Universität Konstanz
Fachbereich Informatik
Nick Krumbholz

January 14, 2024
WS 23/24
IRQ

Extreme Intersection Points

1 Introduction

We are given a set of Lines $\mathcal{L}$. We assume that each line is non-vertical (if they do not, we can transform all lines by using a rotation with small $\varepsilon > 0$). This means a line can be modelled using the function $y = mx + b$. An extreme intersection point (eip) of a line is an outer intersection point, hence all other intersections lie on one side of the point.

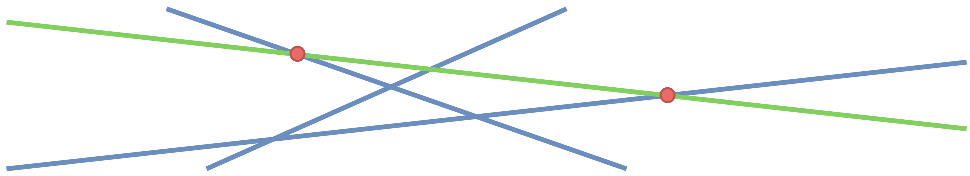


Figure 1: The red points are eip's of the green line.

We let x_p be the x-value associated with point p. A more formal definition can be then given as follows.

Definition 1: Let $I(L)$ be the set of intersection points of line $L \in \mathcal{L}$. An intersection point p_{ij} between two lines L_i and L_j is an *extreme intersection point (eip)* of L_i iff $\forall q \in I(L) : x_{p_{ij}} \leq x_q$ or L_i iff $\forall q \in I(L) : x_{p_{ij}} \geq x_q$. Or in words all intersection points in $I(L_i) \backslash \{p_{ij}\}$ lie on one side of p_{ij}.

Also we will introduce notation to differentiate between the two (or possibly one) eip's of
Definition 2: Let p_{ij} be an eip of L_i. If L_i holds $\forall q \in I(L) : x_{p_{ij}} \leq x_q$ the point p_{ij} is also called a *left eip (leip)* of L_i. Consequently, the point p_{ij} is called *right eip (reip)* of L_i iff $\forall q \in I(L) : x_{p_{ij}} \geq x_q$ holds.

The computation of all eip's is a key part of IRQ. This is because a line L has an intersection in a half-plane iff the half-plane contains at least one eip of L. Therefore in section 2.1 and section 2.2 we propose two algorithms addressing the computation of eip's of a set of lines. To compare runtime, we implemented all algorithms in Rust. This can be found on GitHub at https://github.com/nickrmb/Intersection-Range-Querries.

2 Algorithms

2.1 Block Algorithm

The idea is inspired by existing sweep line algorithms. When sweeping from left to right (or right to left) we build blocks that consist of neighboring lines with known leip (or reip). Doing this, we can efficiently find extreme intersection points.

Definition 3: A Block $b_{ij} \subseteq \mathcal{L}$ is a block consisting all Lines L_k with $i \leq k \leq j$. The outlines of the block can be modelled using two envelopes (an upper and a lower one), whereas each can be constructed using a sequence of line segments.

Definition 4: A block is left-valid with respect to $x \in \mathbb{R}$ iff no line in b_{ij} has an intersection with any line in $\mathcal{L} \backslash b_{ij}$ in the interval $(-\infty, x]$. The same concept can be derived for right-validity.

Algorithm 1 *compute_leip*($\mathcal{L}$)

```
bySlope ← sort L by slope ascending                              ▷ L_i = bySlope[i]
pq ← create priority queue
FI ← create array with length |L|                 ▷ FI[i] corresponds to first inter-
                                                  ▷ section of L_i, initially set to ∞
IW ← create array with length |L|                ▷ IW[i] = j symbolizes that L_i has
                                                 ▷ currently first intersection with L_j
IB ← create array with length |L|                 ▷ IB[i] memorizes whether L_i is
                                                  ▷ in a block, initially set to false
for Line L_i ∈ bySlope do
    x ← first intersection with neighboring lines, ∞ if no exists
    IW[i] ← index of first intersecting neighboring line, None if no exists
    insert i into pq with priority x
end for
while pq non-empty do
    (i, x) ← (key, value) of pq.extractMin()
    j ← IW[i]
    FI[i] ← x
    if IB[j] ≠ false then
        merge L_i into block that consists L_j                ▷ update IB accordingly
    else
        FI[j] ← x
        remove key j from pq
        create new block with L_i and L_j                     ▷ update IB accordingly
    end if
    curBlock ← block of L_i
    check for left-valid merges with neighboring blocks         ▷ with respect to x
                                                  ▷ corresponding to bySlope order
    update intersection with line above / below the (merged) block
    update IW and pq accordingly
end while
return reordered version of FI                    ▷ in respect to the original order
```

The algorithm for *compute$_r$eip* can be simply derived by Algorithm 1. Using both algorithms we find both (or one if leip = reip) eip's.

2.1.1 Runtime Analysis

The algorithm runtime heavily depends on the representation of a block and global data structures handling them. For the following we define $n = |\mathcal{L}|$.

Block representation: We suggest to represent a block using its envelopes, therefore using its upper and lower envelope. Each of these envelopes consists of a sequence of line segments, which we can implement using a list or an array. In our implementation we used a vector (a dynamic array in Rust). We need to consider the following operations:

- $CreateBlock(L_i, L_j)$: The creation of a block obviously uses constant time.
- $IntersectionWithLine(L_i)$: The intersection between an envelope and a line can be implemented using binary search on the line segments of the envelope. Therefore, for an envelope with m segments its runtime is given by $\mathcal{O}(\log m) \subseteq \mathcal{O}(\log n)$.
- $AddLineToBlock(L_i)$: Using the binary search we can also find the intersection point needed in the insertion of a line to an envelope. Afterwards we need to truncate the array, update the last line segment and add the new line segment in constant time. Total time of this operation then equals $\mathcal{O}(\log m) \subseteq \mathcal{O}(\log n)$.
- $MergeBlocks(B_i, B_j)$: We have to merge both upper envelopes, as well as both lower envelopes. To merge two envelopes we can make a binary search one envelope. For each line segment we apply the $InsertionWithLine$ operation with the other envelope to find its intersection point. If none exists we determine whether we need to look at line segments at the left or the right of it. After determining the intersection point we can concatenate the two sub-parts of the envelopes in constant time. As the complete procedure involves two nested binary searches, for two envelopes with k and m segments, its runtime is given by $\mathcal{O}(\log k \cdot \log m) \subseteq \mathcal{O}(\log^2 n)$.

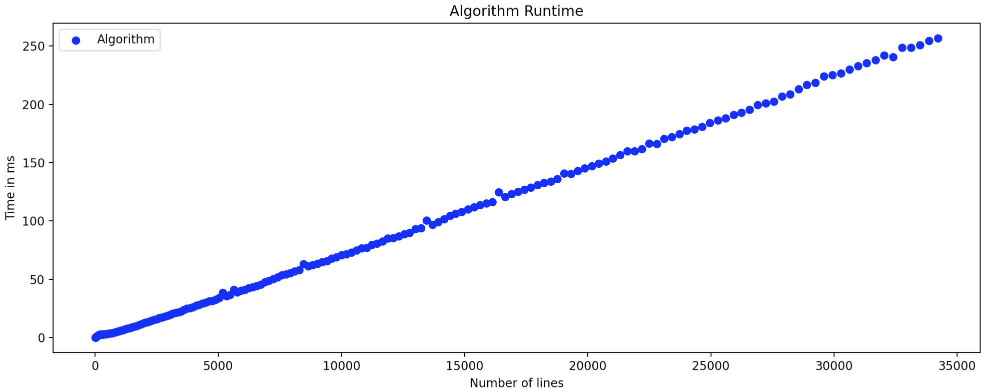
Global structures: Two important structures are already defined in the pseudo-code: IW (intersection with) and IB (in block). Additionally we have an array of blocks (called *blocks*), containing pointers to blocks that we currently manage (initially no blocks exists). Therefore, if a line L_i is in a block, the block can be accessed by $blocks[IB[i]]$.
The IW is a simple array that remembers the line that we currently have an intersection with according to the priority queue entry. As the intersection is always with a neighbor line or block, we can set $IW[i]$ to either $i+1$ or $i-1$, indicating an intersection with L_{i+1} or L_{i-1} accordingly, or with block $blocks[IB[i+1]]$ or $blocks[IB[i-1]]$. Which of this cases hold is determined by whether $IB[i+1]$ or $IB[i-1]$ is defined or not (e.g. -1 equals undefined).
The data structures can be trivially updated when performing $CreateBlock$ and $AddLineToBlock$ constant time.
For the $MergeBlocks$ operation we would have to update $\mathcal{O}(n)$ many values in IB. But it actually suffices to update the new most outer lines (according to *bySlope* order) exclusively, as neighboring lines of the block will only refer to neighboring lines as well. This way we also need constant time to update the data structure for this operation.

The *bySlope* order can be computed using an $\mathcal{O}(n \log n)$ sorting algorithm like QuickSort. Filling the priority queue and initializing our global data structure can be can be achieved in $\mathcal{O}(n)$.
As we remove at least one element in each iteration from the priority queue we have $\mathcal{O}n$ many iterations. In each iteration we update the global data structures and blocks. Each of those operations are executed a constant bounded number of times in an iteration and use at most $\mathcal{O}(\log^2 n)$ time. The reordering before returning can also be achieved in linear time. Thus, the total runtime is given by $\mathcal{O}(n \log^2 n)$.

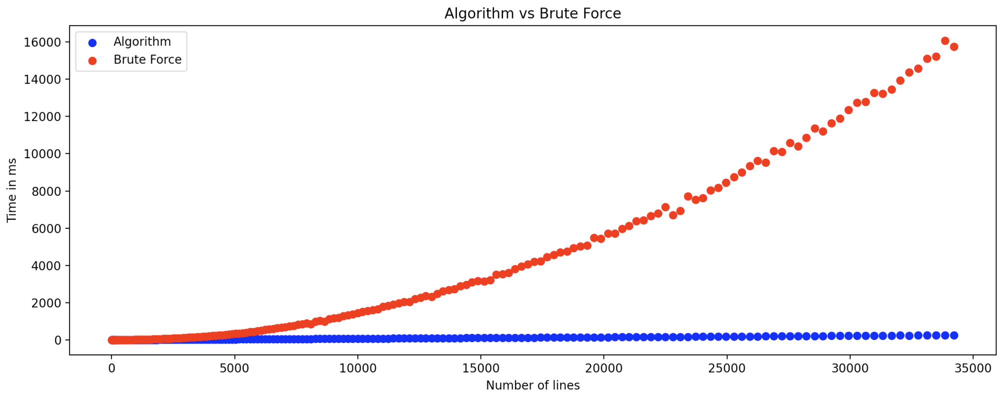
We can change the algorithm trivially to compute all reip's. If we run both algorithms we get all eip's in the same asymptotic time. Alternatively (in our implementation) one may flip all lines horizontally and then computes all leip's again, then flips their x value back by inverting its sign. This approach obviously also does not change the asymptotic runtime.

2.1.2 Empirical Analysis

We conducted a small empirical study to backup our theoretical analysis. To do this we created artificial random instances by creating a set of random lines. These lines have a random slope between -10000 and 10000, as well as a random offset between -10000 and 10000 (uniform distribution). The size of these instances varied between 2 and 40000. Results are given in the following figure.



We can definitely see a linear trend which greatly resembles the theoretical analysis of $\mathcal{O}(n \log^2 n)$.
To get a better understanding of the speedup that we gain compared to a more simpler approach, we compared with a simple "brute force" method. For each line it iterates through all other lines, computes its intersection and keeps track of the most left and most right intersection. Obviously this approach takes $\mathcal{O}(n^2)$ time. The comparison can be seen in the following figure.



Initially, the algorithm exhibits slower performance for instances < 400. However, beyond this point, a noticeable asymptotic difference becomes apparent, with the brute force approach experiencing a substantial increase in growth rate. The suggested algorithm demonstrates an average speed improvement of 22 times compared to the brute force alternative, achieving a maximum speed advantage of over 63 times for larger instances.

2.2 Envelope Algorithm

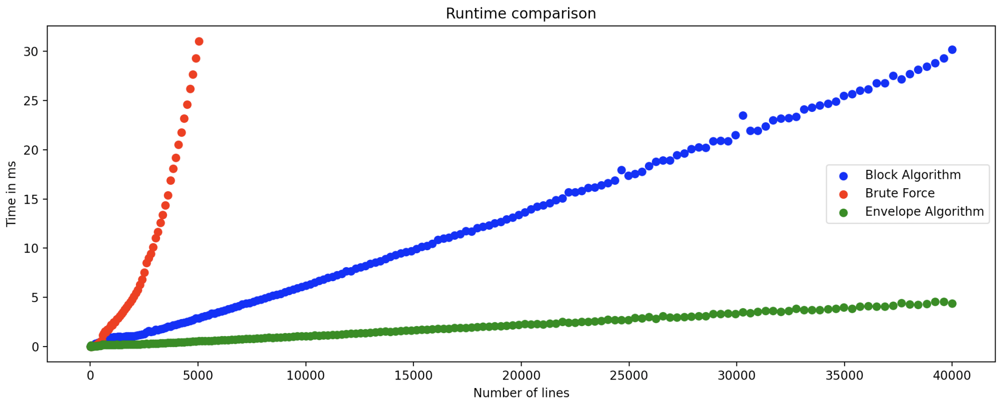
An alternative approach to computing extreme intersection points emerges when examining the collective shape they form. Notably, it becomes evident that each extreme intersection point contributes to the envelope that encompasses all intersection points. Consequently, one could compute this envelope and assess whether each envelope point qualifies as an extreme intersection point. Fortunately, M. Keil [1] has presented a straightforward method for computing the envelope.

2.2.1 Runtime Analysis

The algorithm proposed by M. Keil [1] operates in linear time. Since we can update the current maximum/minimum intersection point for each identified envelope point in constant time, there is no additional asymptotic time complexity introduced. As a result, the adapted algorithm runs in $\mathcal{O}(n)$.

2.2.2 Empirical Analysis

To validate the theoretical findings, we conducted a similar empirical analysis as detailed in section 2.1.2. However, in this case, we extended our investigation by applying the envelope algorithm. This allowed for a comparative assessment with the previously introduced algorithms.



The envelope algorithm demonstrates superior speed, notably due to its absence of additional log scaling. This efficiency also extends to larger instances, as the algorithm efficiently handles any existing overhead found in the block algorithm. This is achieved through the use of simple data structures that require minimal initial setup time. Consequently, the envelope algorithm outpaces the force algorithm for instances with a size ≥ 100. In all cases, the envelope algorithm consistently outperforms the block algorithm, resulting in an average speedup of approximately 450% compared to the block algorithm.

References

[1] M. Keil. “A simple algorithm for determining the envelope of a set of lines”. In: *Information Processing Letters* 39.3 (Aug. 1991), pp. 121–124. ISSN: 0020-0190. DOI: 10.1016/0020-0190(91)90106-r. URL: http://dx.doi.org/10.1016/0020-0190(91)90106-R.