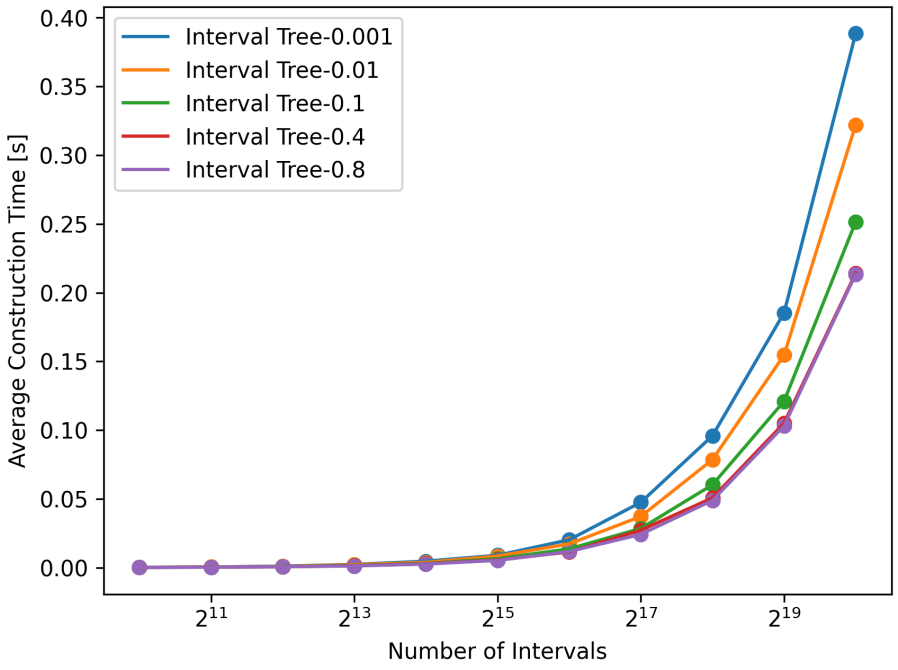


Figure 2: Average construction time Interval Tree.

instances ($n \geq 2^{19}$). For values very small or large values of $p$, the Brute Force method is faster than for medium values of $p$. This can be explained by the branch prediction of the compiler, which become easier for extremer values of $p$. In the case of the Interval Tree the query time significantly

decrease with $p$, since the output size is reduced. For $n$ equal to $2^{20}$ the query time varies between 0.01 milliseconds ($p = 0.0001, 0.001$), 0.08 miliseconds ($p = 0.1$) and 0.35 to 0.8 milliseconds for ($p = 0.4, 0.8$).

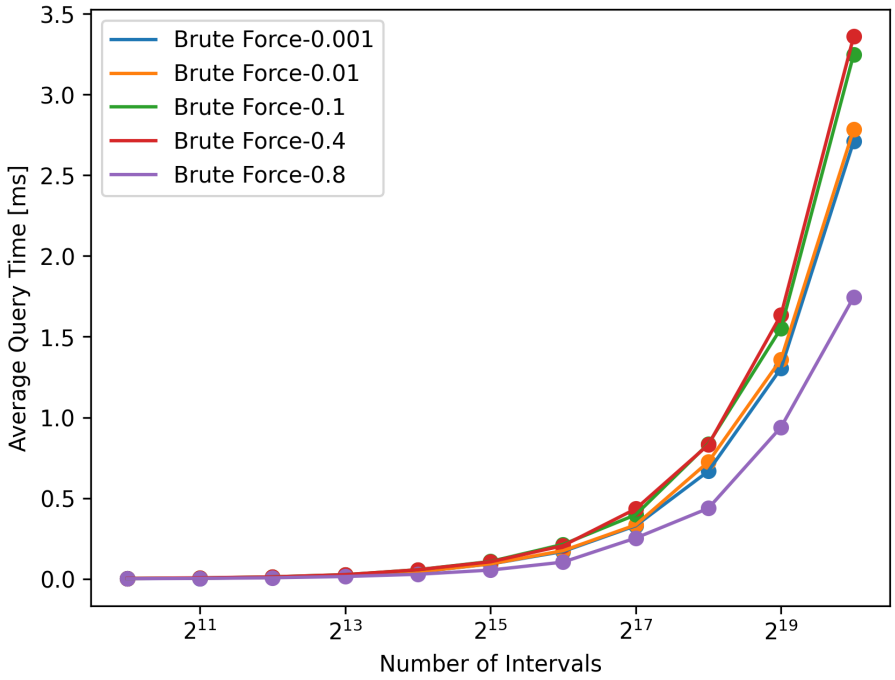


Figure 3: Average query time Brute Force by number of intervals.

In the last plot in Figure 5 we fix the number of intervals at $n = 2^{20}$ and change the parameter $p$. Clearly the query time of the Interval Tree linearly increase with the number of expected intersections. For the Brute Force method, the performance mainly depends on the branch miss prediction, which gets more predictable for extremer values of $p$, thus the hill like shape of the curve. If $p$ is larger than 0.5, the Interval Tree has only one level. Thus both methods essentially scan a list and check a condition for intersection. We think the differences in query time for $p \geq 0.5$ might have several reasons. The Brute Force method always has to scan the complete list, while the Interval Tree scans on average $pn$ elements. Also the branch miss prediction might be easier for Interval Trees, since it has a sequence of hits followed by one miss. In the Brute Force method, each branch is taken with probability $p$. For values of $p$ close to 1, the two methods do not differ significantly. Overall the Brute Force method is not faster, even for large number of intersections $k$.

# 4 Conclusion

In our work we explained a data structure, called Interval Trees, to solve the Line Stabbing problem efficiently and compared it with a simple Brute Force approach. We showed that on random instances where we fix the expected number of intersections, Interval Trees outperform the baseline considerably. The Brute Force method is more then 10 times slower for small number of intersections ($p \leq 0.3$) and $2^{20}$ intervals. When we fix $n$ and the number of intersections goes to $n$, the differences become smaller. For the Interval Tree the query time scales linearly with the number of intersections, whereas for the Brute Force method it mainly depends on the branch miss prediction done by the compiler.

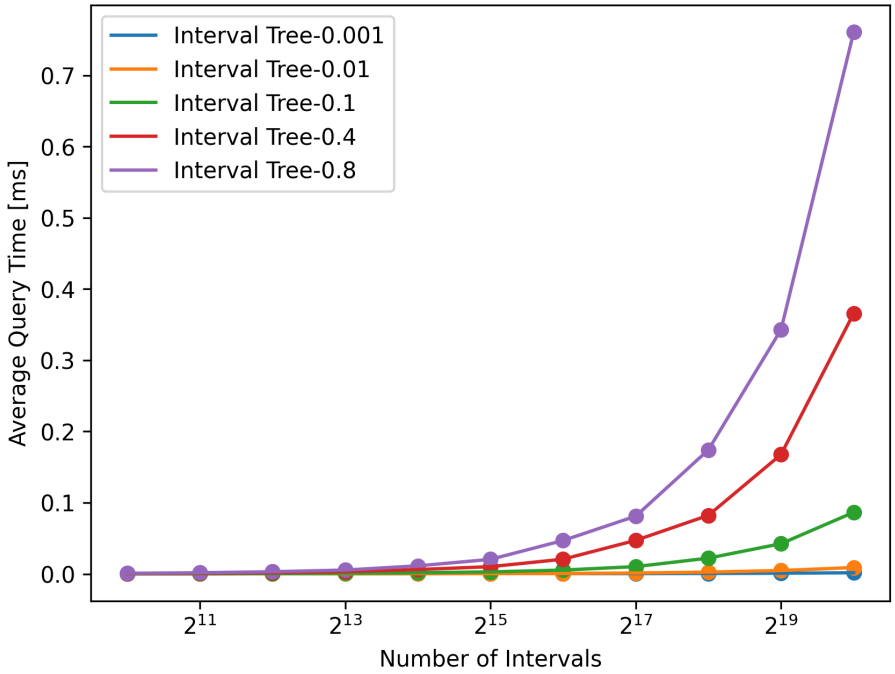


Figure 4: Average query time Interval Tree by number of intervals.

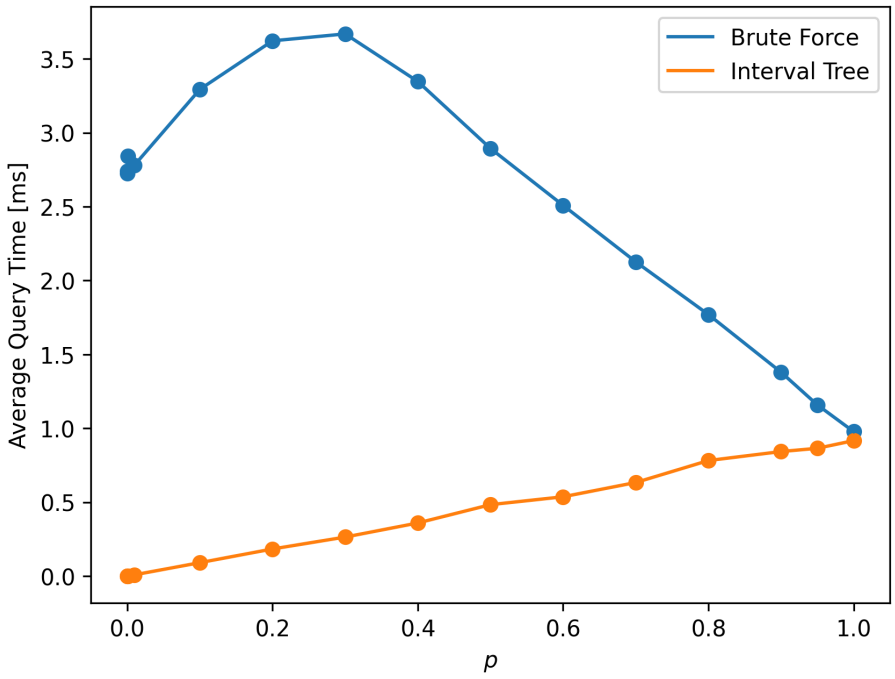


Figure 5: Average query time with $n = 2^{20}$ and varying $p$.