A Segment Tree Optimization for Counting Multi-state Collisions

Edward Chen

October 28, 2022

The Segment Tree is a versatile data structure in handling range update and query operations. Querying objects such as integers or objects of the same type has been well studied. However, querying objects that can be one of multiple states requires more complexity. In this paper, we describe an attempt to solve these queries using a unique application of matrices. We prove that updates and queries on a one-dimensional array can be computed in $O(k^3 \log n)$ and can be used in a variety of applications.

1 Introduction

Many problems in Computer Science requires the use of range queries. Basic query operations such as finding the maximum or sum of integers in a 1-dimensional array can be applied to solve problems such as finding the Lowest Common Ancestor(LCA) of two nodes in a tree.

Segment trees are most notable for being able to compute these queries efficiently. The most basic form of a segment tree is a complete binary tree where each node in the tree stores information about an array interval. These intervals can then combined to form all unique combinations of intervals and solve queries in $O(\log n)$ time.

Segment trees are known to be much more versatile compared to similar data structures, such as a Fenwick tree. However, they still require objects in an array to be of identical types. For example, a traditional segment tree can query the sum on an interval of integers, but if cannot query on an interval of different colored blocks where some combination of blocks cannot be placed next to each other, for example. Other queries of these essence involves querying the number of possible combinations if we want to triple count a combination every time a red block is next to a green block or if we want to count combinations where at least one purple block is on the interval edges.

Finding solutions can be used in different topics in Computer Science including but not limited to networking, computer vision, computational geometry databases etc. to name a few.

What follows in this paper is a formal definition of the problem in Section 2 and an outline of previous research in Section 3. Section 4 gives preliminary information on the methods and proof behind a rudimentary segment tree, and our results are shown in Section 5.

2 Problem

2.1 Problem Specification

Given an array of length n, the problem can be formalized to a series of online range queries and index updates. Each array index contains a value $A[i][k]$ describing the number of unique objects at type k that can be at index i. As for interactions between states, these states can either be represented in a local or global 2-dimensional matrix M of size k by k. An index update should mutate A at index i, either updating $A[i]$ entirely or a value in $S[i]$.

Definition 2.1. *Let the collision of index i and index j denote:*

$$C_{i,j} = M_{S_i,S_j}$$

The below results can be applied to a multitude of queries, but for an initial explanation, a range query on array A of the form $query(A[l:r])$ is the sum of $\forall\{S_l, S_{l+1}, ...S_r\} \in \{0, 1, 2, ...k-1\}$ of:

$$\prod_{x=1}^{r} A[S_x] \prod_{x=l}^{r-1} C_{x,x+1}$$

3 Previous work

The segment tree structure was originally discovered by Bentley in 1974 and published in Computational Geometry: Algorithms and Applications. A number of optimizations to them have followed over the decades. Segment trees over multiple dimensions have been optimized by Lei and Xiaodong, for instance [2]. Initially, it might seem that we solve our initial problem with an $n \times k$ segment tree, however, these states must be fixed for each combination.

Although representing node values as matrices seems to naturally follow, to the best of the author's knowledge, no other information has been published about representing node values as matrices. This is most likely because this particular optimization to segment trees solves a niche problem space that could be glanced over.

4 Heap-based Implementation

This section recapitulates the heap-based implementation of the segment tree described in [3]. It allows for a much simpler and computationally less expensive implementation that removes the complexity of recording the structural information in a tree structure.

Definition 4.1. A nearly complete binary tree or a heap can be defined as follows.

1. The depth of a node v in a binary tree is the length (number of edges) of the path from the root to v.
2. The height (or depth) of a binary tree is the maximum depth of any node, or -1 if the tree is empty. Any binary tree can have at most 2^d nodes at depth d.
3. A complete binary tree of height h is a binary tree which contains exactly 2^d nodes at depth d, $0 \leq d \leq h$. In this tree, every node at depth less than h has two children. The nodes at depth h are the leaves. The relationship between n (the number of nodes) and h (the height) is given by $n = 1 + 2 + 2^2 + \ldots + 2^h = 2^{h+1} - 1$, and thus $h = \log(n + 1) - 1$.
4. A nearly complete binary tree of height h is a binary tree of height h in which
   (a) There are 2^d nodes at depth d for $1 \leq d \leq h - 1$
   (b) The nodes at depth h are as far left as possible
   (c) The relationship between the height and number of nodes in a nearly complete binary tree is given by $2^h \leq n \leq 2^{h+1} - 1$, or $h = \lfloor \log n \rfloor$
5. A heap is a nearly complete binary tree T stored in its breadth-first order as an implicit data structure in an array A, where (1) A[1] is the root of T. (2) The left and right child of $A[i]$ are $A[2i]$ and $A[2i + 1]$ respectively. (3) The parent of $A[i]$ is $A[i/2]$. (4) If i is odd then A[i] is a right child of its parent $A[i/2]$, and $A[i - 1]$ is its left sibling. If i is even then $A[i]$ is a left child of its parent $A[i/2]$, and $A[i + 1]$ is its right sibling.

Definition 4.2. *A heap based segment tree T(0, N) is defined as an array tree [1,2,..., 2N - 1] of tree node elements satisfying the following:*

1. The information associated with a tree node v is:
   - v.cnt the number of intervals allocated to node v
   - v.aux augmented data structure
2. The index i of node tree[i] is called its node number, $1 \leq i \leq 2N - 1$.
3. The N leaf nodes corresponding to the N elementary intervals are stored in tree[N..2N-1] in increasing order of their left end point. In other words, the node tree[N +i] corresponds to the elementary interval [i, i + 1], $0 \leq i \leq N - 1$.
4. The parent node of node tree[i] is $tree[\lfloor i/2 \rfloor]$ for all $1 < i \leq 2N - 1$. The node tree[1] is the root of the heap based segment tree. For each non-leaf node i, $1 \leq i < N$, its left and right children are 2i, and 2i + 1 respectively.

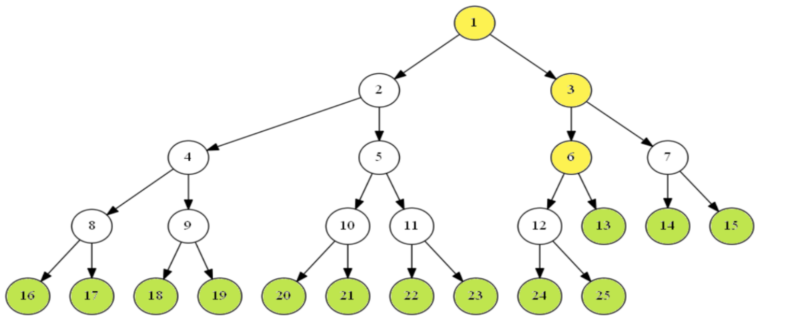

Figure 1: Tree structure of a segment tree where $N = 13$

For example, Figure 1 shows the heap based segment tree $T(0, 13)$. There are three kinds of nodes shown in Figure 1, complete binary tree nodes, nearly complete binary tree nodes and leaf nodes. Those colored in green are leaf nodes, and those colored in yellow are nearly complete binary tree nodes.

5 Results

In this section we present our proposed solution and prove its correctness.

5.1 Transforming the Problem

Definition 5.1. *Given an array of length n, a collision array of length n-1 represents all collisions between adjacent indices.*

Assume we are given the states of each index. Then the range query on the segment is the product of all states and all state interactions. Since the product here is commutative, our range query turns into:

$$\prod_{x=l}^{r} A[S_x] \prod_{x=l}^{r-1} C_{x,x+1} = pre[r+1]/pre[l] \prod_{x=l}^{r-1} C_{x,x+1}$$

Thus, the nature of this problems allows us to frame our problem to that on the collision array. For each index from $i = 0 \to n - 1$, we only require information gained from $C_{i,i+1}$.

Information gained here can be used to construct a segment tree on the collision array with the basic tree structure described Section 4.

This serves us particularly well as matrices effectively serve as compose functions. With matrix multiplication, we have that:

$$mat(l_1, r_2)_{i,k} = \sum_{j=0}^{k-1} mat_{i,j} \cdot mat_{j,k} = \sum_{j=0}^{k-1} mat_{i,j} \cdot mat_{j,k}$$

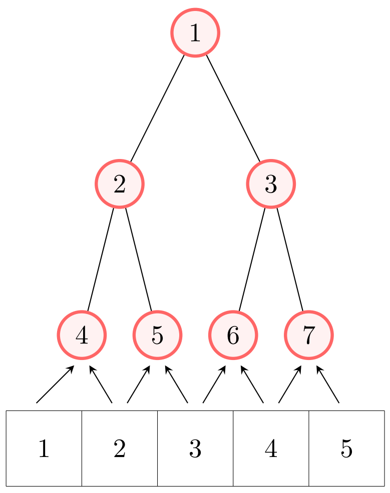

Figure 2: Shown above is an array of length 5, so our segment tree has $N = 4$

To combine intervals is just to guarantee that endpoints are identical states and thus do not overlap. We guarantee that the right-most point on the left interval and the left-most point on the right interval are both j.

5.2 Node Structure and Build

At each of the $2N$ nodes, we store a k by k matrix. Each *node* of our segment tree has:

1. node.left: The left bound on the array interval covered, ranging from 0 to $N - 1$

2. node.right: The right bound, inclusive, on the array interval covered, ranging from 0 to $N - 1$

3. node.matrix: For leaf nodes, all intensive purposes only necessitate the use of a vector of length k, but will be assumed to use a matrix for simplicity. Otherwise, the matrix is the aforementioned given combination matrix of size k by k. The value $node.matrix_{i,j}$ should represent the number of combinations given a state i on the left end and j on the right end.

It is convenient to build these the tree recursively in the other direction, i.e., from the root vertex to the leaf vertices. The construction procedure, if called on a non-leaf vertex, does the following:

1. recursively construct the values of the two child vertices

2. merge the computed values of these children.

In the Algorithm 1, the build function, we are guaranteed to build a nearly-complete or complete binary tree. Given that the floor is taken in division operations among integers by the compiler, $mid - l \geq r - (mid + 1)$ given that $mid = \lfloor \frac{l+r+1}{2} \rfloor$.

Algorithm 1 Build Segment Tree

```
1:  procedure BUILD(i,L,R,ind,new_mat)
2:      if $l = r$ then
3:          $mat[l] \leftarrow A[l].vector * A[R].vector^T$
4:          close;
5:      end if
6:      $mid \leftarrow (l + r + 1)/2$
7:      Build(i*2,L,mid,ind,new_mat)
8:      Build(i*2+1,mid+1,R,ind,new_mat)
9:      $(mat[i]) \leftarrow mat[i * 2] * mat[i * 2 + 1]$
10: end procedure
```

5.3 Update Operations

Updates follow from a normal segment tree. Updates can either be in updating whole vector states or a specific value in a vector.

Algorithm 2 Update node

```
1:  procedure UPDATE(i,L,R,ind,new_mat)
2:      if $l = r$ then
3:          $mat[l] \leftarrow new\_mat$
4:          close;
5:      end if
6:      $mid \leftarrow (l + r + 1)/2$
7:      if $ind \leq mid$ then Update(i*2,L,mid,ind,new_mat)
8:      else Update(i*2+1,mid+1,R,ind,new_mat)
9:      end if
10:     $(mat[i]) \leftarrow mat[i * 2] * mat[i * 2 + 1]$
11: end procedure
```

5.4 Queries

Much like how a vanilla segment tree handles queries, the logic is the same. When querying the bound ql and qr, there are 3 cases:

1. If our current node is fully contained where $ql < l < r < qr$, we return the value of the matrix at our current node
2. If our current node partially contains the query, then propagate query to both children.

3. If our current node is not contained in the query, then return the identity matrix.

It can be proved that the maximum number of segments the query contains is $\lfloor(\log_2(n)\rfloor$.

Algorithm 3 Query Node

```
procedure UPDATE(I,L,R,QL,QR)
    $mid \leftarrow (l + r + 1)/2$
    if $l > r$ then
        return I;
    end if
    if $l \geq qlANDr \leq qr$ then
        return mat[i];
    end if
    return $QUERY(2*i, l, mid, ql, qr) \times QUERY(2*i+1, mid+1, r, ql, qr)$
end procedure
```

5.5 Memory and Time Constraints

Memory and time constraints follow from Bentley's [1] derivations, though combining nodes provides an extra $O(k^2)$ and $O(k^3)$ time and memory factor respectively from matrix multiplication. It is identical from the mere fact that top-down and bottom-up propagation are exactly identical if not for dealing with matrices. Thus, the memory and time upper bounds are $O(k^2N)$ and $O(k^3N \log N)$.

5.6 Optimizing Combining Nodes

Multiplying matrices is potentially very cost-intensive for large k. Implementing the definition for matrix multiplication would grant a time complexity of $O(k^3)$. The time complexity for these operations can be optimized with more time-efficient matrix multiplication algorithms such as Strassen's algorithm which grants $O(k^{log_2 7}) \approx O(k^{2.728})$.

6 Future Directions

There still lies numerous traditional optimizations that can be made to containing matrices in segment tree nodes. Matrix multiplication being non-commutative is currently the largest roadblock to traditional optimizations, but there still might be several solutions. One possibly path is utilizing lazy storage of propagation.

In terms of redefining a more general problem scope, state interactions can be applied to application with multiple dimensions, especially in the 2-D plane.

A sparse segment tree may also be of use given that insertions of new nodes across a fixed dimension will consume large amounts of memory.

References

[1] Mark de Berg, Marc van Kreveld, Mark Overmars, and Otfried Schwarzkopf. Computational geometry. In *Computational geometry*, pages 1–17. Springer, 1997.

[2] Nabil Ibtehaz, M Kaykobad, and M Sohel Rahman. Multidimensional segment trees can do range queries and updates in logarithmic time. *arXiv preprint arXiv:1811.01226*, 2018.

[3] Lei Wang and Xiaodong Wang. A simple and space efficient segment tree implementation. *CoRR*, abs/1807.05356, 2018.