Sweep Line Java 11 implementation
Computational Geometry

Luis Fernando Yang Fong Baeza
fernandofong@ciencias.unam.mx

April 19th, 2020

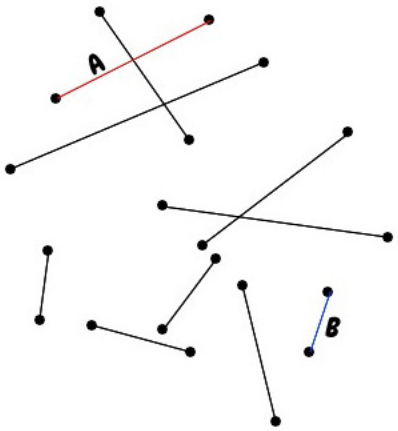
1. Introduction

On this brief document, I will try to explain my reasoning on why implementing this algorithm in this language is optimal, even though there are other implementations written in C++ and it may be faster, this implementation is not intended to be used in programming competitions, due to all of its classes and abstracts classes, considering that you can only send one file in competitive programming competitions, this code is intended to be used for Java applications only, either Android, Kotlin or else.

2. Algorithm

The idea is quite simple, if anyone is not familiarized with Sweep Line Algorithm you MUST NOT skip this part, if you know what it does, its input and output, then you can go to the implementation.

The algorithm receives a set of n segments let's say S, and we wish to calculate all the intersections of the segments in S. We could run an $O(n^2)$ trivial algorithm, with $n = |S|$ but consider the case where you only have 1 or 2 intersections and over $O(10k)$ segments, spend $O(10k^2)$ operations for 2 intersections sounds like a waste of time, this algorithm optimizes that issue, so, we need to find a way to check only smart segments, there is no sense in checking if the marked segments on the next figure intersect.



You can find if two segments intersect with the next mathematical reasoning, you have two segments, $A = (P_1, P_2)$ and $B = (P_3, P_4)$, where P_i is a point in $\mathbb{F}^2$, not necessarily $\mathbb{R}$. Each segment is determined by a grade one polynomial, that is an equation of the form $y = mx + b$, where x needs to be in the range of the according points that make the segment, so we can have two equations, let's say $y_1 = m_1x + b_1$ and $y_2 = m_2x + b_2$, so the first point of this reasoning is that if $m_1 = m_2$ then the segments are parallel, thus they don't intersect, suppose that $m_1 \neq m_2$, we can find the m_i using the slope equation, that is:

$$m = \frac{y_2 - y_1}{x_2 - x_1}$$

and with a little bit of algebra, anyone can see that $b_i = y - m_ix$ once m_i has been found and with a known y which can be one of the coordinates of the point and so if the segments intersect, we just need to solve the equation:

$$m_1x + b_1 = m_2x + b_2$$

On which, also with a little bit of algebra anyone can find the value of x, by doing:

$$x = \frac{b_2 - b_1}{m_1 - m_2}$$

Checking if that x is in range of the segment and if it is, then evaluate in either equation and *presto*, in constant time you can find the intersection of two segments, so with this said, we now may proceed.

So, the sweep line algorithm, can be done in any axis, either the x-axis or the y-axis, it typically its done sweeping the y-axis, so that's what I will explain here, to understand the algorithm, suppose that there is no intersection between three segments at the same time and that no segment is parallel to the y-axis.

Every segment is formed by two points, so the algorithm has a state line, that is a self-balancing tree to keep $O(log\ n)$ height, and it's gonna stop in three events and depending on the event we are going to act on the state line:

1. The beginning of a segment, on which case the segment will be inserted on the state line and we must ask if it intersects with its neighbours at the state line, if they do, obtain the intersection point, add it to the events.

2. The end of a segment, just remove the segment from the state line and check if the neighbours intersect, if they do, then add the intersection point to the event.

3. The point of two segments that intersect, then swap the segments that form the intersection and check if they intersect with its new neighbours, if they do, then add that point to the events.

Since every operation is an insertion or deletion at the state line and has balanced height, $O(log\ n)$ height but the problem is how many times we will do this, we do it 2 times for each segment and one time for every intersection and we can have at most n^2 intersections, but that sounds really dramatic since we can have even less intersections, so let's say it is a number k of intersections which is bounded by $0 \leq k \leq n^2$, so the final complexity time would be $O((n+k)\ log\ n)$.

If we do have the 3 segment intersection, we can just do it this way but by 2 and if, for instance, we have that the segments S_1, S_2, S_3 intersect in a point, if we treat if by non directed pairs, the algorithm will report the intersection of (S_1, S_2), (S_1, S_3) and (S_2, S_3) which is exactly what we wanted, and if we have a segment that is parallel to the axis that we are sweeping on, then just take as beginning the closest point to zero, just by convention, you are just going to stop at the same z coordinate two times, but one with the beginning action and the other point with the beginning action.

Things to consider, you MUST do the events in order, so you need to sort the points according to the axis that you are sweeping on, if two segments have the same coordinate, then take the one with the smallest id, since you have to name them all.

3. Implementation

So there's a ton of things to consider, the segments, the beginning of a segment, the end of a segment, if it intersects with another segment and how we are going to keep the order of the events, so that's why I decided to pick Java, but let's just go issue by issue.

1. The segments, with a class in Java, every segment will know their id, their beginning point and their end point and add extra behavior such as comparing itself with other segment, asking if it intersects with another segment, and a *toString( )* to make it user-friendly.

2. The state line, we have to program a self-balancing structure that allows the IDU (Insert/-Delete/Update) operations and the tree spin operation, we can achieve this with an AVL tree just considering that whenever you have to spin, you have to self-balance the rest of the tree, which is $O(log\ n)$, so it doesn't really matter.

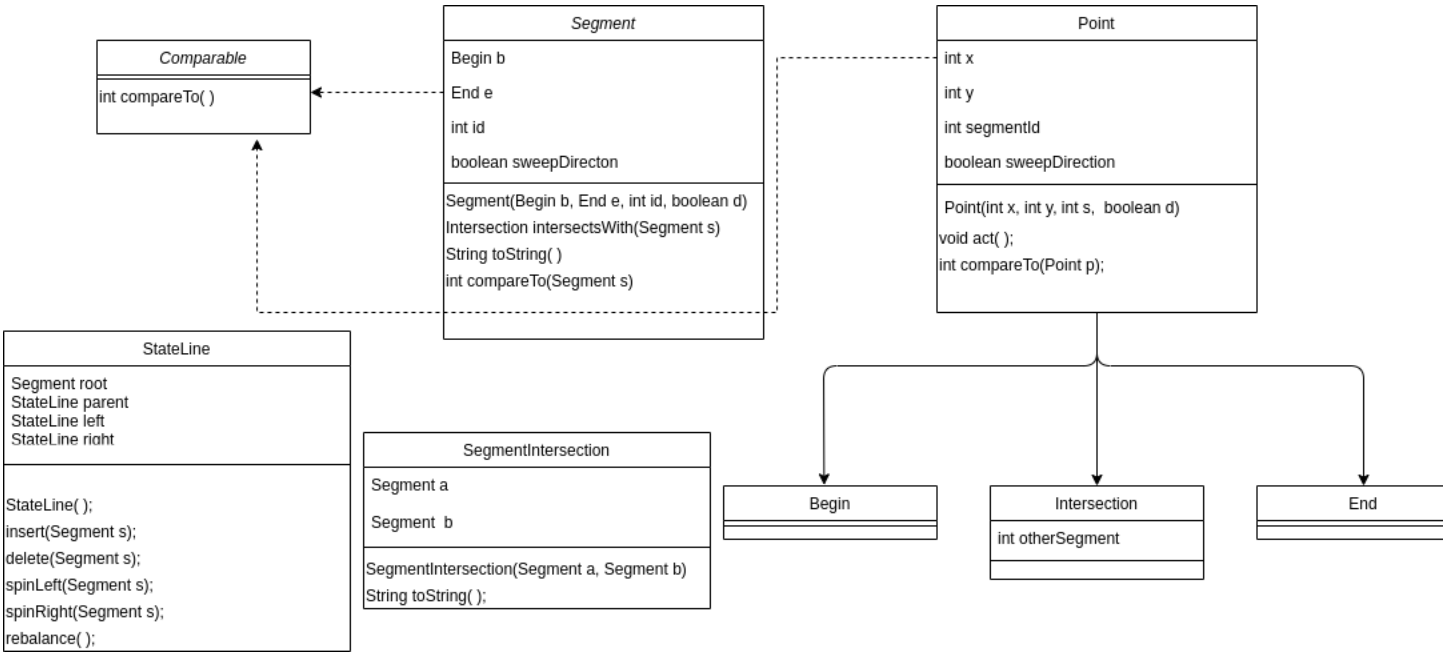
3. The events, how can we achieve to have the events sorted all the time? Moreover, whenever an intersection is found if it's found really far away from the next point, then it doesn't really matter, but if what if it is the next event that we should see? Well, for this we can use a Priority Queue which ensures that has the smallest (or biggest) element up front, so each segment needs to know their respective order.

4. The points have the same structure, by this I mean that each point knows its own coordinates and to which segment they belong, the only thing they are different its in what they do to the state line, so this shouts to be implemented as an abstract class, to be implemented by 3 sub classes which would be, Beginning Point, End Point and Intersection Point on which, Intersection Point must have 2 ids, the two of the ones they intersect.

5. Sweeping direction, since we can sweep either in the x-axis or the y-axis, the program should be able to give the user this option, so there will be but this is not part of the MVP (Most Valuable Product).

6. Once the intersections have been found, we would like to have the segments or the intersection points? For this, there will be a flag, that indicates the program what to return, either the intersection points or the segments that intersect, for this there will be another class that represents them as an non directed pair.

So, due to the fact that we must use inheritance and polymorphism, that's why I decided to type the code in Java and since this sounds like too much to code mentally, here is the class diagram.



Where *Point* is an abstract class, and the only abstract method is act, which receives a *StateLine* that acts as a self-balancing AVL tree, that can turn left, turn right, and IDU operations.

With this said and done, I leave you with nothing but the code, thanks, remember to give credits and this code is under a FREE SOFTWARE MIT License.

4. Applications

1. Determine when 2 convex polygons intersect.
2. Given a map with determined regions and routes, find if there is a route from point A to point B.
3. Monotone polygons triangulation.