Advanced Data Structures

Free Implementation of Interval Trees

Jannik Hösch

May 13, 2024

1 Introduction

The linear stabbing problem is a fundamental problem in computational geometry, where the goal is to efficiently determine which of a set of intervals contain a given query point. An Interval Tree is an advanced data structure designed to solve such problems by organizing intervals in a way that allows for fast querying. At its core, the Interval Tree is a binary search tree where each node stores an interval and potentially additional data structures to manage overlapping intervals efficiently. The tree partitions the space such that queries about which intervals overlap with any given point or another interval can be answered quickly.

This report discusses the implementation, experimentation, and evaluation of Interval Trees in the context of solving the linear stabbing problem. I aim to demonstrate the effectiveness and efficiency of this data structure under various configurations and query loads.

2 Implementation

The Interval Tree implemented in this project is designed to efficiently solve the linear stabbing problem, which involves identifying all intervals that contain a given query point. This section provides a brief description of the structure and functionality of my Interval Tree.

Each node in the Interval Tree contains a median value, computed from the start- and endpoints of the intervals. This median serves to categorize the intervals into three distinct groups: left intervals, which lie entirely to the left of the median; right intervals, entirely to the right; and crossing intervals, which span the median. To facilitate efficient queries, crossing intervals at each node are sorted into two lists: 'Crossing Left', sorted by the starting points of the intervals, and 'Crossing Right', sorted by the ending points in reverse order.

The query mechanism determines whether a point falls within any of the stored intervals by traversing the tree. If the query point is less than the node's median, the function checks against the 'Crossing Left' list; if greater, it checks against 'Crossing Right'. In cases where the point matches the median, all crossing intervals are returned. The tree traversal continues recursively as needed into either the left or right subtree. This process benefits from the sorted order of crossing intervals, enabling early termination of the search when it is clear no further intervals in the list can contain the point.

3 Experiments

The experiments are designed to evaluate the performance of the Interval Tree and compare it to a naive interval querying method. For different numbers of intervals I measured the build time and memory usage of the Interval Tree and calculated the average time per query.

The data set consists of randomly generated intervals with endpoints within the range 1 to 1000. The number of intervals tested ranges up to 100000, incremented by 1000 intervals per experiment

iteration. For each interval set an Interval Tree is built and the building time and memory usage measured. Then, 100 random point queries are executed that are solved both by the Interval Tree method and a naive method, that checks every interval linearly if it contains the query point. The taken time for each query is recorded and averaged in the end.

4 Results

The results obtained from my experiment illustrate the performance characteristics of the Interval Tree in terms of memory usage, build time, and query efficiency.

Each interval is stored twice within the tree structure - in the sorted 'Crossing Left' and 'Crossing Right' lists of one node. The observed linear increase in memory consumption as the number of intervals increases as shown in Figure 1 aligns well with this theoretical assumption of a storage complexity of $O(n)$.

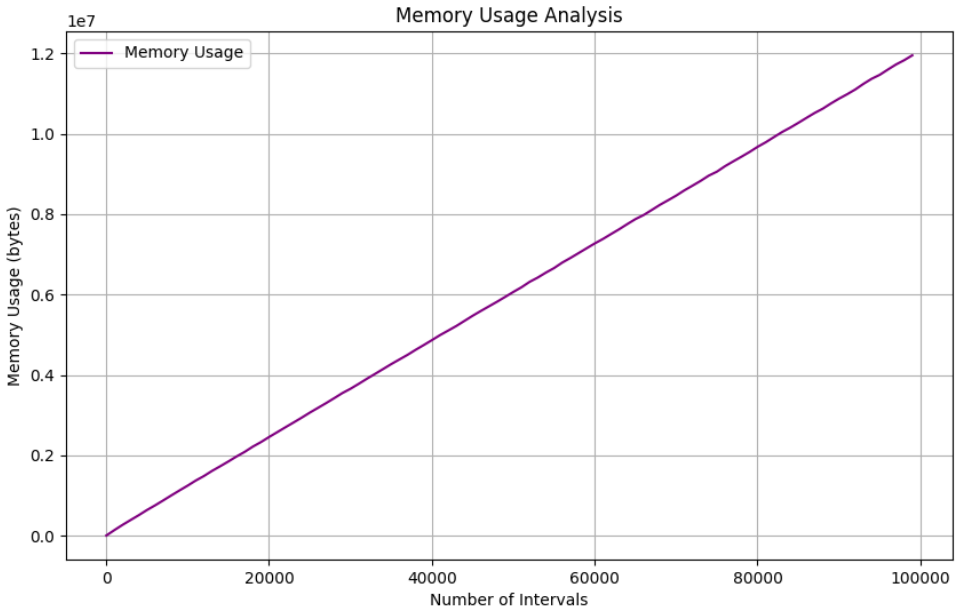

Figure 1: Memory Usage Analysis.

The build time of the Interval Tree (Figure 2) showed an increasing trend, consistent with the preprocessing complexity of $O(n \log n)$. This complexity arises from the need to sort interval endpoints and to determine medians, which are necessary steps in constructing the tree's nodes optimally. The graph indicates that the practical performance follows this expected complexity, with slight fluctuations possibly due to variations in the data distribution or the overhead of recursive tree construction.

The query time comparison between the Interval Tree and the naive method highlights the efficiency of the Interval Tree (Figure 3). The naive method's performance degrades linearly as the number of intervals increases, which is expected given its $O(n)$ complexity per query. In contrast, the Interval Tree maintains a much lower increase in query time, adhering to the $O(\log n + K)$ complexity, where K is the number of intervals in the query result. If all intervals contain the query point the query time is equal to the naive method, because all intervals are checked. The bigger the number of non-overlapping intervals the lower the query time of the tree method, as this enables the efficient usage of the data structure.

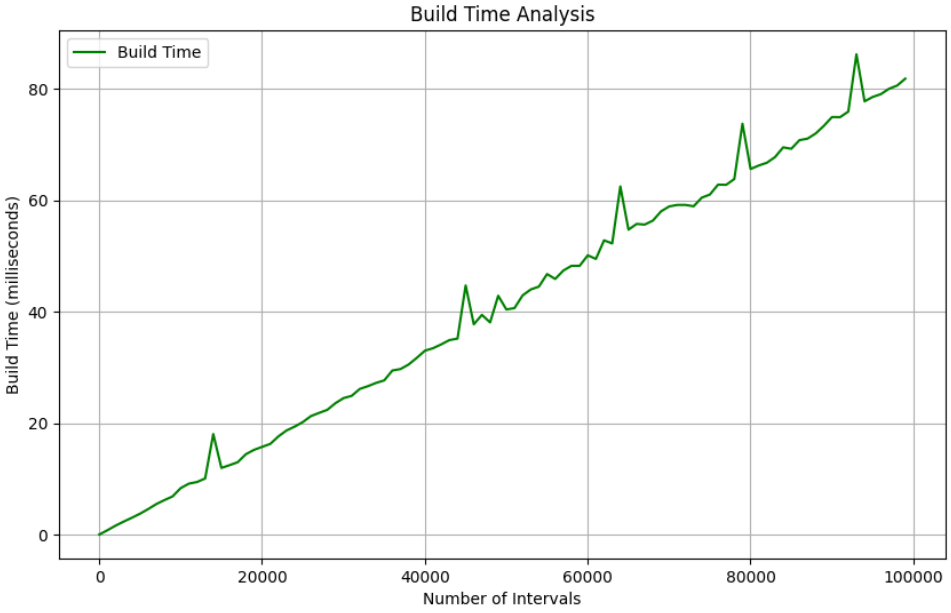

Figure 2: Build Time Analysis.

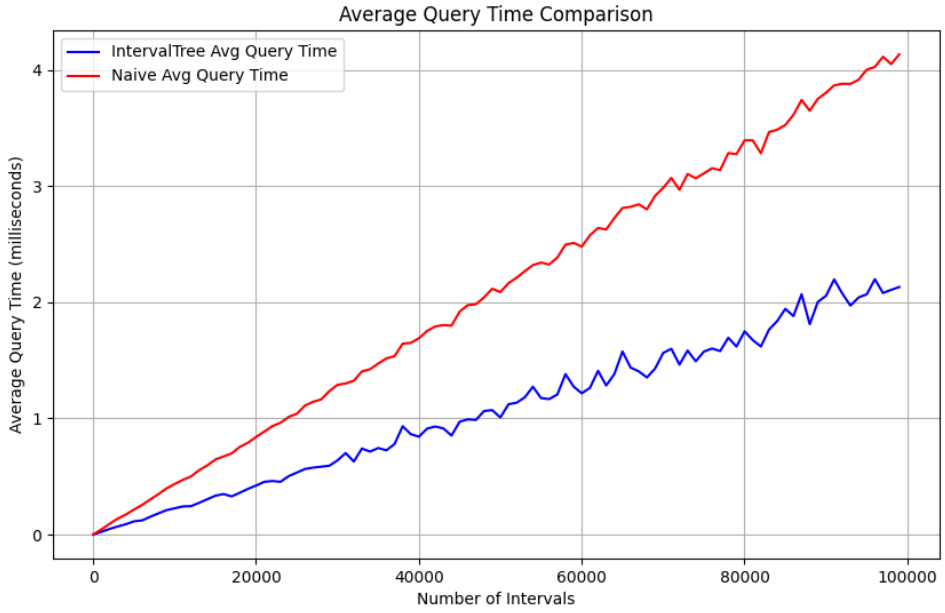

Figure 3: Query Time Analysis.

5 Conclusion

This project validated the effectiveness of the Interval Tree for solving the linear stabbing problem, demonstrating significant improvements in query efficiency over a naive approach. With build and query time complexities of $O(n \log n))$ and $O(\log n + K)$ respectively, the Interval Tree efficiently balances build time, memory usage, and query speed, confirming its theoretical advantages. The results underscore the Interval Tree's suitability for applications that demand fast and frequent interval data queries.

6 Appendix

6.1 IntervalTree.py

```python
class IntervalTreeNode:
    """
    A node in the Interval Tree, representing a range of values with median-based
    partitioning.
    """
    def __init__(self, median, crossing_left, crossing_right):
        self.median = median
        self.left = None
        self.right = None
        self.crossing_left = crossing_left
        self.crossing_right = crossing_right


class IntervalTree:
    """
    Data structure to efficiently find all intervals that overlap with any given
    point or interval.
    """
    def __init__(self, intervals):
        self.root = self.build_tree(intervals)

    def build_tree(self, intervals):
        """Recursively build the interval tree from a list of intervals."""
        if not intervals:
            return None

        # Calculate median
        endpoints = sorted(p for interval in intervals for p in interval)
        median = endpoints[len(endpoints) // 2]

        # Create partitions
        left, right, crossing = [], [], []
        for start, end in intervals:
            if end < median:
                left.append((start, end))
            elif start > median:
                right.append((start, end))
            else:
                crossing.append((start, end))

        # Sort partitions
        crossing_left = sorted(crossing, key=lambda x: x[0])
        crossing_right = sorted(crossing, key=lambda x: x[1], reverse=True)

        # Create tree node
        node = IntervalTreeNode(median, crossing_left, crossing_right)

        # Recursively build left and right subtree
        node.left = self.build_tree(left)
        node.right = self.build_tree(right)
```

```
49
50          return node
51
52      def query(self, q, node=None):
53          """Query the tree for intervals containing the point q, starting from the
    node (or root by default)."""
54          if node is None:
55              node = self.root
56
57          result = []
58          if q < node.median:
59              for i in node.crossing_left:
60                  if i[0] <= q <= i[1]:
61                      result.append(i)
62                  else:
63                      break
64              if node.left:
65                  result += self.query(q, node.left)
66          elif q > node.median:
67              for i in node.crossing_right:
68                  if i[0] <= q <= i[1]:
69                      result.append(i)
70                  else:
71                      break
72              if node.right:
73                  result += self.query(q, node.right)
74          else:
75              result += node.crossing_left
76
77          return result
```

6.2 experiment.py

```
1  from IntervalTree import IntervalTree
2  import random
3  import time
4  import matplotlib.pyplot as plt
5  from pympler import asizeof
6
7
8  def naive_stabbing(intervals, q):
9      """Perform a naive stabbing query to find intervals containing the point q."""
10     return [interval for interval in intervals if interval[0] <= q <= interval[1]]
11
12
13 def generate_intervals(n):
14     """Generate a list of random intervals."""
15     intervals = []
16     for _ in range(n):
17         start = random.randint(1, 1000)
18         end = random.randint(1, 1000)
19         if start > end:
20             start, end = end, start
21         intervals.append((start, end))
22     return intervals
23
24
25 def run_experiment(n_intervals):
26     """Run the experiment on a set number of intervals, collecting data on build and
       query times."""
27     print(f"Running experiment for {n_intervals} intervals...")
28     intervals = generate_intervals(n_intervals)
29
30     # Build the tree with random intervals
31     build_start = time.time()
```

```
    tree = IntervalTree(intervals)
    build_time = (time.time() - build_start) * 1000

    # Perform 100 random queries
    query_times_tree = []
    query_times_naive = []
    for _ in range(100):
        q = random.randint(1, 1000)

        # Solve with tree
        query_start = time.time()
        tree.query(q)
        query_times_tree.append(time.time() - query_start)

        # Solve naive
        query_start = time.time()
        naive_stabbing(intervals, q)
        query_times_naive.append(time.time() - query_start)

    avg_query_time_tree = (sum(query_times_tree) / len(query_times_tree)) * 1000
    avg_query_time_naive = (sum(query_times_naive) / len(query_times_naive)) * 1000
    memory = asizeof.asizeof(tree)

    return build_time, avg_query_time_tree, avg_query_time_naive, memory


def collect_data(n_range):
    """Collect data over a range of interval sizes."""
    results = {'build time': [], 'average query time': [], 'average query time naive
    ': [], 'memory usage': []}
    for n in n_range:
        build_time, avg_query_time, avg_query_time_naive, memory = run_experiment(n)
        results['build time'].append(build_time)
        results['average query time'].append(avg_query_time)
        results['average query time naive'].append(avg_query_time_naive)
        results['memory usage'].append(memory)
    return results


def plot_results(n_range, results):
    """Plot results of the experiment."""
    # Plot for Average Query Times
    plt.figure(figsize=(10, 6))
    plt.plot(n_range, results['average query time'], label='IntervalTree Avg Query
    Time', color='blue')
    plt.plot(n_range, results['average query time naive'], label='Naive Avg Query
    Time', color='red')
    plt.xlabel('Number of Intervals')
    plt.ylabel('Average Query Time (milliseconds)')
    plt.title('Average Query Time Comparison')
    plt.legend()
    plt.grid(True)
    plt.show()

    # Plot for Build Times
    plt.figure(figsize=(10, 6))
    plt.plot(n_range, results['build time'], label='Build Time', color='green')
    plt.xlabel('Number of Intervals')
    plt.ylabel('Build Time (milliseconds)')
    plt.title('Build Time Analysis')
    plt.legend()
    plt.grid(True)
    plt.show()

    # Plot for Memory Usage
    plt.figure(figsize=(10, 6))
```

```
    plt.plot(n_range, results['memory usage'], label='Memory Usage', color='purple')
    plt.xlabel('Number of Intervals')
    plt.ylabel('Memory Usage (bytes)')
    plt.title('Memory Usage Analysis')
    plt.legend()
    plt.grid(True)
    plt.show()


def main():
    n_range = range(10, 100000, 1000)
    results = collect_data(n_range)
    plot_results(n_range, results)


if __name__ == "__main__":
    main()
```