Lecture Notes: Distribution Sweeping

Yufei Tao
Department of Computer Science and Engineering
Chinese University of Hong Kong
taoyf@cse.cuhk.edu.hk

In this lecture, we will study the *distribution sweeping* technique proposed by Goodrich et al. [2]. This can be regarded as the combination of the *divide-and-conquer* and *planesweep* paradigms, and has been leveraged to solve a large number of computational geometry problems in EM. Next, we will discuss two representatives of such problems.

1 Skyline

The *skyline* (a.k.a. *maxima*) *problem* is defined as follows. The input is a set P of N points in $\mathbb{R}^d$ in *general position*, i.e., no two points of P share the same coordinate on any dimension. Given a point $p \in \mathbb{R}^d$, denote its i-th ($1 \le i \le d$) coordinate as $p[i]$. A point p_1 is said to *dominate* another point p_2, represented as $p_1 \prec p_2$, if p_1 has a smaller coordinate on all d dimensions, namely:

$$\forall i = 1, ..., d,\ p_1[i] < p_2[i].$$

The goal is to compute the *skyline* of P, denoted as $SKY(P)$, which includes all the points in P that are not dominated by any other points, namely:

$$SKY(P) = \{p \in P \mid \nexists p' \in P \text{ s.t. } p' \prec p\}. \tag{1}$$

The skyline problem can be easily solved in linear I/Os in 2D space, after sorting the points of P by x-coordinate (think: how). We will show that the 3D version of the problem can also be settled in $O(\frac{N}{B} \log_{M/B} \frac{N}{B})$ I/Os. Our description below follows that of [3].

Algorithm. Our algorithm accepts as input a dataset P whose points are sorted by x-coordinate. If $N \le M$ (i.e., the memory capacity), we simply find the skyline of P using a memory-resident algorithm[1]. The I/O cost incurred is $O(N/B)$.

Now consider $N > M$. We divide P into $s = \Theta(M/B)$ partitions $P(1), ..., P(s)$ with the same size $\lceil N/s \rceil$ (except perhaps one partition), such that each point in $P(i)$ has a smaller x-coordinate than all points in $P(j)$ for any $1 \le i < j \le s$. As P is already sorted on the x-dimension, the partitioning can be done in linear cost, while leaving the points of each $P(i)$ sorted in the same way.

The next step is to obtain the skyline $\Sigma(i)$ of each $P(i)$, i.e., $\Sigma(i) = SKY(P(i))$. Since this is identical to solving the original problem (only on a smaller dataset), we recursively invoke our algorithm on $P(i)$. Now consider the moment when all $\Sigma(i)$ have been returned from recursion. Our algorithm proceeds by performing a *skyline merge*, which finds the skyline of the union of all $\Sigma(i)$, that is, $SKY(\Sigma(1) \cup ... \cup \Sigma(s))$, which is exactly $SKY(P)$. We enforce an invariant that, $SKY(P)$ be returned in an array where the points are sorted in ascending order of y-coordinate (to be used by the upper level of recursion, if any). Due to recursion, the invariant implies that, the same ordering has been applied to all the $\Sigma(i)$ at hand.

[1]Note that this is not entirely rigorous because if $N = M$, then our memory has been fully occupied such that we cannot even store any other "auxiliary" information needed to find the skyline in memory. But this is a very minor issue; think: why?

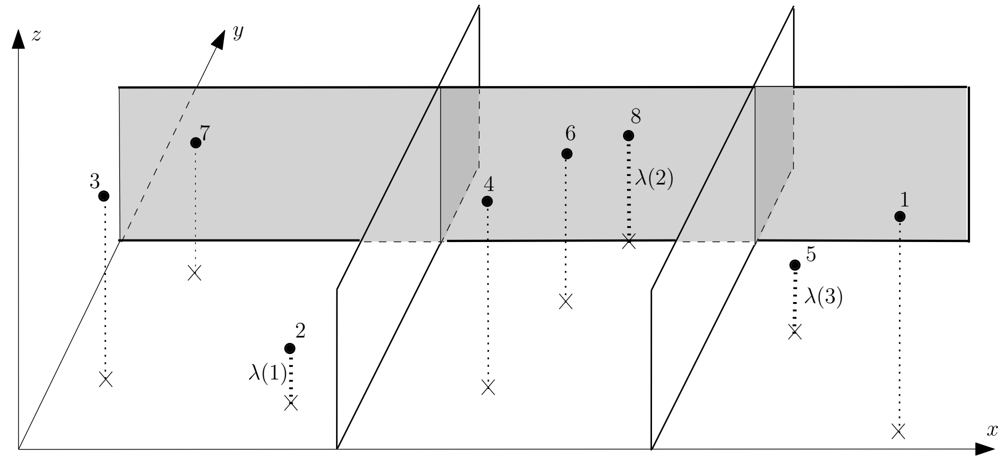

Figure 1: Illustration of 3d skyline merge. The value of s is 3. Only the points already encountered are shown. Points are labeled in ascending order of y-coordinate (which is also the order they are fetched). Point 8 is the last one seen. Each cross is the projection of a point in the x-y plane. $\Sigma(1)$ contains points 2, 3, 7, $\Sigma(2)$ includes 4, 6, 8, and $\Sigma(3)$ has 5, 1. $\lambda(1), \lambda(2), \lambda(3)$ equal the z-coordinate of point 2, 8, 5, respectively. Point 8 does not belong to $SKY(P)$ because its z-coordinate is larger than $\lambda(1)$ (i.e., it violates Inequality 2 on $j = 1$).

We now elaborate on the details of the skyline merge. $SKY(P)$ is empty in the outset. $\Sigma(1), ..., \Sigma(s)$ are scanned synchronously in ascending order of y-coordinate. In other words, the next point fetched is guaranteed to have the lowest y-coordinate among the points of all $\Sigma(i)$ that have not been encountered yet. As $s = \Theta(M/B)$, the synchronization can be achieved by assigning a block of memory as the input buffer to each $\Sigma(i)$. We maintain a value $\lambda(i)$, which equals the minimum z-coordinate of all the points that have already been seen from $\Sigma(i)$. If no point from $\Sigma(i)$ has been read, $\lambda(i) = \infty$.

We decide whether to include a point p in $SKY(P)$ when p is fetched by the synchronous scan. Suppose that p is from $\Sigma(i)$ for some i. We add p to $SKY(P)$ if

$$p[3] < \lambda(j), \forall j < i \tag{2}$$

where $p[3]$ denotes the z-coordinate of p. See Figure 1 for an illustration. The lemma below shows the correctness of this rule.

Lemma 1. *$p \in SKY(P)$ if and only if Inequality 2 holds.*

Proof. Clearly, p cannot be dominated by any point in $\Sigma(i+1), ..., \Sigma(s)$ because p has a smaller x-coordinate than all those points. Let S be the set of points in $\Sigma(j)$ already scanned before p, for any $j < i$. No point $p' \in \Sigma(j) \setminus S$ can possibly dominate p, as p has a lower y-coordinate than p'. On the other hand, all points in S dominate p in the x-y plane. Thus, *some* point in S dominates p in $\mathbb{R}^3$ if and only if Inequality 2 holds. □

We complete the algorithm description with a note that a single memory block can be used as an output buffer, so that the points of $SKY(P)$ can be written to the disk in linear I/Os, by the same order they entered $SKY(P)$, namely, in ascending order of y-coordinate. Overall, the skyline merge finishes in $O(N/B)$ I/Os.

I/O Cost. Denote by $F(N)$ the I/O cost of our algorithm on a dataset with cardinality N. It is clear from the above discussion that

$$F(N) = \begin{cases} O(N/B) & \text{if } N \le M \\ \sum_{i=1}^{s} F(|P(i)|) + O(N/B) & \text{otherwise} \end{cases}$$

where $\sum_{i=1}^{s} |P(i)| = N$, and $|P(i)| \le \lceil N/s \rceil$. Solving the recurrence gives $F(N) = O(\frac{N}{B} \log_{M/B} \frac{N}{B})$.

Remark. Note the "distribution" and the "sweep" components of the above algorithm. Also note that the algorithm used the idea of "order alternation". Namely, the dataset we feed into the next level of recursion is sorted by x-coordinate, whereas on return from the recursion, we have got a dataset sorted by y-coordinate. It is also known that $\Omega(\frac{N}{B} \log_{M/B} \frac{N}{B})$ is a lower bound for the class of all comparison-based algorithms (the algorithm we have presented belongs to this class).

2 Orthogonal Segment Intersection

In the *orthogonal segment intersection problem*, we are given a set R of horizontal segments and a set S of vertical segments, all in the plane $\mathbb{R}^2$. The goal is to report all pairs of segments $(r, s) \in R \times S$ such that r intersects s. We assume once again that the endpoints of the segments in $R \cup S$ are in general position: no two endpoints share the same x-coordinate or y-coordinate.

Set $N_1 = |R|$ and $N_2 = |S|$. We will describe an algorithm proposed in [1] that solves the problem in $O(\frac{N_1}{B} \log_{M/B} \frac{N_1}{B} + \frac{N_2}{B} \log_{M/B} \frac{N_2}{B} + K/B)$ I/Os, where K is the number of pairs reported.

2.1 Linked List

We will need a *linked list* to support the following operations on a set S of elements:

- *Insertion:* add an element into S.
- *Scan:* visit all the elements of S.

In RAM, it is easy to design such a linked list that occupies $O(|S|)$ space, supports an insertion in $O(1)$ time, and a *scan* in $O(|S|)$ time. In EM, if allowed a dedicated block of memory, we can implement a linked list with the following guarantees:

- The space consumption is $O(|S|/B)$ at all times.
- Each insertion takes $O(1/B)$ amortized I/Os.
- A *scan* takes $O(|S|/B)$ I/Os.

We leave the design of such a linked list to you.

2.2 Algorithm

Let us now return to the orthogonal segment intersection problem. First, if $N_1 + N_2 \le M$, then we can easily solve the problem in memory, entailing cost $O((N_1 + N_2)/B + K/B)$ I/Os. Next, we consider $N_1 + N_2 > M$.

We assume that all the segments of $R \cup S$ are contained in some (big) rectangle $D = [\alpha, \beta] \times \mathbb{R}$ (note that α may be $-\infty$ and β may be ∞). We refer to D as the *data space*. A segment $r \in R$ is (i) *two-sided* if r touches neither the left nor the right boundary of D, or (ii) *one-sided* if r touches exactly one of the left and right boundaries of D. We assume that no segment $r \in R$ touches both the left *and* right boundaries of D, and that no segment $s \in S$ touches the left *or* right boundary of D (think: why are these fair assumptions?).

Let X be the set of x-coordinates of all *non-grounded* endpoints p of the segments in $R \cup S$ satisfying, namely, p is on neither the left nor the right boundary of D. Let Y be the set of y-coordinates of the endpoints in $R \cup S$. Set $N = |X|$. Prior to invoking our algorithm, we assume that we are given X and Y, both of which should have been sorted. The sorting demands only $O(\frac{N}{B} \log_{M/B} \frac{N}{B})$ I/Os, which is affordable.

Divide the data space D into $s = \Theta(M/B)$ disjoint vertical *slabs*, each of which has the form $[x_1, x_2] \times \mathbb{R}$, and should contain $\lceil N/s \rceil$ x-coordinates in P (except perhaps one slab). Denote by $\sigma_1, \sigma_2, ..., \sigma_s$ the slabs in ascending order of x-coordinate. We initialize an empty linked list (Section 2.1) for each slab, by allocating it a memory block.

We perform a *sweeping process* to move a horizontal line ℓ from $y = -\infty$ to $y = \infty$. We carry out different actions when ℓ hits different endpoints of the segments in $R \cup S$:

- *Event 1:* If ℓ hits the lower endpoint of a segment $s \in S$, insert it into the linked list of the slab that contains s.
- *Event 2:* If ℓ hits the upper endpoint of a segment in S, do nothing.
- *Event 3:* Now consider ℓ hits a segment $r \in R$ (since r is horizontal, this means that the entire r is contained in ℓ). If r does not span any slab, ignore it, and continue the sweeping. Otherwise, suppose that r spans slabs $\sigma_{i_1}, \sigma_{i_1+1}, ..., \sigma_{i_2}$, for some i_1, i_2 such that $1 \le i_1 \le i_2 \le s$. For each $j \in [i_1, i_2]$, we perform a *scan* operation on the linked list of σ_j, and divide the segments there into two sets: (i) the set $\Sigma_1(r, j)$ of segments completely below ℓ, and (ii) the set Σ_2 of segments crossing ℓ. For each segment $s \in \Sigma_2(r, j)$, we report (r, s) as a result pair. After this is done, we destroy the linked list entirely, and rebuild it by inserting the segments only in $\Sigma_2(r, j)$. The segments of $\Sigma_1(r, j)$ are discarded—notice that they cannot intersect any other segments of R to be hit by ℓ later.

The sweeping is a continuous process, but the events are discrete, and can be generated easily by scanning Y.

After the sweeping process, we recurse into each slab. Specifically, for each $i \in [1, s]$, define:

$$\begin{aligned} R_i &= \{r \cap \sigma_i \mid r \in R, \text{ and } r \text{ does not span } \sigma_i\} \\ S_i &= S \cap \sigma_i. \end{aligned}$$

We now recursively solve the orthogonal segment intersection problem on R_i and S_i in the data space σ_i, for each $i \in [1, s]$ separately. Note that a two-sided r may become one-sided in a slab.

Analysis. We will first analyze the cost of a sweeping process. Suppose that the process reports K' result pairs. We will prove that the process performs $O(N/B + K'/B)$ I/Os. For this purpose, we will bound the cost of handling Events 1, 2, and 3 separately. First, notice that the generation of all the events requires $O(N/B)$ I/Os (by scanning Y once). Second, Event 1 produces $O(N)$ insertions to linked lists, which perform $O(N/B)$ I/Os in total, while Event 2 entails no cost. Next, we focus on Event 3.

It is easy to see that Event 3, if triggered by ℓ hitting a segment $r \in R$, performs $O(\sum_j (|\Sigma_1(r, j)| + |\Sigma_2(r, j)|)/B)$ I/Os. Therefore, the total cost of handling Event 3 is bounded by

$$O\left(\sum_{r \in R} \sum_j \frac{|\Sigma_1(r, j)| + |\Sigma_2(r, j)|}{B} \right). \tag{3}$$

Let us make two crucial observations:

- Every segment $s \in S$ belongs to the $\Sigma_1(r, j)$ of at most one r. This is because once s falls into $\Sigma_1(r, j)$, it will be discarded right away. It thus follows that $\sum_{r \in R} \sum_j |\Sigma_1(r, j)| \le N$.
- $|\Sigma_2(r, j)|$ equals precisely the number of result pairs that are produced by r in the sweeping process. It thus follows that $\sum_{r \in R} \sum_j |\Sigma_2(r, j)| \le K'$.

Therefore, (3) is bounded by $O(N/B + K'/B)$.

Now it remains to analyze the recursive part. Two facts are easy to observe. First, recall that we have assumed the availability of two ordered lists X and Y. Hence, before starting the recursion into slab σ_i ($i \in [1, s]$), we must prepare the corresponding ordered lists X_i, Y_i. We can prepare all of $X_1, ..., X_s, Y_1, ..., Y_s$ in $O(N/B)$ I/Os (think: how?). Second, every result pair is reported only once.

Let $F(N) + O(K/B)$ be the cost of solving the orthogonal segment intersection problem. Define $N_i = |X_i|$ for each $i \in [1, s]$. Then, we have:

$$F(N) = \begin{cases} O(N/B) & \text{if } N \le M \\ \sum_{i=1}^{s} F(N_i) + O(N/B) & \text{otherwise} \end{cases}$$

Notice that $\sum_{i=1}^{s} N_i = N$, because (i) each non-grounded endpoint of a segment in R falls in exactly one slab, and (ii) both endpoints of a segment in S fall in the same slab. It also holds that $N_i \le \lceil N/s \rceil$. Solving the recurrence gives $F(N) = O(\frac{N}{B} \log_{M/B} \frac{N}{B})$.

Remark. We have obtained an algorithm solving the orthogonal segment intersection problem in $O(\frac{N_1}{B} \log_{M/B} \frac{N_1}{B} + \frac{N_2}{B} \log_{M/B} \frac{N_2}{B} + K/B)$ I/Os. The term K/B is clearly optimal. One can prove that the other terms are also optimal in the class of comparison-based algorithms.

References

[1] L. Arge. External-memory algorithms with applications in gis. In *Algorithmic Foundations of Geographic Information Systems*, pages 213–254, 1996.

[2] M. T. Goodrich, J.-J. Tsay, D. E. Vengroff, and J. S. Vitter. External-memory computational geometry. In *FOCS*, pages 714–723, 1993.

[3] C. Sheng and Y. Tao. Finding skylines in external memory. In *PODS*, pages 107–116, 2011.