Literature Survey of the "Slope Selection Problem"

Varun Aravapalli

January 7th 2024

Contents

1 Problem Definition

Given n distinct points in general position that create $\binom{n}{2}$ slopes and an integer k such that $1 \leq k \leq \binom{n}{2}$, how do we find the k^{th} smallest slope?

2 General Selection By Cole

2.1 Pseudo Code

1. **Initialization:** Begin by writing a permutation π(a) for the intercepts of the lines at x = a in descending order. This permutation will help in sorting the intercepts efficiently.

2. **Building Sequential Algorithms:** Apply Megiddo's technique of building sequential algorithms from parallel ones, which involves an implicit binary search over the intersection points

3. **Sorting Intercepts:** Sort the intercepts of the lines at x = a using the permutation π(a). This sorting gives us O(n log n) "intersection points," where each intersection point can be compared in the form "yi(a) $<$ yj(a)?". The O(n log n) answers will yield the permutation (π*) that sorts these intercepts.

4. **Sweeping**: After the sorting step, there is a sweeping phase where the algorithm sweeps through the intersection points to find the kth smallest x-coordinate. This involves tracing through the points and lines to identify the intersection with the kth smallest x-coordinate.

5. **Counting inversions:** After the sorting network, the algorithm uses inversion counts to calculate the number of inversions in the sorted intercepts. This is done to determine the relative ordering of the intercepts based on the rankings obtained from the sorting network.

6. **Partitioning:** Based on the inversion counts and the sorted intercepts, the algorithm partitions the intercepts at a specific intersection point. The goal is to create subgroups of intercepts based on a given threshold for the number of inversions.

7. **Halving:** In this step, the algorithm halves the partition size and updates the inversion counts accordingly. This is done to refine the partition and adjust the number of inversions in each subgroup.

8. **Resolving:** Finally, based on the refined partitions and inversion counts, the algorithm resolves the desired slope, determining its rank and relative position within the set of intercepts.

2.2 Runtime Analysis

- Resolving each question (yi(a*) < yj(a*)) involves ranking, which takes O(n log n) time.

- Explanation of the ranking: The rank of u_{ij} is the number of inversions in $\pi(u_{ij})$. If $\pi(u_{ij}) > k$ we know that $u_{ij} > t_k$ so the answer is "no;" lines i and j have not yet crossed at t_k. Similarly if $\pi(u_{ij}) < k$, $u_{ij} < t_k$ and the answer is "yes." If $\pi(u_{ij}) = k$, $t_k = u_{ij}$. u_{ij} may be computed in time $O(n \log n)$. We use merge-sort to sort the slopes $m_1, \ldots, m_n$, and count the inversions that were performed.

- Counting inversions, necessary for resolving questions, is repeated for each level of the sorting network, adding another log n factor.

- Since there are O(log n) levels in the sorting network and each level requires O(n log n) steps, the total complexity is $O(n(logn)^3)$.

3 Optimizing the runtime

3.1 Using an approximation algorithm

3.1.1 Pseudo Code

1. **Approximation:** The new idea is to use an approximate rank for each point chosen by the sorting network. The algorithm will use O(n log n) time to find the rank of each of the O(log n) weighted median points. This rank determined the sign of the approximate rank minus tk, which allows resolving the relevant points based on their relative order.

2. **Reference point maintenance:** Maintain two reference points, rL <= tk and rR >= tk, with respective orderings p(rL) and p(rR) and calibrated partition sizes TL and TR for both reference points. The reference point with the smaller value of T is designated as "active."

3. **New Query Point:** When the network gives a new query point x=q, if q < rL or q > rR, immediately resolve the point. Otherwise, construct the partition at x=q and then resolve it.

4. **Partition Calculation:** Calculate the partition at the new query point q, taking care to ensure that the partition follows the constraints set for its derivation from the partition at the reference points.

5. **Resolving the Point:** If the partition is accurate enough to determine the relative position of q and tk, resolve the point. Otherwise, proceed to recalculate the partition for the new query point at the other reference point, and repeat the process until the partition is accurate enough for resolution.

6. **Finalization:** Once the relative position of the new query point with respect to tk is known, perform the necessary actions to set up the variables for the next iteration. This includes changing the active reference point based on the new T value and preparing for the next query point.

These steps with the approximation algorithm differ from the previous algorithm as it introduces the concept of using an approximate rank for each point chosen by the sorting network, leading to improved efficiency in finding the rank of a point for the slope selection problem.

3.1.2 Runtime Analysis

Instead of spending O(n log n) time to find the exact rank of each median point, the improved algorithm used O(n) (amortized) time to develop an approximation to the desired rank. This approach relied on the fact that if the error in the approximation was small enough, it could determine the relative ordering of the current (z) and the target (k).

This strategy reduced the amount of work needed for ranking and refining approximations, thus bringing down the overall complexity of the algorithm to $O(nlog^2n)$.

3.2 Using Cuttings

3.2.1 Pseudo Code

1. **Slabs:** This algorithm uses "Vertical Slabs" which are defined as a portion of the 2-D plane between two vertical lines. It maintains two permutations ($\pi(l)$ and $\pi(r)$) that track the order of the lines and intersections.

2. **Trapezoids:** A group of vertical trapezoids that cover the entire slab is stored along with their "conflict lists" (sets of lines that cross the trapezoids).

3. **Invariants:**

I1 and I2 maintain the inversions.

I3 makes sure that each trapezoid is crossed by a limited number of lines.

I4 limits the size of each conflict list.

4. **Slab Adjustment:** "Critical Points" within the slab are identified and their median is computed. The slab is then divided on this median and the "winning slab" (portion most likely to contain the solution) is chosen.

5. **Trapezoid Adjustment:**

- Trapezoids that don't intersect the winning slab are removed.
- Conflict lists are updated based on the slab's new boundaries.
- The blocked permutations are halved and relocked when necessary.

6. **Cutting Adjustment:** This step is used when slab adjustment isn't sufficient. We compute a new for each conflict list and update the trapezoids to further reduce the number of lines intersecting each trapezoid.

3.2.2 Runtime Analysis

The algorithm alternates between slab adjustment and cutting adjustment steps a log number of times. Each reblocking, halving, and cutting refinement requires $O(n)$ time and the algorithm iterates for $\log(n)$ times giving a total runtime of $O(n \log n)$.

3.3 Using Expander Graphs

3.3.1 Pseudo Code

1. **Setup:** Start with a vertical slab that contains the vertices, partitioned into "vertical trapezoids" which are each associated with subsets of lines that cross them.

2. **Properties:**

- The union of trapezoids covers the slab.
- Each trapezoid is intersected by a certain amount of lines.
- The total number of lines crossing all trapezoids is regulated.
- Each vertex within the slab is at an intersection of lines that belongs to one of the trapezoids.

3. **Expander Graph:**

- At each stage, for each trapezoid, an expander graph is constructed over the set of lines that crosses the trapezoid.
- The number of edges in each expander graph is proportional to the number of lines crossing the trapezoid.

4. **Partition Adjustment:** Each large trapezoid is partitioned into smaller trapezoids, each crossed by a certain number of lines, to ensure that the properties are maintained.

5. **Slab Adjustment:** The algorithm shortens the slab by using binary searches among the sub-slabs, which each contain a manageable number of critical points.

6. **Termination:**

- After $O(\log n)$ stages, the algorithm ends with the slab containing v_k and a partition into trapezoids, each crossed by a number of lines held constant.
- Then we can compute all the intersection points within each trapezoid's crossing list.
- The union of all of these points gives us the vertices in the final slab.

3.3.2 Runtime Analysis

Since there are $O(\log n)$ stages and each takes $O(n \log n)$ time, resulting in a runtime of $O(n \log^2 n)$ but since our algorithm generates $O(\log n)$ oracle calls, we can use the parametric searching technique by Megiddo to reduce another log factor. Resulting in an overall runtime of $O(n \log n)$.

References

[1] R. Cole, J. Salowe, W. Steiger and E. Szemer6di, Optimal slope selection, SIAMZ Comput. 18 (1989) 792-810.

[2] J. Matougek, Randomized optimal algorithm for slope selection, Inform. Process. Lett. 39 (1991) 183-187.

[3] N. Megiddo, Applying parallel computation algorithms in the design of serial algorithms, J. A C M 30 (1983) 852-865.

[4] Matthew J Katz, Micha Sharir, Optimal slope selection via expanders, Information Processing Letters 47 (1993) 115-122 14 September 1993

[5] Herve Bronnimann, Bernard Chazelle, Optimal slope selection via cuttings, Computational Geometry 10 (1998) 23-29