ADS - Final Project

GEOMETRIC DATA STRUCTURES

June 18, 2018

Kymry Burwell
UPC

Contents

Introduction

This paper introduces 4 different geometric data structures: range trees, segment trees, interval trees, and binary indexed trees. Each data structure is first explained in detail. Following this, tests are performed to validate the theoretical construction times of interval, segment, and binary indexed trees. Finally, an experimental comparison of query operations on interval and segment trees is performed.

Geometric Data Structures and Practical Uses

Geometric data structures are designed to capture the structural intricacies of geometric objects. They are used in various fields such as robotics, computer graphics, computer vision, integration circuit design, data mining, and geographical informations systems (GIS). Classic data structures such as simple trees, lists, and graphs often fail to sufficiently capture the structural properties of geometric objects and they don't always provide an efficient means for operations such as query and update to be performed. This is where the benefit of geometric data structures can be seen. For example, suppose we have a set of points in 2-dimensional space. If the points are stored in a simple array, a range query operation will take $O(n)$ time, however, if a geometric data structure is used, the same operation can be performed in $O(logn + k)$ time, where k is the number of points returned. This difference can be drastic if the dataset consists of millions of points.

In short, geometric data structures aim to: Capture structural information (such as dimensionality) of geometric objects, perform different types of query operations efficiently, perform updates (and oftentimes deletion) in an efficient manner, and use as little space as possible.

To motivate the discussion of the 4 geometric data structures that will be discussed in this paper, here are just a few examples of the many practical uses of range trees, segment trees, interval trees, and binary indexed trees.

- (Interval or segment tree) Windowing queries in which we have a set of line segments and a specific rectangular window and we want to find all of the line segments that intersect the window. For example, while using a GPS, find all roads that intersect a rectangular viewing area. Imagine zooming into a specific region of Europe and finding all the roads that pass through that specific region.

- (Range tree) Windowing queries in which we want to find all the line segments that have an endpoint in the window.
- (Segment tree) Range minimum or maximum queries. These are queries for finding the minimum or maximum value in a sub-array.
- (Binary Indexed Tree) Used in Arithmetic Coding Algorithms, which are encoding algorithms used in data compression.
- (Interval tree) Given a database of time intervals, determining which events occurred during a specific interval. For example, suppose you have a database that stores the daily high and low temperatures for all cities in Europe. You may wish to query the database and find all cities that had a temperature within a given range.
- (Interval tree) Used extensively in computer graphics. For example, used in video games to find all elements that are visible from a specific viewpoint.
- (Segment tree) Finding the maximum clique in a set of intervals.
- (Range tree) Segment intersection problems in which there is a set of segments and we want to find how many intersections there are between them.

INTERVAL TREES

Definition: Interval trees are a data structure that store intervals and provide an efficient means for maintaining a searchable database of intervals. At the core, they are a variation of a rooted balanced binary search tree. Consider a set of intervals, I. The root node of an interval tree created from I has a key that is equal to the median, m, of the endpoints of I. In addition, it has an auxiliary data structure that stores all the intervals that overlap with the m. It has pointers to two subtrees. The left subtree contains all of the intervals that lie fully to the left of m, and the right subtree contains those that fall to the right of m. Both of these subtrees are then recursively constructed in the same way. Figure 1 below, taken from [3], shows how an interval tree can be visualized. Note that this is just one variation of an interval tree.

Applications: Interval trees have many applications in practice that are comprised of two main types of queries.

- Type 1 - Queries asking questions of the form, "Given a set of Intervals, S, and a query point, q, which of the intervals contain the query point?"

- Type 2 - Queries asking questions of the form, “Given a set of intervals, S, and a query interval, Q, which of the intervals in S overlap with Q?”

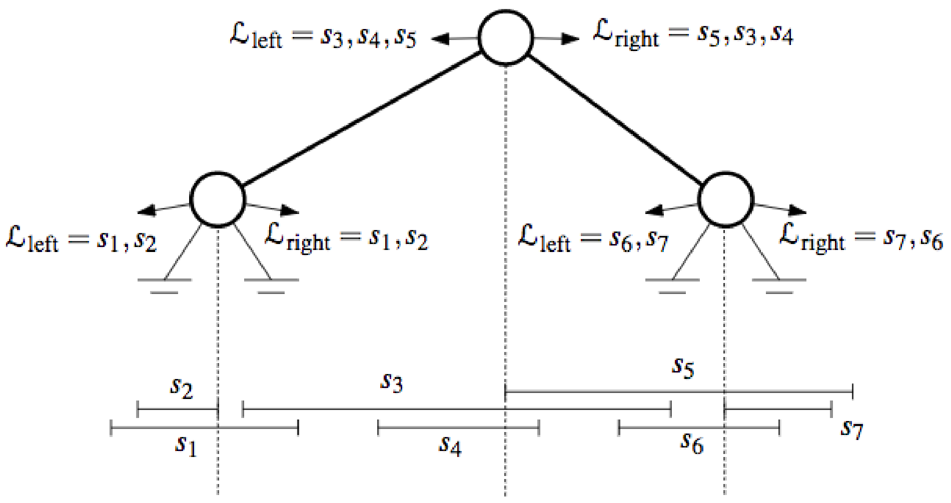
Interval trees excel at answering questions of type 2. Other data structures, such as segment trees (described below), are more tailored to answer queries of type 1.

Construction: There are two main variations of the interval tree: *Centered Interval Trees* [3] and *Augmented Interval Trees* [2]. *Centered Interval Trees* are constructed as follows:

1. Given a set of n intervals, find the median, M, of the endpoints of all intervals. M will act as the root node of the tree.
2. Separate the intervals into three distinct sets. Set_Low consists of all intervals that lie completely to the left of M. Set_High consists of all intervals that lie completely to the right of M. Set_Mid contains all intervals that contain M.
3. Next, Set_Low and Set_High are recursively divided in the same manner until there are no intervals left.
4. The intervals contained in Set_Mid are stored in a separate data structure consisting of two lists. One containing all intervals sorted by their left end point and the other sorted by their right end point.

An *Augmented Interval* Tree is constructed as follows:

1. Given a set of n intervals, construct a binary search tree using the left endpoint from each node.
2. An additional data point is added to each node, recording the maximum right endpoint of all the intervals in its subtrees, including itself.
3. This then produces a binary search tree in which each node contains an interval.



$\mathcal{L}_{\text{left}} = s_3, s_4, s_5$
$\mathcal{L}_{\text{right}} = s_5, s_3, s_4$
$\mathcal{L}_{\text{left}} = s_1, s_2$
$\mathcal{L}_{\text{right}} = s_1, s_2$
$\mathcal{L}_{\text{left}} = s_6, s_7$
$\mathcal{L}_{\text{right}} = s_7, s_6$
s_1
s_2
s_3
s_4
s_5
s_6
s_7

Figure 1: Centered Interval Tree

Performance: In single dimension, interval trees take $O(n)$ space, can be constructed in $O(nlogn)$ and can be queried in $O(k + logn)$ where k is the number of intervals that are returned by the query.

Range Trees

Definition: A Range Tree is an ordered data structure that holds a list of points in ≥ 1 dimensions. Consider a set, S, of points in d-dimensional space. Each point can be considered a d-tuple of the form $(p^1, p^2, p^3, ..., p^d)$. A *Range Tree* supports queries of the form $q : [l^1, r^1] \times [l^2, r^2] \times ... \times [l^d, r^d]$. The output of the query is a set of points, P, such that for each $p \in P$, $p^i \in [l^i, r^i]$ for all $i = 1, 2, ..., d$. In one dimension, a range tree can be simply a binary search tree or a sorted array. A range tree that is 2 or more dimensions is a multi-level binary search tree, where each level is a binary search tree on one of the dimensions.

Applications: Range trees help answer questions of the form, "which points fall within a given interval?".

Construction: Using a recursive algorithm, range trees can be constructed in $O(nlog^{d-1}n)$ time. In order to better visualize range trees, consider a set of points that consist of two coordinates, the x-coordinate and the y-coordinate. A range tree for these points will consist of a binary search tree, T, built on the x-coordinates. For each node, $v \in T$, let's define the set of points contained in the subtree of v to be the set of points, p(v). The node v stores a pointer to a binary search tree containing p(v) sorted on the y-coordinate. Figure 2, taken from [3], shows a 2-dimensional range tree using x and y coordinates. As described above, the primary tree contains the x-coordinates, and each node has an associated binary search tree data structure built on the y-coordinates.

Performance: In general, a range query can be performed in $O(k + log^d n)$, where k is the number of points returned by the query and d is the number of dimensions. However, using a technique called fractional cascading, this can be reduced to $O(k + log^{d-1} n)$. As we can see, as the number of dimensions increases, both query and construction time increases quite rapidly and therefore range trees are not very effective in high dimensions. They take $O(nlog^{d-1} n)$ space and can be constructed in $O(nlog^{d-1} n)$ time.

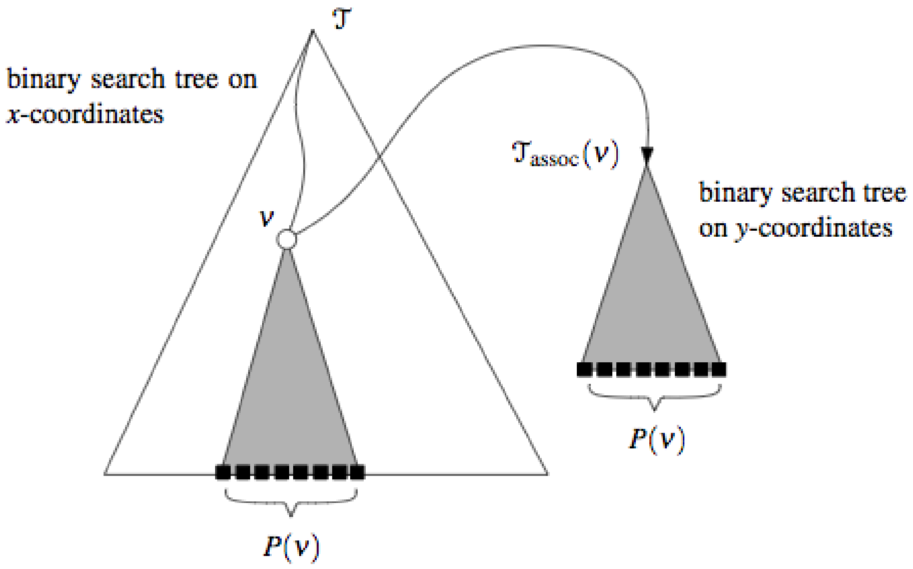

$\mathcal{T}$
binary search tree on x-coordinates
$\mathcal{T}_{assoc}(v)$
binary search tree on y-coordinates
v
$P(v)$
$P(v)$

Figure 2: Range Tree [3]

Segment Trees

Definition: A segment tree is a type of augmented binary search tree used for storing intervals, in which each node of the tree is associated with an interval. Given a set S of intervals, and an array I, which consists of the ordered endpoints of the intervals with duplicates removed, the root of a segment tree will represent the entire array $I[0 : N-1]$ where N is the size of I. Each leaf of the tree is an elementary interval in the array. The leftmost leaf corresponds to the leftmost elementary interval, and the rightmost leaf corresponds to the rightmost elementary interval in I. The internal nodes correspond to the union of the intervals of its subtree. Each node or leaf has an augmented data structure that stores a subset of the intervals from S such that each the node interval is fully contained within every interval in it's subset and it's parent node's interval is not. In other words, let $Int(v)$ be the interval of node v. The subset from S stored in node v's augmented data structure all have the properties such that $Int(v) \subset [x : x']$ and $Int(parent(V)) \not\subset [x : x']$. Figure 3, taken from [3] demonstrates this. One last note is that segment trees have the unique feature that each interval is stored at most twice at each level of the tree.

Applications: Segment trees help answer questions of the form, "Which of these intervals contain a given point?", also known as Stabbing Queries. They can be useful in determining the measure of a set of intervals, that is, the length of the union of the set. In addition, they can be used to find the maximum clique of a set of intervals. As with the other data structures presented, segment trees can be generalized to higher dimensions.

Construction: Segment trees can be constructed using a recursive algorithm. The input is the an array $I[s : t]$ that is the set of ordered endpoints of the intervals to be inserted. The algorithm called $contructSegmentTree(s, t)$ is as follows.

1. Let v be a node, v.B = s, v.E = t, v.left = v.right = nullptr, v.aux is the empty auxiliary data structure.
2. If s + 1 = t, return
3. Let v.key = m = $\lfloor \frac{v.B + V.E}{2} \rfloor$
4. v.left = constructSegmentTree(s,m)
5. v.right = constructSegmentTree(m,t)

With this tree skeleton, the intervals are then inserted one by one.

Performance: Segment trees have both storage space and construction time of $O(nlogn)$ and query time of $O(logn + k)$, where k is the number of intervals returned.

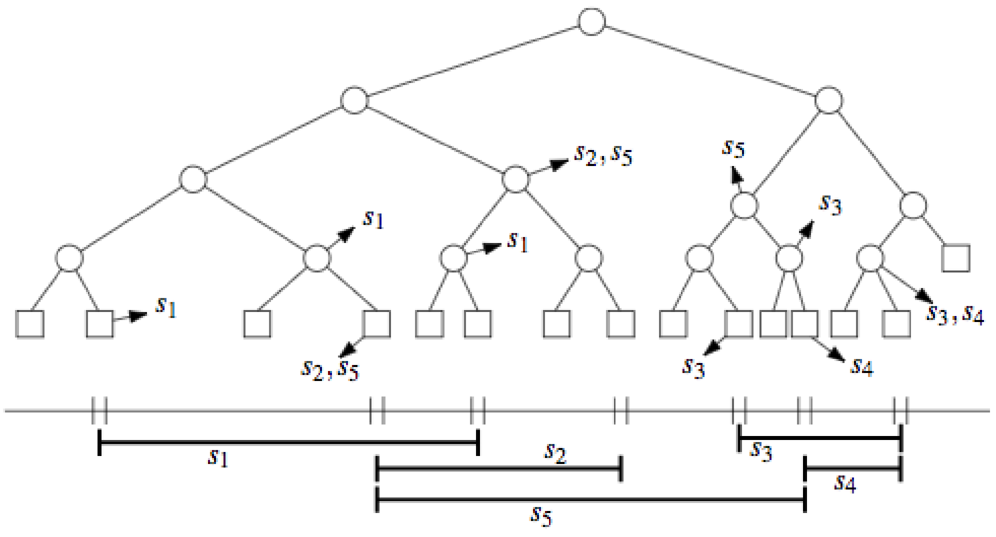


Figure 3: Segment Tree [3]

Binary Indexed Trees

Definition: A Binary Indexed Tree (BIT), also known as a Fenwick tree, is a simple array data structure that is based on a binary tree and is used to store cumulative frequencies. That is, it stores precomputed prefix sums in order to allow fast range queries and index updates. It provides a middle ground in performance compared to normal arrays and prefix trees.

- **Array:** update in $O(1)$ and range/prefix query in $O(n)$.
- **Prefix Tree:** update in $O(n)$ and range/prefix query in $O(1)$.
- **Binary Indexed Tree:** update in $O(log n)$ and range/prefix query in $O(log n)$

As we can see, binary indexed trees provide a good trade-off between update and query performance.

Applications: BITs answer questions of the form, "How many items are there between index m and n?". That is, they answer prefix and range queries. For example, given an array $I[1,2,4,1,2,5,3,18]$. We can determine how many elements are in the range $I[3,5]$ (8 elements), and we can determine how many elements prefix $I[3]$ (8 elements).

Construction: Binary indexed trees are so named because during construction, query, and update procedures, the binary form of the indexes are utilized. They start at index 1 instead of 0 and the first element in the array is simply ignored. The following pseudo-code is just one way to construct a BIT on an array of integers.

Algorithm 1 Construct BIT

Input: Array I

```
1: for each index in I do
2:     index2 = index + (index & -index)
3:     if I[index2] < array.size() then
4:         array[index2] += array[index]
```

In line 2 of the above algorithm, the & is the bitwise and operator. Figure 4 below shows an example of a BIT. The rectangles below the array show the range sum that each element stores. For example, element 4 stores the sum of elements 1-4 and element 7 stores the sum of elements 6-7.

Performance: As mentioned previously, binary indexed trees can be updated and queried in $O(logn)$ time. They can be constructed in either $O(n)$ or $O(nlogn)$ time depending on which construction method is used. The method above is $O(n)$ time. As BIT are simply arrays they take $O(n)$ space.

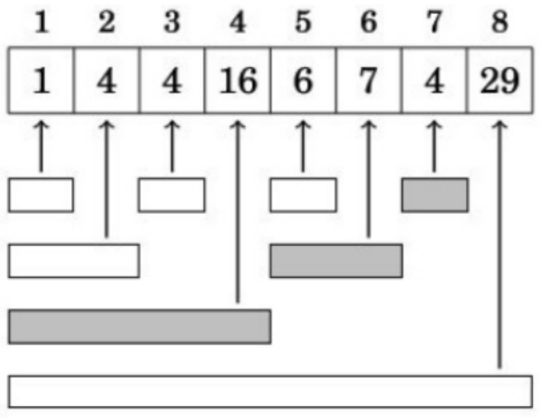


Figure 4: Binary Indexed Tree

Performance Summary

Figure 5 below summarizes the performance of the four types of trees explained in this paper. As we can see, the average performance is quite similar among the four types. However, it is important to keep in mind the type of queries that each structure is optimized for. Range trees are generalized to higher dimensions as this data structure is often used for 2 or more dimensions.

	Segment Tree	Interval Tree	BIT	Range Tree ≥ 2 dimensions(d)
Construction	$O(nlogn)$	$O(nlogn)$	$O(n)$	$O(nlog^{d-1}n)$
Query	$O(k+logn)$	$O(k+logn)$	$O(logn)$	$O(k+log^{d}n)$
Space	$O(nlogn)$	$O(n)$	$O(n)$	$O(nlog^{d-1}n)$

Figure 5: Tree Performance

Experiments to be Performed

The objective of the experiments is two fold. First, to validate the theoretical construction time of segment, interval, and binary indexed trees. Second, to compare point and interval overlap query performance between segment and interval trees. My goal is to validate the claim that segment trees are optimized for point (i.e. stabbing) queries and interval trees are optimized for interval overlap queries.

For all of the experiments, I used a random number generator in c++ that draws uniformly and independently at random from a provided range.

In order to test the theoretical construction time of interval and segment trees, I generated random intervals of integers with varying ranges. Using these intervals I then constructed multiple trees, starting from size n = 1,000 to n = 100,000, increasing n by 1,000 each time. I repeated this with the following interval ranges: [0,10], [0,100], [0,1000], [0,10000], [0,100000]. The time is recorded in milliseconds.

In order to test the theoretical construction time of binary indexed trees, I populated a vector with $1x10^6$ randomly generated integers in the range of $[0,100000]$. These integers acted as the starting element counts for each index. Using this vector, I then constructed the binary indexed tree. I repeated this process multiple times, increasing the tree size by $1x10^6$ elements each time. The time is recorded in milliseconds.

In order to compare point and range overlap queries between segment and interval trees, I first constructed trees of size n = 50,000 with an interval range of [0,5000]. I used the same set of intervals for both trees. I then generated q = 500 queries and ran the query search on both trees using the same set of queries. I recorded the time to perform all queries in seconds. I repeated this process multiple times increasing q by 1,000 each time, until q = 10,500. I then repeated the entire process again but increased the interval range to [0,50000]. I did this for both point and interval overlap queries. This is further explained in the experimental results section.

All experiments were conducted in c++ and R was used to plot the results.

Experimental Results

Tree Construction Performance

Segment Tree: Figure 6 below shows the construction time, in milliseconds, of segment trees with varying parameters. The Y-axis is the total construction time. The X-axis is the number of intervals added during the construction of the tree. Each colored line represents varying interval ranges. For example, the red line only allows intervals in the range of $[0,10]$ and the bright green line allows intervals in the range of $[0,1000]$. As we would expect, the construction time increases as the number of intervals increases. In addition, with small interval sizes, the construction time doesn't increase greatly when the number of intervals increases. However, this isn't the case with larger interval ranges. For example, when the interval range is limited to $[0,10]$, a tree with 100,000 intervals takes only a couple hundred milliseconds longer to construct than a tree with 25,000 intervals. In contrast, a tree with 100,000 intervals in the range of $[0,100000]$ takes over 1,500 milliseconds longer to construct than a tree with 25,000 intervals.

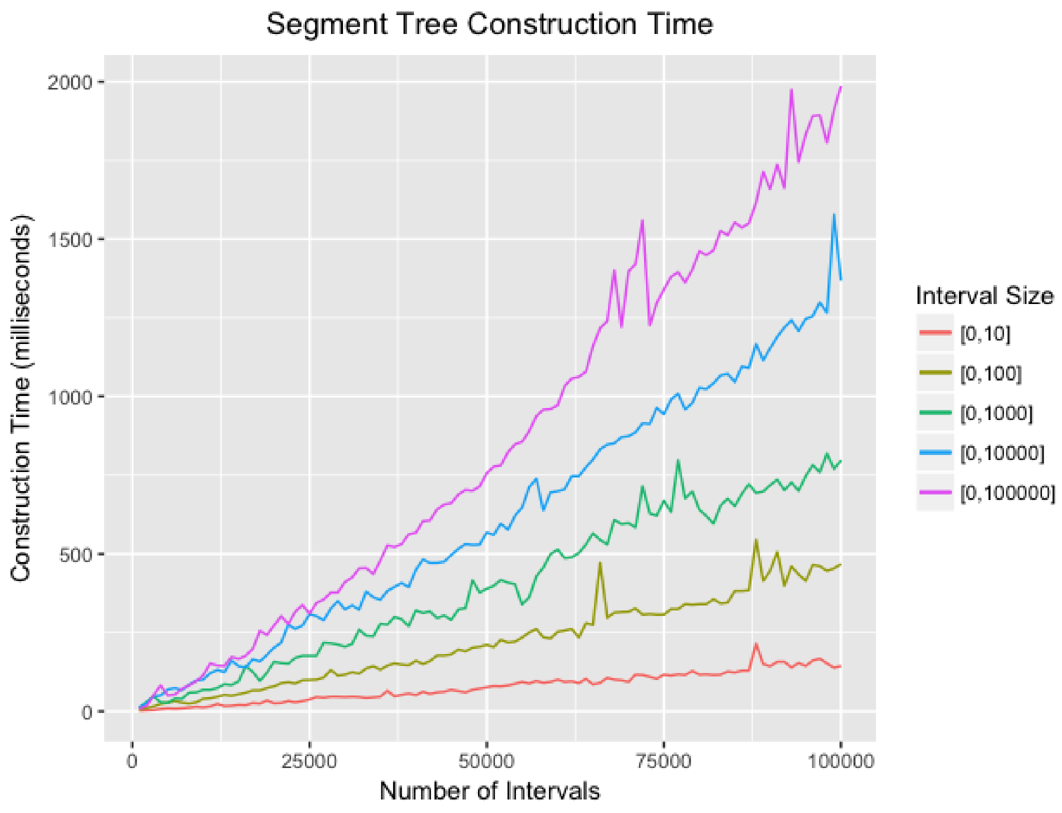


Figure 6: Segment Tree Construction Performance

Interval Tree: Figure 7 below shows the construction time, in milliseconds, of interval trees with varying parameters. Again, the X-axis is the number of intervals added during the construction of the tree and the colored lines represent different interval ranges. As is the case with segment trees, the interval size affects the construction time of the tree. As the interval sizes increase, the construction time increases. However, this isn't as pronounced as it was with segment trees.

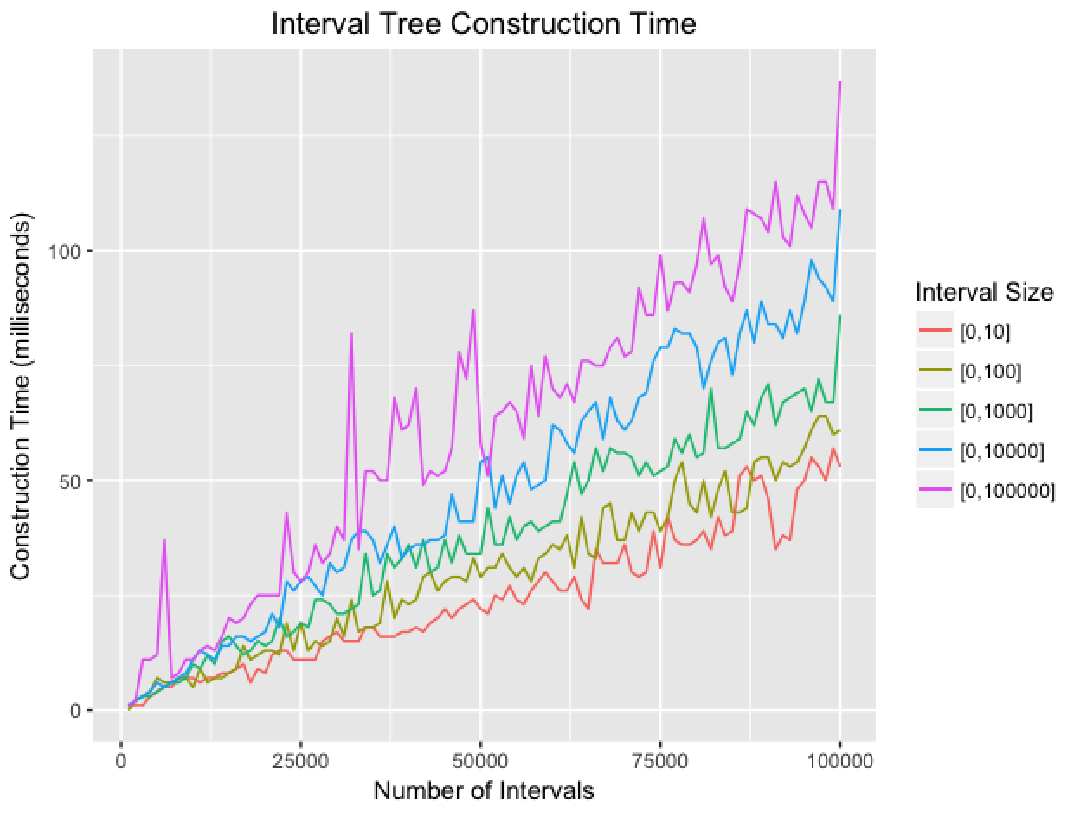


Figure 7: Interval Tree Construction Performance

Comparing figure's 6 and 7 above, we can see that segment trees take significantly longer to construct than interval trees. I believe much of this has to do with the implementations I chose. I made the segment trees a bit more robust than the interval trees and I believe this adds to the increased construction time. However, the important concept to see is that construction time of both trees increases at a similar asymptotic rate. In short, both appear to follow the theoretical construction rate of $O(nlogn)$.

Binary Indexed Tree: Figure 8 below shows the total construction time of binary indexed trees of varying sizes. The Y-axis is the construction time in milliseconds and the X-axis is the number of elements in the tree (in millions). As we can see, the construction time increases linearly with the tree size, thus validating the construction time of $O(n)$.

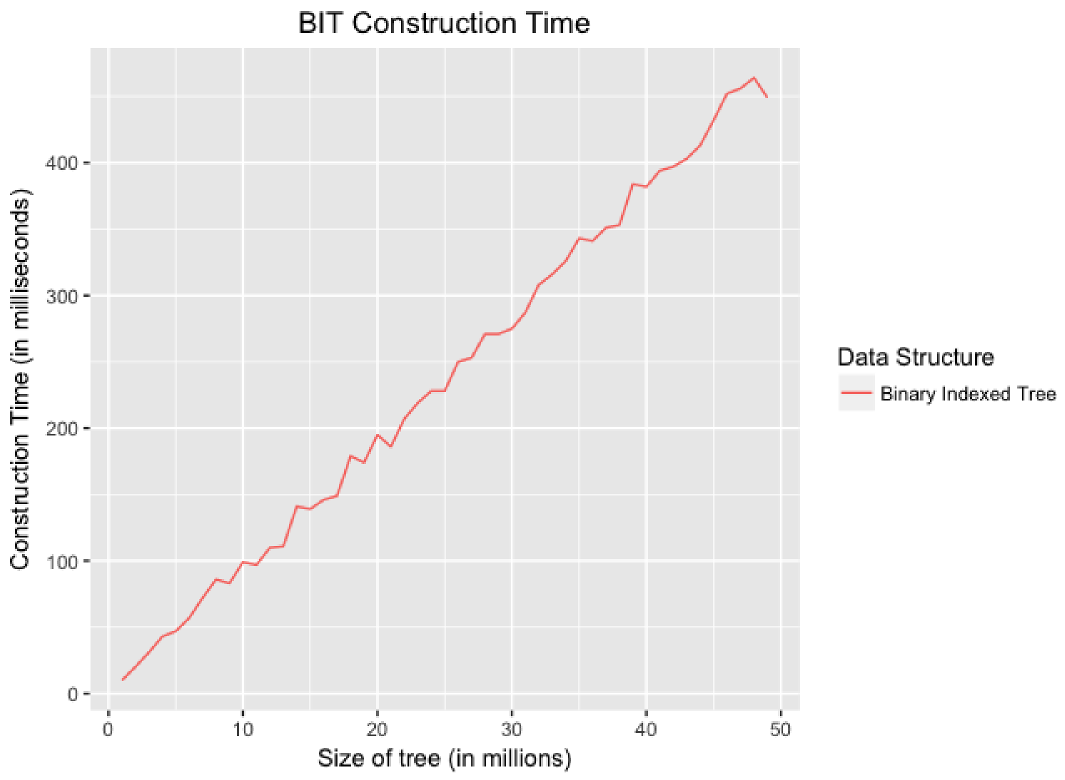


Figure 8: Binary Indexed Tree Construction Performance

Point Query Performance

This sections presents the results of performing point queries (also known as stabbing queries) on both the interval and segment tree data structures. A point query determines which intervals contained in the tree intersect a given point. In order to show the benefit of using a tree structure to perform queries of this type, I also included a standard array search. The array simply contains every interval and the search uses a brute force approach by checking every element of the array for intersection. The next two graphs present the results. The first graph includes intervals in the range of $[0, 5000]$ and the second increases the range of the intervals to $[0, 50000]$. I decided to use two different ranges to see how this might affect the queries. In both cases, 50,000 intervals were used to create the data structures. The Y-axis is the total time to perform all queries (in seconds) and X-axis is the number of queries performed. For all experiments, the same intervals were used to to create the data structures, and the same queries were used.

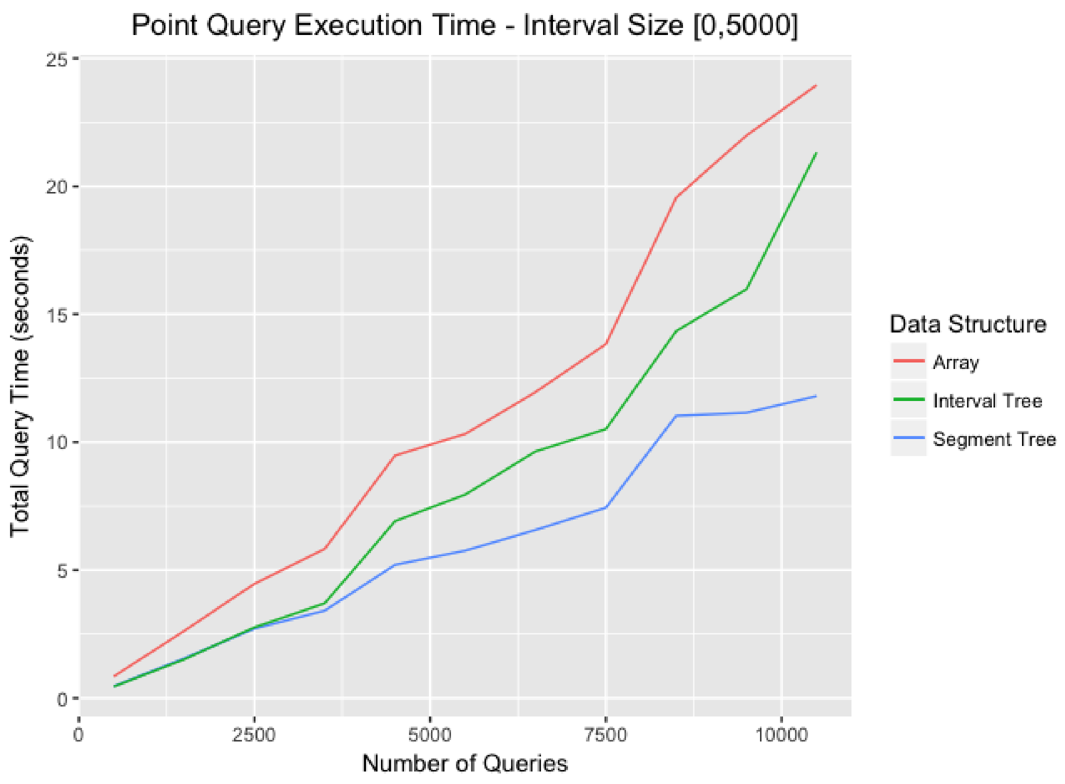


Figure 9: Point Query Performance - [0,5000]

In figure 9 above, we can see that the segment tree performed the best, followed by the interval tree, and lastly the standard array. This is expected, as segment trees are optimized to perform point queries, whereas interval trees are really best used for overlapping interval queries. Additionally, we can see that as the number of queries increased, the segment tree started improving compared to the others and the benefit became more pronounced.

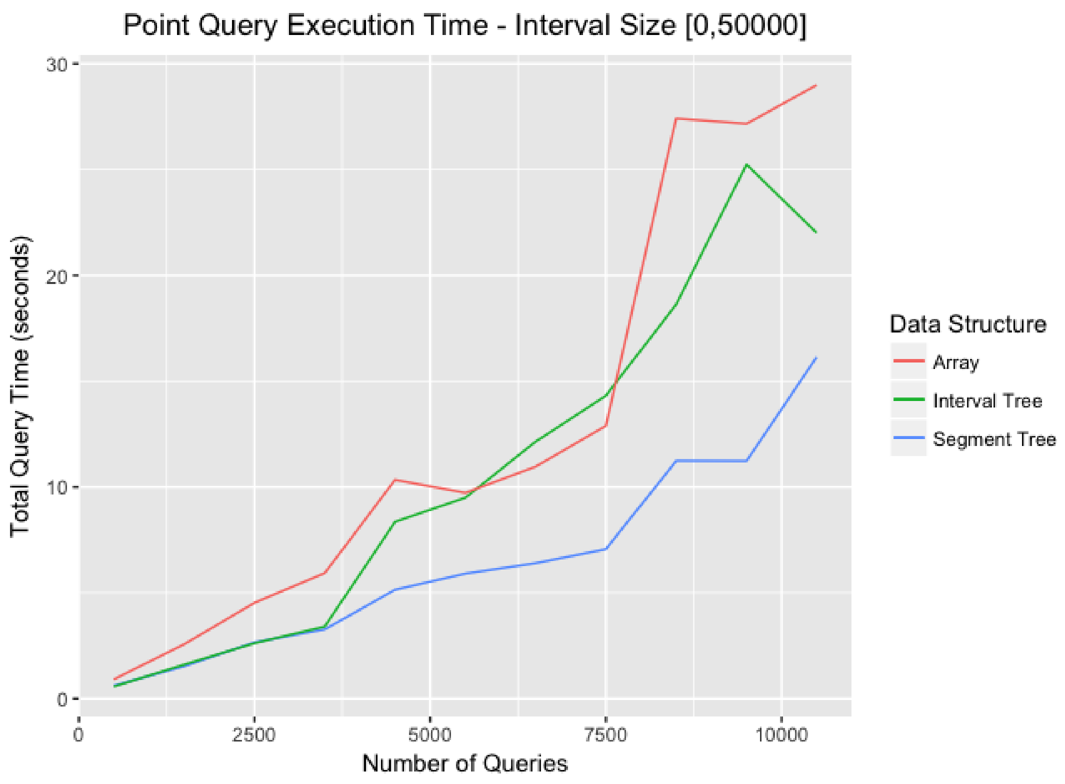


Figure 10: Point Query Performance - [0,50000]

In figure 10 above, we can see that the segment tree again performed the best when the interval range increase to [0,50000], while the interval tree and array both performed poorly. As one would expect, the overall execution time also increased for all three of the data structures due to the increased range size.

Overlap Query Performance

This section presents the results of performing overlap interval queries on both the segment and interval trees, as well as the standard array. An overlapping interval query determines which intervals in the data structure overlap with a query interval. As with the point query, I performed two different experiments, one with an interval range of [0, 5000] and another with a range increased to [0, 50000]. I used a total of 50, 000 intervals in constructing the data structures. As before, the same intervals were used to create all three data structures and the same queries were used. The Y-axis is the total time to perform all queries (in seconds) and the X-axis is the number of queries performed.

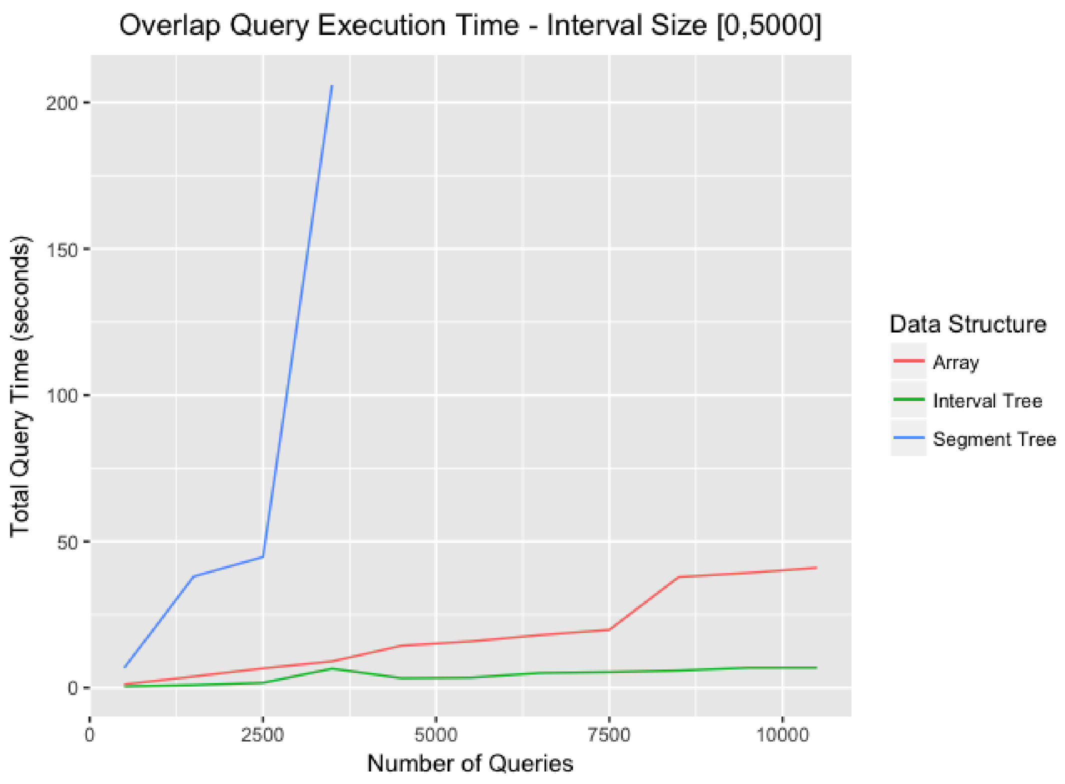


Figure 11: Overlap Query Performance - [0,5000]

From figure 11 above, we see that the interval tree was the best by far, followed by the array and finally the segment tree. It's a bit surprising that the segment tree took so much longer to perform the queries than the other structures. Because of this, I stopped the segment tree search after performing 3,500 queries. In general, segment trees aren't optimized to perform overlapping queries. This is partly due to the fact that it is often impossible to determine which subtree to check when executing the search, so both subtrees must be checked, thus increasing the search time. Additionally, a segment tree may store any given interval twice per level, therefore increasing the size of the search output, assuming duplicates are allowed in the output.

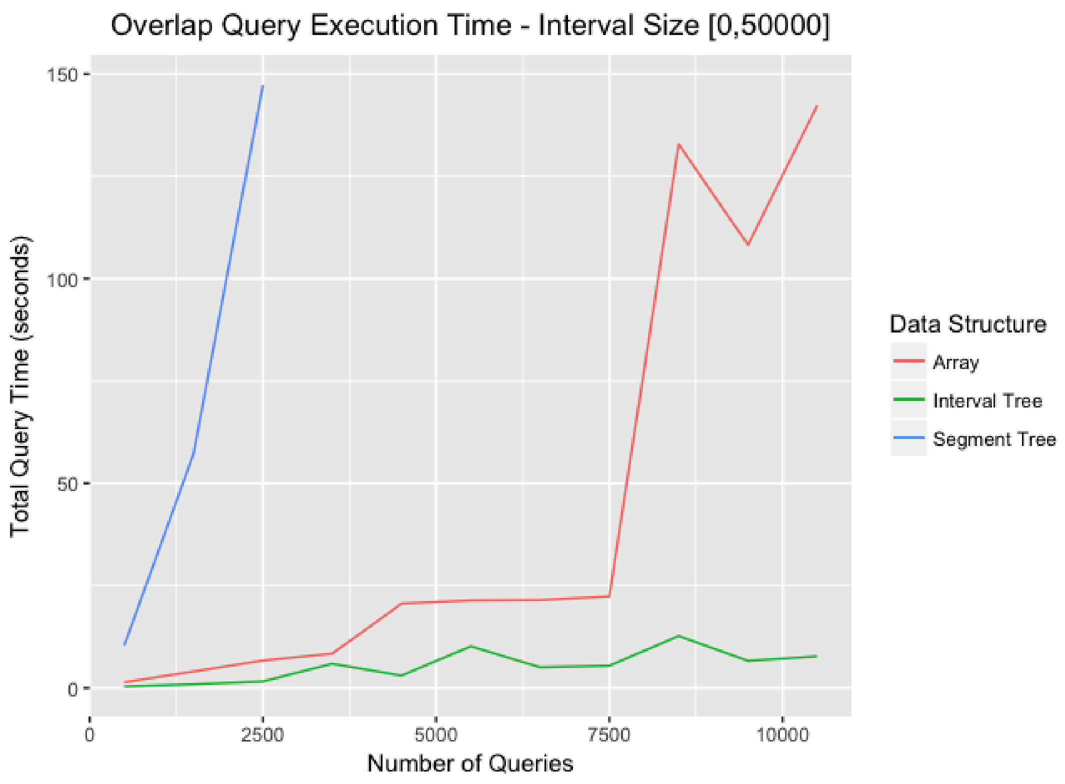


Figure 12: Overlap Query Performance - [0,50000]

In figure 12 we see a similar performance of the segment and interval trees when the range size is increased to $[0, 50000]$. However, the array takes significantly longer as the number of queries increases.

In conclusion, we can see that interval trees are best for overlap queries, while segment trees are best for point queries. If only a few intervals are being stored and only a few queries are being performed, an array might be the easiest option, simply because any modern programming language has built in arrays that can be utilized. However, as the tree size or number of queries to perform increases, using a segment tree or interval tree becomes more appealing.

FINAL THOUGHTS

As we can see from the experiments, the construction time of segment, interval, and binary indexed trees follow the theoretical bounds. Furthermore, we saw that depending on the interval range size, the construction time can fluctuate quite greatly, although the asymptotic bounds remain the same. Additionally, we saw that the optimal choice of data structure depends heavily on the type of query to be performed. This was tested with overlapping and stabbing queries, however, we can infer that this will likely be true with other types of queries and geometric data structures as well. In conclusion, it is not only important to know the different types of data structures that exist, but which types of queries each is optimized for.

Bibliography

[1] Mehta, Dinesh P., and Sartaj Sahni, eds. Handbook of data structures and applications. CRC Press, 2004.

[2] Cormen, Thomas H. Introduction to algorithms. MIT press, 2009.

[3] De Berg, Mark, et al. "Computational geometry." Computational geometry. Springer, Berlin, Heidelberg, 2000. 1-17.

3