layout	title
post	Understanding Segment Trees

Oh... Segment Trees!

This is a follow up post to my previous article, The Range Minimum Query Problem, where I discussed the Sparse Table solution.

The Sparse Table method may be overkill and prohibitive due to its preprocessing time complexity. It's great that we can reply any query in \(\mathcal{O}(1)\) time, but sometimes, having a preprocessing step taking \(\mathcal{O}(n log(n))\) time can be a showstopper. In such scenarios, using a Segment Tree is the way to go. A segment tree is a perfectly balanced binary tree that can be built in \(\mathcal{O}(n)\) and allows us to answer range queries in \(\mathcal{O}(log(n))\) time.

This article is a wrap up of several resources that I found about segment trees on the web, and attempts to sum it all up in a concise and straightforward way. Time to have some fun!

What is a segment tree?

First and foremost, we will start by explicitly defining the purpose of a segment tree, and when, as well as why, we would ever want to use it. **A Segment Tree is a data structure that represents values associated to specific intervals of an array**. The input to build a segment tree is an array of length \(N\), typically zero-indexed. Let's call this array \(A\). Now, say that you have a binary associative function, \(f\), that works on this array's values. For example, \(f\) may be the function \(min(a,b)\), which returns the minimum between \(a\) and \(b\), where \(a\) and \(b\) are elements inside \(A\). The only requisite is that \(f\) must be associative. A segment tree structures data in such a way that it can tell you the value of \(f\) for any range \([i,j]\) inside \(A\), where \(0 <= i <= j < N\) in \(\mathcal{O}(log(n))\) time. To put it simply, it answers the value of \(f(A[i], A[i+1], A[i+2], ..., A[j])\) in logarithmic time. Actually, since the function is binary and associative, the correct way to write that would be \(f(A[i], f(A[i+1], f(A[i+2], ... f(A[j-1], A[j]))))\), but you get the point.

The purpose of a segment tree is to store the values of \(f(A[i], A[i+1], A[i+2], ..., A[j])\) for some values \([i, j]\), so that we can speed up and ultimately minimize the calls to \(f\). Segment trees can have several applications; basically, we can use them every time we want to evaluate a binary associative function \(f\) in a range. Also, by this time, you will hopefully have figured that the same applies to the Sparse Table method - we don't have to use it only for the Range Minimum Query problem; any binary associative function \(f\) can be used in the Sparse Table method. Consider this a gift for reading it this far, lots of readers have fallen asleep by this point. Interesting binary associative functions include, but are not limited to, `max()`, `min()`, `sum()`, `product()`, and a bunch of others (although for `sum()` you might want to consider a Binary Indexed Tree.

So, as you can imagine, there's lots of applications for Segment Trees. Since Segment Trees can be used to solve the Range Minimum Query Problem, they can also be used to solve the Lowest Common Ancestor problem, which I will get into in my next post. In fact, any method that solves the RMQ problem also solves the LCA problem - oh, isn't *that* beautiful? The power of Computer Science...

How is a Segment Tree structured?

The idea is simple. Consider that a node n stores the value of f for an interval $[i,j]$. Then, the left child, l, will store the value of f for $[i, (i+j)/2]$, and the right child, r, will store the value of f for the interval $[(i+j)/2+1, j]$. We essentially break the interval in two halves, and store information for each of those halves in the children. The root node matches the entire array, that is, it corresponds to the interval $[0..N-1]$. The left child of the root matches the interval $[0, (N-1)/2]$, and right child matches $[(N-1)/2+1, N-1]$. This will keep happening until we reach a point where intervals are one element only - this happens on leaf nodes.

So, to put it all together: leaf nodes correspond to one-element only intervals. In an array of size N, we will have N leaf nodes: one for the interval $[0, 0]$, one for $[1, 1]$, one for $[2, 2]$, etc. The parent of $[0, 0]$ and $[1, 1]$ will store $[0, 1]$. In a bottom up approach, we can visualize the tree by combining every pair of intervals to form a new parent node.

For example, let's say we have an array of length 9. Then (part of) the corresponding segment tree can be visualized as follows:

One important and interesting point to keep in mind is that we can build a segment tree either in a top-down approach, where building the root causes 2 recursive calls to build its children, which in turn will cause 4 recursive calls, etc, or in a bottom-up approach, where we start by building every leaf node, and then we combine each pair to build the next layer, just as I mentioned before. Both forms are equivalent, and none of them is necessarily harder to implement than the other. I will go with top-down implementation on this post - bottom-up will be left as an exercise for the astute reader.

How a segment tree is built

The fact that f is a binary associative function makes the task of building a segment tree easier than it looks: it just so happens that

$$ \begin{split} f(A[i], &A[i+1], ..., A[j]) = \\ &f(f(A[i], A[i+1], ..., A[k]), f(A[k+1], A[k+2], ..., A[j])) \end{split} $$

Where $k = (i+j)/2$. To build node n, we need to build its left and right children, l and r, and then store $f(l, r)$. This process stops when i and j are equal - the leaf nodes.

Let's look at an example. Consider that \(A = [-2, 5, 3, 0, -1, 4]\) and \(f\) is the \(min()\) function. The root node will end up storing the minimum of the interval \([0, 5]\), i.e., the minimum in the whole array. To compute such a value, first, it recursively builds the left child, which stores the minimum for \([0, 2]\). Then it builds the right child, which stores the minimum for \([3, 5]\). Clearly, the minimum in the whole interval \([0, 5]\) is the smaller of the minimum of \([0, 2]\) and the minimum of \([3, 5]\). And there you go - of course it's not that simple, because building each of the children will end up doing more recursive calls, until we get to the base case of a one-element interval, but the unwinding procedure is basically what I described above.

Building a segment tree - implementation

I will provide a basic C implementation for building a segment tree. An easy and common approach to representing balanced binary tree structures is to use an array. In a zero-indexed array, the left and right children of node \(i\) are given by \(i*2+1\) and *\(i*2+2\). I will make use of this technique in my code samples.

My implementation is not the most elegant and it is far from being production-quality code. The point is to illustrate the algorithm details. Ideally, we would want to make this a shiny pretty C++ class to store a pointer to \(f\), and we would want to store the interval endpoints that each node is responsible for, instead of passing them around in the recursive call. Depending on the problem constraints, we should consider using an explicit left and right pointer, dynamically allocating each node. There's much more to say about implementation details, but I am here to write about the algorithm, not about software engineering rules. Thus, for the sake of illustration, I'll just implement the tree using a statically allocated array. A relatively simple, yet powerful, improvement, is to convert the code to C++ and implement the tree using a vector instead. Since vectors grow as needed, we don't have to worry about overflowing the array. Again, this small exercise is left for the reader.

```
#define SEGTREE_MAX_NODES 1000

static int segtree[SEGTREE_MAX_NODES];

static void build_segtree_aux(int a[], size_t pos, size_t start, size_t end, int (*f)(int, int))
{
    if (start == end) {
        segtree[pos] = f(a[start], a[start]);
        return;
    }

    size_t mid = start+(end-start)/2;
    size_t left = pos*2+1;
    size_t right = left+1;

    build_segtree_aux(a, left, start, mid, f);
    build_segtree_aux(a, right, mid+1, end, f);

    segtree[pos] = f(segtree[left], segtree[right]);
}

/* Assumes len > 0 */
void build_segtree(int a[], size_t len, int (*f)(int, int))
{
    build_segtree_aux(a, 0, 0, len-1, f);
}
```

User code should call `build_segtree()` to kick out the building process. This function receives the input array, its size, and a pointer to the binary associative function \(f\). The construction algorithm itself is implemented in `build_segtree_aux()`. The auxiliary function receives the array \(A\), so that it can read it when needed; the position of the current node in the underlying array that represents the segment tree; the start and end of the interval for the current node being built; and a pointer to \(f\), so that we can call it throughout the building process. The algorithm is short and concise; it implements a basic top-down segment tree construction procedure.

Querying a Segment Tree

We shall now turn to the problem of answering queries for arbitrary intervals. The challenge is to use the predefined set of intervals that we have to compute \(f\) for arbitrary interval ranges. Let's get back at our previous example for an array \(A\) of 9 elements. Say we want to query the tree for the interval \([1, 5]\). How do we do that?

As it turns out, we have to "build" this interval by merging a bunch of our predefined intervals. Simply put, we want to find the smallest set of nodes for which their union represents the desired interval. For \([1, 5]\), we would need to look at nodes \([1, 1]\), \([2, 2]\), \([3, 4]\), and \([5, 5]\). Do notice that any valid interval can be built by bringing together one or more nodes of a segment tree. This might not be intuitive at first. It's just a matter of finding the right nodes to bring together. For example, what if we wanted \([1, 3]\)? Then we'd need nodes \([1, 1]\), \([2, 2]\) and \([3, 3]\). What if we wanted \([0, 8]\)? Then we just need to look at the root. And \([1, 8]\)? We would need nodes \([1, 1]\), \([2, 2]\), \([3, 4]\), and \([5, 8]\).

How do we do this algorithmically? After doing it visually for a while, the steps to follow become clear: at any given node, we test the left child and right child intervals against the input interval. If both of the children's intervals intersect with the input interval, we recurse on both sides of the tree, adjusting the input interval endpoints accordingly for each recursive call. If only the left child intersects with the input interval, we just recurse on the left side. Otherwise, we recurse on the right side.

The base case is when we have an exact match between the interval that the current node represents and the input interval. This is where things stop - possibly at a leaf node.

Nothing speaks more for itself than the code:

```
/* Returns 1 if intervals [i, j] and [p, k] intersect; 0 otherwise */
static int intervals_intersect(size_t i, size_t j, size_t p, size_t k)
{
    return j > k ? i <= k : p <= j;
}

static int query_segtree_aux(size_t pos, size_t i, size_t j,
                 size_t start, size_t end, int (*f)(int, int))
{
    if (i == start && j == end)
        return segtree[pos];

    size_t mid = i+(j-i)/2;

    int recurse_left = intervals_intersect(i, mid, start, end);
    int recurse_right = intervals_intersect(mid+1, j, start, end);
    if (recurse_left && recurse_right)
        return f(query_segtree_aux(pos*2+1, i, mid, start, mid, f),
                 query_segtree_aux(pos*2+2, mid+1, j, mid+1, end, f));
    else if (recurse_left)
        return query_segtree_aux(pos*2+1, i, mid, start, end, f);
    else
        return query_segtree_aux(pos*2+2, mid+1, j, start, end, f);

}

int query_segtree(size_t a_len, size_t start, size_t end, int (*f)(int, int))
{
    return query_segtree_aux(0, 0, a_len-1, start, end, f);
}
```

User code calls `query_segtree()` when a query comes in. It receives the input array's length, the endpoints of the interval being queried, and the function \(f\). It then passes this information to `query_segtree_aux()`. This auxiliary function receives the current node being visited, the endpoints of the current node's interval, the endpoints of the input interval, and, of course, a pointer to \(f\). The code is high in brevity but somewhat low on intuition; in particular, it is very easy to mess up the endpoints in the recursive calls. Notice that the input interval changes if we recurse on both sides - this makes sense, since we have to adjust the input interval's endpoints to the largest possible endpoint to the left and to the right. But this doesn't happen when we recurse only on one side: after all, if we only dive into one direction, it means that the input interval is contained in that node's interval, so we clearly cannot change the input endpoints. Otherwise, we would end up querying an interval bigger than the desired interval - *poof*. Other than that, the algorithm is not something very complicated. The querying process is naturally recursive, there's just not much more to tell.

Usage

A typical usage might be to build a segment tree to solve the range minimum query problem. In that case we would want to define \(f\) to be \(min()\):

```
static int f_min(int x, int y)
{
    return x < y ? x : y;
}
```

Then, we simply call `build_segtree()`, and we're ready to roll - any subsequent query can be answered in \(\mathcal{O}(log(N))\) time. The cost of preprocessing is linear on the input size (think about the leaf nodes - we end up calculating \(f\) for every single element of \(A\)).

As for space complexity, a tree for \(N\) elements will have at most \(2^{\lceil log(N) \rceil + 1}\) elements.

Updates

Unlike the Sparse Table method, segment trees support updating, that is, the values of \(A\) can be changed without much impact.

Bringing the tree to a consistent state after updating \(A[i]\) is a process similar to what we do during construction phase. We can either do it bottom-up or top-down. In a bottom-up approach, we would start by updating the value in the corresponding leaf node to the new value of \(f(A[i], A[i])\). Then, we would update each node along the path up to the root, by recalculating \(f\). In a top-down approach, we start on the root, and at each step we decide to recurse left or right, depending on where the node for \([i, i]\) is. As the recursion unwinds, we recalculate \(f\) in each node along the path:

```
static void update_segtree_aux(size_t pos, int a[], size_t i, size_t start,
                               size_t end, int (*f)(int, int))
{
    if (start == end) {
        /* i == start && start == end */
        segtree[pos] = f(a[i], a[i]);
        return;
    }

    size_t mid = start+(end-start)/2;
    if (i <= mid)
        update_segtree_aux(pos*2+1, a, i, start, mid, f);
    else
        update_segtree_aux(pos*2+2, a, i, mid+1, end, f);

    segtree[pos] = f(segtree[pos*2+1], segtree[pos*2+2]);
}

void update_segtree(int a[], size_t i, size_t len, int (*f)(int, int))
{
    update_segtree_aux(0, a, i, 0, len-1, f);
}
```

Again, the function that is exported to user code is `update_segtree()`. It receives the array \(A\), the position that was just changed, the array's length, and a pointer to \(f\). The actual work is done in `update_segtree_aux()`, which receives the current node being scanned, the array, the position in the array that was changed, the endpoints of the interval for the current node, and a pointer to \(f\). The algorithm recurses down the tree, finding the path to the leaf responsible for the interval \([i, i]\), and updating every node in the path as the recursion unwinds.

Since Segment Trees are fully balanced binary trees, the cost of an update is, unsurprisingly, \(\mathcal{O}(log(N))\).

Closing thoughts

Segment Trees are a good alternative to the Sparse Table method for solving the Range Minimum Query problem. They are particularly useful for situations where we want to support updates to the original array. The preprocessing time for building a segment tree is asymptotically lower than the time for Sparse Table method, but this comes at a cost of \(\mathcal{O}(log(N))\) time for queries, as opposed to \(\mathcal{O}(1)\) with Sparse Table.

The sample implementations showed here are far from being perfect. In particular, we think that good OO design will reduce the amount of parameters being passed in the recursion.

Both methods can be used with any binary associative function, which means that the range minimum query is just an example.

There is much more to explore.

In the next article, we will be looking at the lowest common ancestor problem, its applications, and how it can be reduced to the range minimum query problem. This implies that we can actually answer any LCA query in \(\mathcal{O}(1)\), which is wonderful.

Have fun with segment trees.