Sweepline Algorithm for Line Segment Intersections

Víctor Franco Sanchez

2022

Abstract
In this paper we implement a visualization for the sweepline algorithm for finding all the line segment intersections, together with all the required data structures for the algorithm, and discuss the efficiency of this alternative over the trivial algorithm.

1 Introduction

The problem of line segment intersection is the problem of, given a set of line segments, return all the points at which these line segments intersect. For this document, we'll assume no two line segments are colinear. A trivial algorithm for solving this problem is the following:

```
def bruteforce(linesegments):
    for i in 1 ... len( linesegments ):
        for j in (i+1) ... len( linesegments ):
            if intersects(linesegments[i], linesegments[j]):
                report intersection
```

This algorithm quite simply iterates over every pair of line segments and reports every intersection found. Although silly, this algorithm is already asymptotically optimal, since this algorithm is $O(n^2)$, and the total number of intersections can end up being $O(n^2)$ as well, and since we need constant time to output every intersection, no algorithm can do better in the worst case. Then, how can we do better? The answer is: An output sensitive algorithm. That is, an algorithm that does better for simpler outputs and worse for more complicated outputs, so if the number of intersections is $O(n)$ instead of $O(n^2)$, we can therefore get an algorithm with a better runtime than $O(n^2)$.

2 Sweepline Algorithm

The sweepline algorithm looks as follows:

```
def sweepline(linesegments):
    DS <- new Dictionary
    PQ <- new PriorityQueue
    for ls in linesegments:
        PQ.add( start(ls.startpoint) )
        PQ.add( end(ls.endpoint) )
    while PQ is not empty:
        top <- PQ.pop()
        if top = start(point):
            DS.add( point.linesegment )
            n1, n2 <- DS.neighbours(point.linesegment)
            x1 <- intersection_after(point.x, n1, point.linesegment)
            x2 <- intersection_after(point.x, n2, point.linesegment)
            if some of the two exists:
                PQ.add( cross(earliest(x1, x2)) )
        if top = end(point):
            n1, n2 <- DS.neighbours(point.linesegment)
            DS.remove(point.linesegment)
            if intersects_after(point.x, n1, n2):
                x <- intersection(n1, n2)
                if cross(x) is not in PQ:
                    PQ.add( cross(x) )
        if top = cross(point):
            report intersection at point
            lt <- point.toplinesegment
            lb <- point.botlinesegment
            ltt <- DS.topneighbour(lt)
            lbb <- DS.botneighbour(lb)
            DS.swap(lt, lb)
            if intersects_after(point.x, lb, ltt):
                x <- intersection(lb, ltt)
                if cross(x) is not in PQ:
                    PQ.add( cross(x) )
            if intersects_after(point.x, lt, lbb):
                x <- intersection(lt, lbb)
                if cross(x) is not in PQ:
                    PQ.add( cross(x) )
```

Inserting into a priority queue takes $O(\log |PQ|)$ time, this is done once per iteration, and the number of elements in the queue cannot exceed $2n + m$, being n the number of elements and m the number of intersections, which is at most $O(n^2)$, meaning the cost of inserting into the priority queue is $O(\log n^2) = O(\log n)$. Every operation regarding the dictionary takes $O(\log |DS|)$, and the dictionary stores line segments, meaning it can have at most n elements and thus the cost of every operation is $O(\log n)$. The while loop runs for $2n + m$ iterations, doing a constant number of calls to both structures each iteration,

meaning the final cost is $O((n+m)\log n)$. If $m = O(n)$, the final cost would be just $O(n \log n)$, which is better than $O(n^2)$. However, the algorithm is worse for some large values of m, as if $m = O(n^2)$, then the cost of the algorithm is $O(n^2 \log n)$, which is even worse than the original brute force algorithm. On a latter section we discuss how to improve upon this. For now, let's focus on how to implement the required data structures to make this work.

2.1 Priority queue

The priority queue was implemented on top of the array prototype for JavaScript. The heapify function is an auxiliary function that sinks the element at a particular position down to the desired place. This function takes $O(\log n)$ time worst case.

```
Array.prototype.heapify = function(m, i, cfunction = (a, b) => a - b){
        let largest = i;
        let l = (i << 1) + 1, r = (i << 1) + 2;
        if(l < m && cfunction(this[l], this[largest]) < 0)
                largest = l;
        if(r < m && cfunction(this[r], this[largest]) < 0)
                largest = r;
        if(largest !== i){
                [ this[i], this[largest] ] = [ this[largest], this[i] ];
                this.heapify(m, largest, cfunction);
        }
}
```

The addWithPriority function serves as a push in a normal heap. It adds the element at the end of the array, and then floats the element up to the correct place. Its runtime is $O(\log n)$.

```
Array.prototype.addWithPriority = function(x, cfunction = (a, b) => a - b){
        let dad = (k) => (k - 1) >> 1;
        this.push(x);
        let i = this.length - 1;
        while( i > 0 && cfunction( this[dad(i)], this[i] ) > 0 ){
                [ this[dad(i)], this[i] ] = [ this[i], this[dad(i)] ];
                i = dad(i);
        }
}
```

The popWithPriority method serves as a pop in a normal heap. It removes the element at the top, gets the element at the bottom and places it at the top, and then sinks the element down using the heapify function. Everything but the heapify call is constant, therefore having a cost of $O(\log n)$.

```
Array.prototype.popWithPriority = function(cfunction = (a, b) => a - b){
        let ret = this[0];
        this[0] = this[this.length - 1];
        this.pop();
        this.heapify(this.length, 0, cfunction);
        return ret;
}
```

The last implemented function is the makeHeap function, which uses the heapify function to turn an array into a heap in $O(n)$ time. The proof for this is cost not trivial.

```
Array.prototype.makeHeap = function( cfunction = (a, b) => a - b ){
        for(let i = (this.length >> 1) - 1; i >= 0; --i){
                this.heapify( this.length, i, cfunction);
        }
}
```

2.2 Dictionary (Treap)

The data structure implemented for the dictionary was a treap. On a treap, as in a BST, every node has a value, a left child, a right child and (in this implementation) a parent child, but together with that, every node in a treap also has a priority. Treaps must meet the following invariants:

- Given a node, its value its greater than any value on the left subtree, and lesser than any value on the right subtree.
- Given a node, its priority is greater than the priority of any of the two children.

These two properties, if priorities are chosen at random, guarantee that the treaps have the same depth as random BSTs (assuming the precision for the priority is enough) all while doing a minimal number of calls to the random number generator.

```
{priority: Math.random(), value: ls, l: null, r: null, p: null}
```

The data structure per se only contains a pointer to the first node of the tree and a counter of how many elements there are in the tree.

```
class LineSegmentDictionary{
    constructor(){
        this.tree = null;
        this.size = 0;
    }
    //...
}
```

Also, for this data structure to work, it's useful to implement two functions caller "rotations" which, in short, will change a node with one of its children all while keeping the first invariant intact. This is useful for balancing the tree.

Due to a small misunderstanding, in this implementation "lrot" refers to what normally in the literature is called a "right rotation" and "rrot" refers to what normally in the literature is called a "left rotation". On the literature, "left" or "right" refer to the direction in which the child moves, while in my implementation "l" or "r" refer to which child you're putting on top. Regardless, the idea is the same.

```
lrot(x) {
    let y = x.l, z = y.r;
    y.r = x;
    x.l = z;
    y.p = x.p;
    x.p = y;
    if(z !== null) z.p = x;
    return y;
}
rrot(y){
    let x = y.r, z = x.l;
    x.l = y;
    y.r = z;
    x.p = y.p;
    y.p = x;
    if(z !== null) z.p = y;
    return x;
}
```

There are four operations to implement: Insert, Delete, Swap and Find Neighbours.

The Insert operation receives a line segment and stores the segment somewhere on the tree such that the line segment follows the first invariant according to the y coordinate of the line segment at the x coordinate of the sweepline (this is what the "compare" function does), and the second invariant is met according to some randomly-generated value for the priority. First, ensures the first invariant by traversing the tree just as you would traverse a BST, and inserts the element at the end, but to keep the second invariant begins doing tree rotations upwards, similar to the float operation on a heap. Finally, the insert operation returns a pointer to the node inserted. This is useful for efficiency during the algorithm, since you don't need to find the node afterwards, but it's not required, and was certainly a bad idea, since this is the reason for why we store the parent of every node, which was a headache to keep track of, together with some other complications mentioned later on.

Regardless, the insert function is quite intuitive once you understand the reasoning behind it.

```
insert(ls, nptr = new Object, node = this.tree){
    if(node === null) {
        node = {priority: Math.random(),
                value: ls,
                l: null,
                r: null,
                p: null};
        nptr.ptr = node;
        this.size++;
        if(this.tree === null) this.tree = node;
        return node;
    }
    let n = node;
    if( compare( n.value, ls ) > 0 ){
        n.l = this.insert(ls, nptr, n.l);
        n.l.p = n;
        if(n.l.priority > n.priority) {
            let par = n.p;
            n = this.lrot(n);
            if(par !== null && par.l === node) par.l = n;
            if(par !== null && par.r === node) par.r = n;
        }
    } else if( compare( n.value, ls ) < 0 ) {
        n.r = this.insert(ls, nptr, n.r);
        n.r.p = n;
        if(n.r.priority > n.priority) {
            let par = n.p;
            n = this.rrot(n);
            if(par !== null && par.l === node) par.l = n;
            if(par !== null && par.r === node) par.r = n;
        }
    }
    if(node === this.tree) this.tree = n;
    return n;
}
```

The delete method receives a node and removes it. The node was obtained throughout the insert procedure described above. The idea is also rather simple, but one must be careful with all the cases. The idea is: If one node is missing, replace the node with the child that's not missing, otherwise, do an appropriate rotation to sink the node further down (by setting as a parent the child node with greatest priority) and try again. It's obvious that this method will terminate in $O(h)$ steps, being h the depth of the tree, since the node is getting dragged down the tree once at every step.

```
delete(node, first=true){
    let root = node;
    if(root.l === null) {
        if(root.p !== null && root.p.r === root) root.p.r = root.r;
        if(root.p !== null && root.p.l === root) root.p.l = root.r;
        if(root.r !== null) root.r.p = root.p;
        root = root.r;
        this.size--;
    } else if(root.r === null) {
        if(root.p !== null && root.p.r === root) root.p.r = root.l;
        if(root.p !== null && root.p.l === root) root.p.l = root.l;
        if(root.l !== null) root.l.p = root.p;
        root = root.l;
        this.size--;
    } else if(root.l.priority > root.r.priority){
        let par = node.p;
        root = this.lrot(root);
        if(par !== null && par.l === node) par.l = root;
        if(par !== null && par.r === node) par.r = root;
        this.delete(root.r, false);
    } else {
        let par = node.p;
        root = this.rrot(root);
        if(par !== null && par.l === node) par.l = root;
        if(par !== null && par.r === node) par.r = root;
        this.delete(root.l, false);
    }
    if(first && node === this.tree) this.tree = root;
    return root;
}
```

The swap operation just switches the position of two elements in the tree. This should have been simple, but because outside of the data structure we have pointers pointing to nodes inside of the structure, it's not enough to switch the values of the nodes, but instead we need to change *everything else*, that is, the parent pointer, the children pointers, the parent pointer of the children, the child pointer of the parent and the priority. Otherwise, we would have a pointer from outside that used to point to a line segment that now points to another. Thanks to this, this operation goes from potentially $O(\log n)$ to $O(1)$ time, but no cost analysis will ever come close to quantify the cost of my loss of sanity during the implementation of this function.

```
swap(node0, node1){
    let p0 = node0.p, l0 = node0.l, r0 = node0.r;
    let p1 = node1.p, l1 = node1.l, r1 = node1.r;
    node1.p = p0 === node1 ? node0 : p0;
    node1.l = l0 === node1 ? node0 : l0;
    node1.r = r0 === node1 ? node0 : r0;
    node0.p = p1 === node0 ? node1 : p1;
    node0.l = l1 === node0 ? node1 : l1;
    node0.r = r1 === node0 ? node1 : r1;
    if(l0 !== null && l0 !== node1) l0.p = node1;
    if(r0 !== null && r0 !== node1) r0.p = node1;
    if(l1 !== null && l1 !== node0) l1.p = node0;
    if(r1 !== null && r1 !== node0) r1.p = node0;
    if(p0 !== null && p0 !== node1 && p0.l === node0) p0.l = node1;
    if(p0 !== null && p0 !== node1 && p0.r === node0) p0.r = node1;
    if(p1 !== null && p1 !== node0 && p1.l === node1) p1.l = node0;
    if(p1 !== null && p1 !== node0 && p1.r === node1) p1.r = node0;
    let tmp = node0.priority;
    node0.priority = node1.priority;
    node1.priority = tmp;
    if(node0 === this.tree) this.tree = node1;
    else if(node1 === this.tree) this.tree = node0;
}
```

Finally, the find neighbours operation is rather straight fordward, and is done the same as you would do a find operation in a BST. Suppose you want to find the element to the left of a node: First, you begin from the root of the tree, and you remember the node furthest to the right that's to the left of the node, and once you reach the node, move to the left node, then find the rightmost node from there, and compare the last node with the closest node you found from above, and return the closest one. If you want to find the element to the right of a node, just change left and right on the previous description.

2.3 Introspection

Consider the following, arguably rather silly algorithm:

```
def IntrospectiveLineIntersect(linesegments):
    run the line sweep algorithm on linesegments for k*n*n steps
    if it finished before the established time limit:
        return the result
    otherwise:
        return bruteforce(linesegments)
```

Where $k > 0$ is some fixed constant.

This algorithm now has a cost of $O(min\{n^2, (n+m)\log n\})$, asymptotically getting the best of both worlds. Some further improvements can be done, such

as remembering which line segments went through the "end" part of the line sweep and just skip those during the bruteforce algorithm, since those points will have already been reported by the other algorithm. Of course, there's some fine tuning with the k constant if one wishes to get the best results, but other than that this algorithm is sound.

This algorithm was not implemented, for the main objective of this project was the implementation of the sweepline algorithm, but I still feel it's worth mentioning the existence of this algorithm.

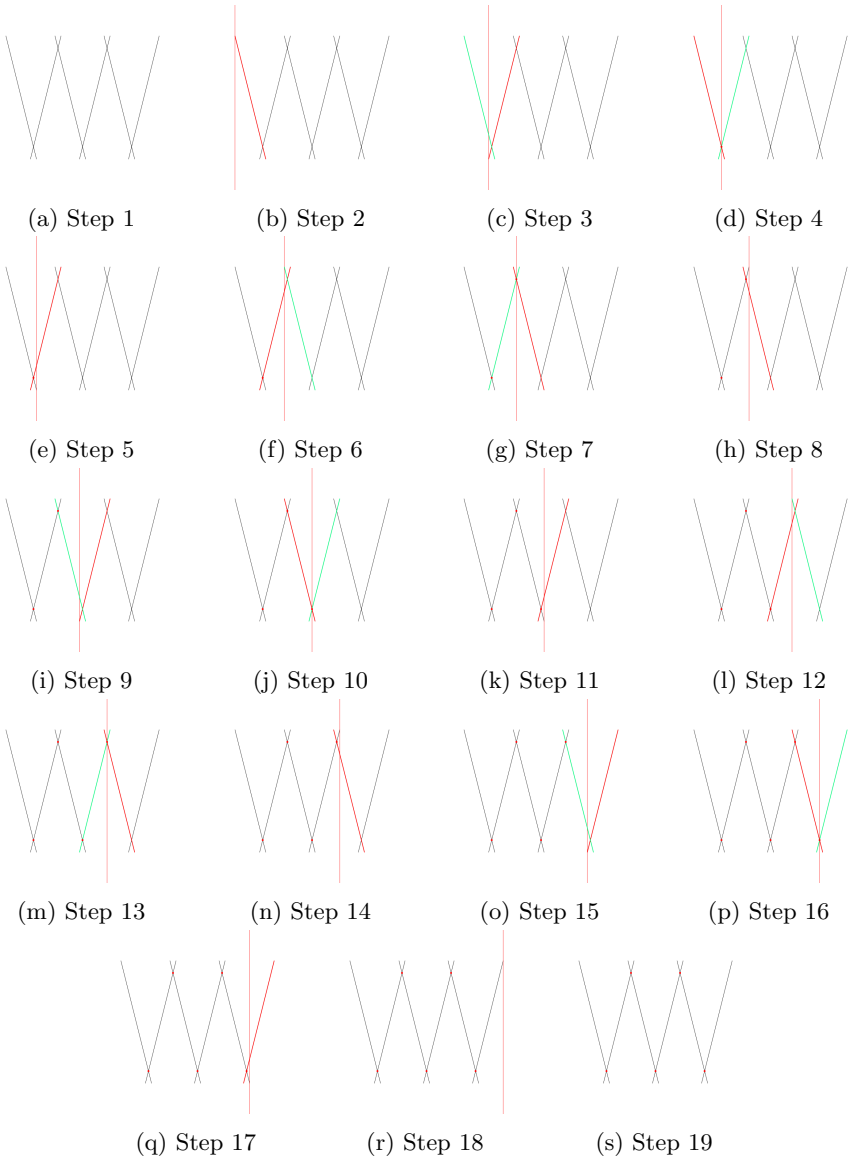
3 Results

The program can be found in the GitHub link at the bottom of this document. The visualization for the algorithm shows a thin vertical line that represents the sweepline, and represents the line segments in the dictionary as thicker colored line segments. The hue represents the position in the dictionary: Redder means more at the bottom, more purple means more at the top.

Figure 1: Hue spectrum. The more to the right on this spectrum the hue of a line segment is, the higher it is stored on the dictionary.

On the first example (Figure 2), we can see an example of a line-segment intersection problem with a linear number of intersections (or, at least, it would be if the pattern was further generalized). One key observation is that, for this example, at every moment there are at most two line segments in the dictionary, meaning the operations on the data structure are $O(1)$.

Also quite importantly, notice how at every intersection (Steps 4, 7, 10, 13 and 16) there is a swap in the dictionary, and the line segment that used to be above in the dictionary is now below and vice versa. Because there are $O(n)$ intersections, the number of iterations is $O(n)$, and even though the operations on the dictionary are $O(1)$, the operations on the priority queue are still $O(\log n)$, and thus the final cost of the algorithm for this generalized family of problems is $O(n \log n)$.



(a) Step 1 (b) Step 2 (c) Step 3 (d) Step 4

(e) Step 5 (f) Step 6 (g) Step 7 (h) Step 8

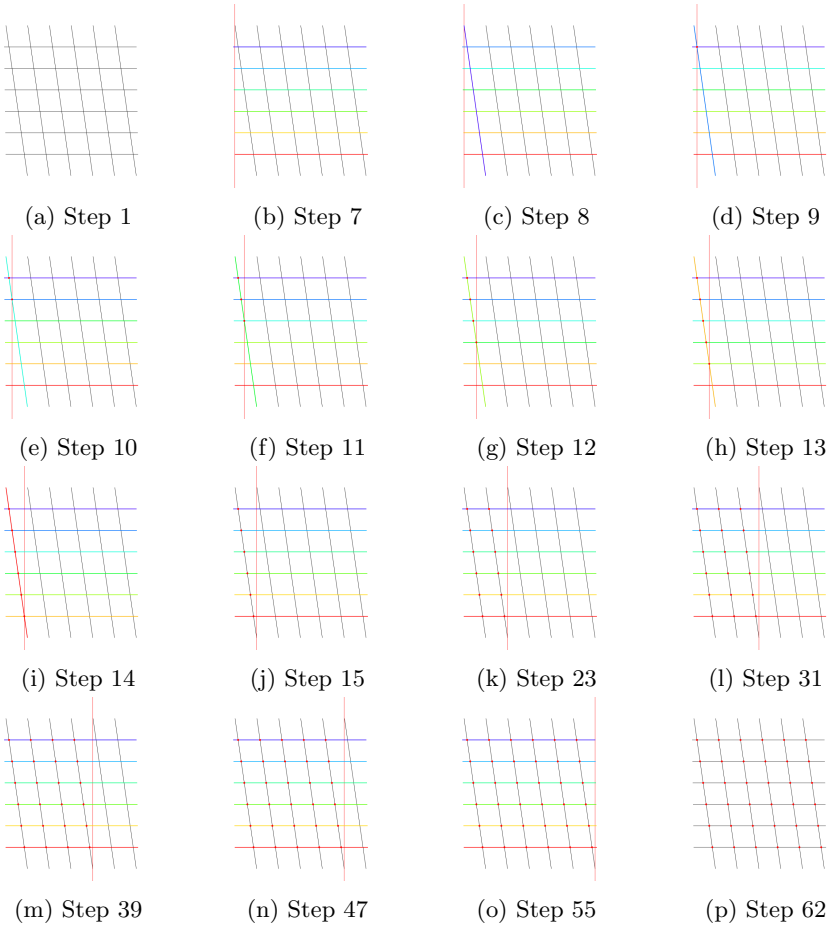
(i) Step 9 (j) Step 10 (k) Step 11 (l) Step 12

(m) Step 13 (n) Step 14 (o) Step 15 (p) Step 16

(q) Step 17 (r) Step 18 (s) Step 19

Figure 2: The sweepline algorithm running over an example with a linear number of intersections. Steps 2, 3, 6, 9, 12 and 15 are "start" operations, steps 4, 7, 10, 13 and 16 are "cross" operations, and steps 5, 8, 11, 14, 17 and 18 are "end" operations.

We can also construct an example in which the number of intersections is quadratic with respect to the number of line segments if we split the segments half and half and put them in a grid pattern (Figure 3). The number of intersections would be $n^2/4 = O(n^2)$, meaning the runtime of the algorithm for this family of problems is $O(n^2 \log n)$, asymptotically worse than the brute force method.



(a) Step 1 (b) Step 7 (c) Step 8 (d) Step 9

(e) Step 10 (f) Step 11 (g) Step 12 (h) Step 13

(i) Step 14 (j) Step 15 (k) Step 23 (l) Step 31

(m) Step 39 (n) Step 47 (o) Step 55 (p) Step 62

Figure 3: The sweepline algorithm running over an example with a quadratic number of intersections.

4 Conclusions

The sweepline algorithm for this problem provides a situational improvement over the brute force algorithm. This algorithm is interesting if you know the line segments on the data you'll be working with will be generally disperse and therefore it's unlikely to get a quadratic explosion on the number of intersections. If you don't have such guarantee, you will be better off with the quadratic algorithm, and if you only have this guarantee *sometimes*, you might want to consider an introspective algorithm.

5 Source Code

The source code for this project can be found in the following link:
https://github.com/hectobreak/Line-Segment-Intersection-Line-Sweep

If I'm being honest, I recommend you to check it out for yourself, it's actually quite fun to play around with it if I do say so myself.