Universität Konstanz
Fachbereich Informatik
Nick Krumbholz

January 3, 2024
WS 23/24
IRQ

Extreme Intersection Points

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}.

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.

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*($\mathcal{L}$)

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

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 $\mathcal{O}(\log k \cdot \log m) \subseteq \mathcal{O}(\log^2 n)$.

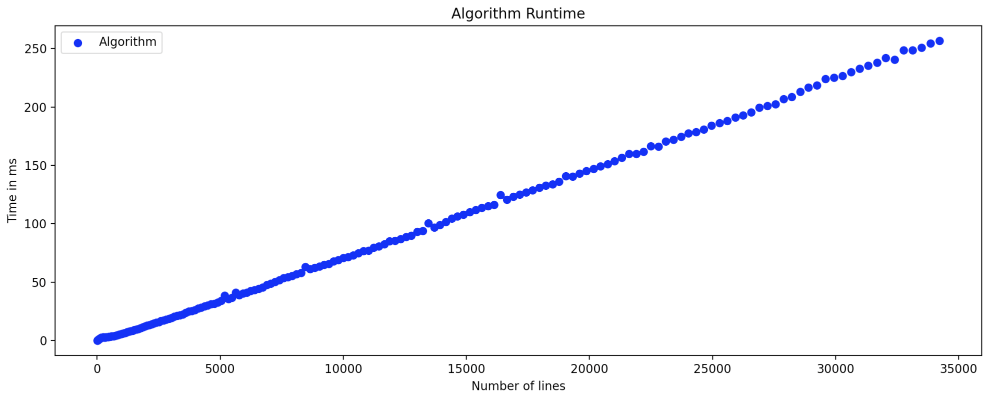
Global structures: Two important structures are already defined in the pseudo-code: IW (intersection with) and IB (in block). Additionally we have an array of blocks (called *blocks*), containing pointers to blocks that we currently manage (initially no blocks exists). Therefore, if a line L_i is in 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 $\mathcal{O}(n \log^2 n)$.

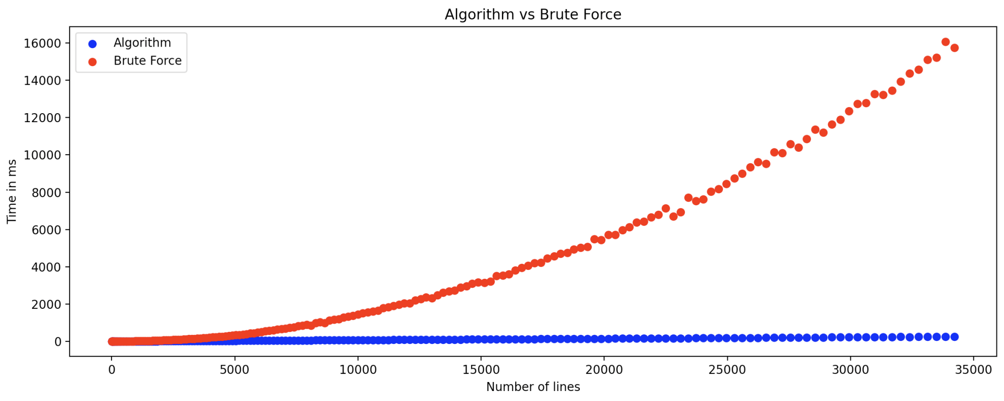
We can change the algorithm trivially to compute all reip's. If we run both algorithms we get all eip's in the same asymptotic time. Alternatively (in our implementation) one may flip all lines horizontally and then computes all leip's again, then flips their x value back by inverting its sign. This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

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 is slower for instances with $n < 400$. But from there on we can definitely see the asymptotic difference, as the brute force approach grows largely faster. The proposed algorithm showed an average speed increase of 22 times compared to the brute force alternative, reaching a maximum speed advantage of over 63 times for larger instances.