Segment Tree and Its Applications

Prabal Sharma
(2023CSB1145)

Aditya Khajuria
(2023MCB1323)

Abstract

Segment Trees are advanced data structures designed for efficient range queries and updates on arrays and matrices. This report provides a comprehensive study of Segment Trees, including their theoretical foundations, construction, advanced variants such as lazy propagation, and practical applications such as maximum subarray sum and next greater element queries. The extension to two dimensions and real-world significance are also discussed, supported by pseudocode and formal analysis.

Contents

1. Introduction

Overview

Segment Trees are fundamental data structures in computer science, enabling efficient computation of range queries and updates over arrays. Their recursive, tree-based structure allows for the decomposition of complex interval problems into manageable subproblems. Segment Trees are widely used in competitive programming, database query optimization, computational geometry, and real-time systems.

2. Background and Motivation

2.1. The Range Query and Update Problem

Given a sequence of data, it is often necessary to answer queries about subarrays (such as their sum, minimum, maximum, or other associative operations) and to update elements or ranges efficiently. Traditional methods, such as iterating over the range, are computationally expensive for large datasets. Segment Trees address this challenge by supporting both queries and updates in logarithmic time.

2.2. Comparison with Other Data Structures

Data Structure	Range Query Time	Update Time
Segment Tree	$O(\log n)$	$O(\log n)$
Binary Indexed Tree	$O(\log n)$	$O(\log n)$
Array	$O(n)$	$O(1)$
Prefix Sum Array	$O(1)$	$O(n)$

Segment Trees are particularly advantageous when both range queries and updates are frequent and performance-critical.

3. Theoretical Foundations

3.1. Definition and Structure

A Segment Tree is a complete binary tree where each node represents a segment $[l, r]$ of the array. The root node covers the entire array, and each leaf node corresponds to a single element. Internal nodes represent the union of their children's segments. The tree is typically stored as an array of size $4n$ for an input array of size n.

3.2. Properties

- **Height:** $O(\log n)$, as the segment size halves at each level.
- **Number of Nodes:** At most $2n - 1$ for an array of size n.
- **Space Complexity:** $O(n)$.

- **Immutability of Structure:** Only the values stored in the nodes are updated; the structure remains static after construction.

3.3. Recursive Construction

The Segment Tree is constructed using a divide-and-conquer approach. The array is recursively divided into two halves until single elements are reached. The value at each node is computed by merging the values of its children, using an operation appropriate for the problem (e.g., sum, min, max).

4. Basic Segment Tree: Construction and Operations

4.1. Supported Operations

- **Build:** Construct the tree from an initial array.
- **Range Query:** Compute an aggregate value over a subarray $[l, r]$.
- **Point Update:** Update the value at a specific index.

4.2. Algorithmic Details

4.2.1. Build Operation

Listing 1: Segment Tree Build Operation

```
def build(node, l, r):
    if l == r:
        tree[node] = arr[l]
    else:
        mid = (l + r) // 2
        build(2*node, l, mid)
        build(2*node+1, mid+1, r)
        tree[node] = merge(tree[2*node], tree[2*node+1])
```

The build operation constructs the tree in $O(n)$ time, where each internal node's value is computed by merging its children's values.

4.2.2. Range Query

Listing 2: Segment Tree Range Query

```
def query(node, l, r, ql, qr):
    if qr < l or r < ql:
        return neutral
    if ql <= l and r <= qr:
        return tree[node]
    mid = (l + r) // 2
    left = query(2*node, l, mid, ql, qr)
    right = query(2*node+1, mid+1, r, ql, qr)
    return merge(left, right)
```

A range query decomposes the query interval into $O(\log n)$ disjoint segments, each corresponding to a node in the tree.

4.2.3. Point Update

Listing 3: Segment Tree Point Update

```python
def update(node, l, r, idx, val):
    if l == r:
        tree[node] = val
    else:
        mid = (l + r) // 2
        if idx <= mid:
            update(2*node, l, mid, idx, val)
        else:
            update(2*node+1, mid+1, r, idx, val)
        tree[node] = merge(tree[2*node], tree[2*node+1])
```

Point updates are performed in $O(\log n)$ time by updating the relevant leaf and propagating changes up the tree.

4.3. Complexity Analysis

- **Build:** $O(n)$
- **Query:** $O(\log n)$
- **Update:** $O(\log n)$

4.4. Variants and Supported Operations

By changing the merge function, Segment Trees can support:

- **Sum/Product:** Store the sum or product of elements in the segment.
- **Minimum/Maximum:** Store the minimum or maximum value in the segment.
- **GCD/XOR/AND/OR:** Store the result of the respective operation over the segment.

5. Advanced Segment Tree: Lazy Propagation

5.1. Motivation

For frequent range updates (e.g., incrementing all elements in a subarray), the standard Segment Tree becomes inefficient, as each update may affect many nodes. Lazy propagation defers updates and applies them only when necessary, maintaining logarithmic complexity.

5.2. Lazy Propagation Mechanism

Each node maintains a *lazy* value indicating pending updates. When an update is requested, the node is marked as needing an update, and the actual update is propagated to its children only when required.

5.3. Algorithmic Implementation

Listing 4: Lazy Propagation Range Update

```python
def updateRange(node, l, r, ul, ur, val):
    if lazy[node] != 0:
        push(node, l, r)
    if ur < l or r < ul: return
    if ul <= l and r <= ur:
        tree[node] += (r-l+1)*val
        if l != r:
            lazy[2*node] += val
            lazy[2*node+1] += val
        return
    mid = (l + r) // 2
    updateRange(2*node, l, mid, ul, ur, val)
    updateRange(2*node+1, mid+1, r, ul, ur, val)
    tree[node] = tree[2*node] + tree[2*node+1]
```

Listing 5: Lazy Propagation Range Query

```python
def queryRange(node, l, r, ql, qr):
    if lazy[node] != 0:
        push(node, l, r)
    if qr < l or r < ql: return 0
    if ql <= l and r <= qr: return tree[node]
    mid = (l + r) // 2
    left = queryRange(2*node, l, mid, ql, qr)
    right = queryRange(2*node+1, mid+1, r, ql, qr)
    return left + right
```

5.4. Complexity Analysis

- **Build:** $O(n)$
- **Range Update:** $O(\log n)$
- **Range Query:** $O(\log n)$

6. Specialized Segment Trees

6.1. Maximum Subarray Sum (Kadane's Segment Tree)

6.1.1. Problem Statement

Given an array, answer queries for the maximum sum of any contiguous subarray within a specified range, supporting dynamic updates.

6.1.2. Theory

Kadane's algorithm solves the maximum subarray sum problem in $O(n)$ time for static arrays. To support dynamic queries and updates, each node in the Segment Tree stores:

- **Total sum** of the segment
- **Maximum prefix sum**
- **Maximum suffix sum**
- **Maximum subarray sum**

6.1.3. Algorithmic Implementation

Listing 6: Kadane's Segment Tree Merge Function

```
class Node:
    def __init__(self, total, prefix, suffix, max_sum):
        self.total = total
        self.prefix = prefix
        self.suffix = suffix
        self.max_sum = max_sum

def merge(left, right):
    total = left.total + right.total
    prefix = max(left.prefix, left.total + right.prefix)
    suffix = max(right.suffix, right.total + left.suffix)
    max_sum = max(left.max_sum, right.max_sum, left.suffix +
       right.prefix)
    return Node(total, prefix, suffix, max_sum)
```

6.2. Next Greater Element in Range

6.2.1. Problem Statement

For a query range $[l, r]$ and value x, find the smallest element in the range that is greater than or equal to x.

6.2.2. Theory

A merge sort tree is used, where each node stores a sorted list of its segment. Queries use binary search within each relevant node's list, and updates maintain sorted order.

6.2.3. Algorithmic Implementation

Listing 7: Next Greater Element in Range

```
def build(node, l, r):
    if l == r:
        tree[node] = [arr[l]]
    else:
        mid = (l + r) // 2
        build(2*node, l, mid)
        build(2*node+1, mid+1, r)
        tree[node] = merge(tree[2*node], tree[2*node+1])

def query(node, l, r, ql, qr, x):
    if qr < l or r < ql: return INF
    if ql <= l and r <= qr:
        idx = lower_bound(tree[node], x)
        return tree[node][idx] if idx < len(tree[node]) else INF
    mid = (l + r) // 2
    left = query(2*node, l, mid, ql, qr, x)
    right = query(2*node+1, mid+1, r, ql, qr, x)
    return min(left, right)
```

7. 2D Segment Tree

7.1. Theory

A 2D Segment Tree extends the concept to matrices, enabling efficient queries and updates over submatrices. Each node represents a submatrix and contains a segment tree for columns. The structure recursively partitions both rows and columns.

7.2. Algorithmic Implementation

Listing 8: 2D Segment Tree Build (Sketch)

```
def build_x(vx, lx, rx):
    if lx != rx:
        mx = (lx + rx) // 2
        build_x(2*vx, lx, mx)
        build_x(2*vx+1, mx+1, rx)
    build_y(vx, lx, rx, 1, 0, m-1)

def build_y(vx, lx, rx, vy, ly, ry):
    if ly == ry:
        if lx == rx:
            tree[vx][vy] = matrix[lx][ly]
        else:
            tree[vx][vy] = tree[2*vx][vy] + tree[2*vx+1][vy]
    else:
        my = (ly + ry) // 2
        build_y(vx, lx, rx, 2*vy, ly, my)
        build_y(vx, lx, rx, 2*vy+1, my+1, ry)
        tree[vx][vy] = tree[vx][2*vy] + tree[vx][2*vy+1]
```

8. Applications

8.1. Application 1: Maximum Subarray Sum

Problem Statement: Given an array, answer queries for the maximum sum of any contiguous subarray within a specified range, with support for dynamic updates.

Solution: Construct a Kadane's Segment Tree. For each query, merge the relevant nodes to compute the maximum subarray sum in $O(\log n)$ time. Point updates are handled by updating the corresponding leaf and propagating changes upward.

8.2. Application 2: Next Greater Element in Range

Problem Statement: For a query range $[l, r]$ and value x, find the smallest element in the range that is greater than or equal to x.

Solution: Construct a merge sort tree. For each query, traverse the relevant nodes and use binary search to find the answer in each node's sorted list. The final answer is the minimum among all candidates.

8.3. Application 3: 2D Range Queries

Problem Statement: Given a 2D grid, efficiently answer sum/min/max queries over arbitrary submatrices, with support for dynamic updates.

Solution: Build a 2D Segment Tree. For each query, recursively combine results from the relevant submatrices. Updates are handled by modifying the affected cells and propagating changes through both row and column trees.

9. Discussion

Segment Trees and their variants demonstrate significant improvements in computational efficiency for a wide range of range query and update problems. Their recursive structure, modularity, and adaptability to various operations make them indispensable in both theoretical and practical contexts. Lazy propagation and advanced node structures enable the handling of complex operations with minimal overhead. The extension to two dimensions further broadens their applicability to real-world scenarios involving multidimensional data.

10. Conclusion

Segment Trees are a versatile and efficient solution for a wide range of range query and update problems. Their modularity allows adaptation to various operations, and their extensions, such as lazy propagation and 2D trees, address increasingly complex requirements in computational tasks.

11. References

- GeeksforGeeks, "Segment Tree Data Structure", `https://www.geeksforgeeks.org/segment-tree-data-structure/`
- CP-Algorithms, "Segment Tree", `https://cp-algorithms.com/data_structures/segment_tree.html`