"The plane has two independent directions, so it is natural to move around by alternatingly following one or the other. When to take which direction is an issue that determines how long it takes to get there."

2 Planar point location with segment trees

One of the applications of segment trees is to answer point location queries in the plane; these will be defined shortly. Such queries have been studied a great deal in computational geometry, and several solutions are available that achieve optimal time and storage in the asymptotic sense, see e.g. [2, 3, 5]. The solution with the segment tree goes back to [7] and falls short of optimality because it requires $\Omega(n \log n)$ storage.

Problem definition. In the above references, the planar point location problem requires a decomposition of the plane into pairwise disjoint regions. It turns out that connectivity is not essential for most solutions, and that weaker assumptions on the input suffice. We define the ***planar point location*** problem for a collection, S, of pairwise non-crossing line segments in $\mathbf{R}^2$: for a query point $x \in \mathbf{R}^2$, find the line segment vertically above x, if it exists, see figure 2.1.

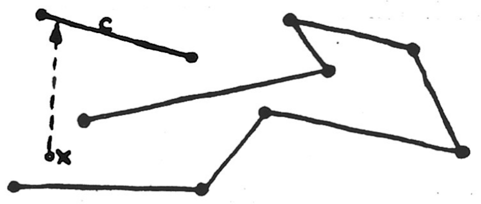

Figure 2.1: The line segment vertically above point x is c.

Segment tree approach. We can use segment trees to answer point location queries by projecting the line segments vertically to the x_1-axis. The resulting set of intervals is stored in a segment tree. The segment of a node κ corresponds to a vertical slab, namely the set of all points whose vertical projection to the x_1-axis belongs to $s(\kappa)$. A line segment ***spans*** a slab if it meets both vertical lines delimiting the slab. Each $L(\kappa)$ can be viewed as a set of line segments, namely the line segments in S that span the slab of κ but not the slab of the parent of κ, see the definition of $L(\kappa)$ in section 1. Observe that the line segments in a set L are naturally ordered from top to bottom, because they intersect the same slab and they do not cross each other. In the data structure, the line segments in $L(\kappa)$ are stored in sorted order in a linear array.

A preliminary analysis shows that this data structure takes time $O(n \log^2 n)$ for construction and a query takes time $O(\log^2 n)$. The amount of storage is still $O(n \log n)$. The $\log^2 n$ term in the construction comes from the need to sort lists of total size $O(n \log n)$; in the worst case there are a few long lists that take all the time. We show below how to reduce this to $O(n \log n)$. The $\log^2 n$ term in the query comes from the fact that $O(\log n)$ lists are searched, each by binary search. Again, a bad case occurs if all lists are long, so each binary search takes time $O(\log n)$. The binary searches are not independent, and we will see how to take advantage of the dependence to improve the query time to $O(\log n)$.

Vertically ordering line segments. For a line segment $a \in S$, let a' be its vertical projection to the x_1-axis. It is convenient to assume that a is non-vertical, so a' is indeed an interval and not just a point. For two line segments, $a, b \in S$, define $a \prec b$ if $\operatorname{int} a' \cap \operatorname{int} b' \neq \emptyset$ and a lies below b, that is, there is vertical line, ℓ, that intersects both line segments in their interiors and the x_2-coordinate of $\ell \cap a$ is less than that of $\ell \cap b$, see figure 2.2. We thus have a relation $(S, \prec)$, and the transitive closure is a partially ordered set.

If the line segments are added to the segment tree in an order compatible with $\prec$, then each list representing a set L will automatically be sorted. These orderings are precisely the linear extensions of $(S, \prec)$. Even though

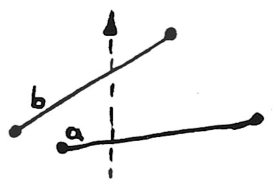

Figure 2.2: $a \prec b$.

the number of pairs related by $\prec$ can be as large as $\binom{n}{2}$, we show that it is possible to construct a relation $\prec_s$ whose size it at most $3n$, and whose transitive closure is the same as that of $\prec$. A linear extension of $(S, \prec_s)$ is also a linear extension of $(S, \prec)$ and can be found in additional time $O(n)$ by topologically sorting the corresponding directed acyclic graph, see e.g. [1, chapter VI] or [4, chapter 2].

Plane sweep. The sparse order, $\prec_s$, can be constructed by sweeping a vertical line, ℓ, from left to right. The sweep is conceptually continuous, but computations are necessary only at a discrete set of positions. An *event* happens whenever ℓ encounters the left or right endpoint of a line segment. At any point in time, the *active* line segments are the ones intersected by ℓ, and they are stored in sorted order along ℓ. Two types of events need to be distinguished.

Event 1. The left endpoint of a line segment $b \in S$ is encountered, see figure 2.3. Add b to the set of active line segments. Add (a, b) and (b, c) to $\prec_s$, where a and c are the active line segments right below and above b, provided they exist.

Event 2. The right endpoint of $b \in S$ is encountered, see figure 2.3. Delete b from the set of active line segments. Add (a, c) to $\prec_s$, where a and c are as in event 1.

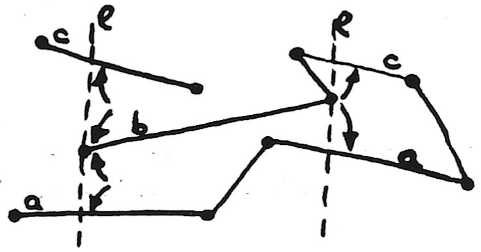

Figure 2.3: When the left endpoint of b is encountered, it is added to the data structure for active line segments. At the same time, two pairs are added to $\prec_s$. When the right endpoint of b is encountered, it is removed from the data structure for active line segments. In addition, a new pair is added to $\prec_s$.

This algorithm can be implemented so it runs in time $O(n \log n)$ using a balanced tree [1, chapter III] for the active line segments. When a left or right endpoint is encountered, we can find a, b, c in time $O(\log n)$ by searching the balanced tree. To determine whether a point, x, lies above or below a line segment pq, we check whether pqx forms a left or a right turn. A line segment can be added to or removed from the collection of active line segments in time $O(\log n)$. The events are obtained by sorting the endpoints along the x_1-direction and scanning the sorted list once. It is possible that an endpoint is shared by several line segments, in which case it gives rise to several events.[1] In this case, events of type 2 are to precede those of type 1. Each line segment generates at most two pairs when its left endpoint is encountered, and at most one when its right endpoint is encountered. This shows that $\prec_s$ contains at most $3n$ ordered pairs. Finally, observe that if $a \prec c$ then there is a sequence $a \prec_s b_1 \prec_s b_2 \prec_s \ldots \prec_s b_j \prec_s c$. It follows that (a, c) is in the transitive closure of $\prec_s$, so the transitive closures of $\prec$ and $\prec_s$ are the same.

Decomposing lists. With the above segment tree, the time to answer a point location query is $O(\log^2 n)$. As mentioned earlier, this can be improved by exploiting the dependence between the various binary searches. The

[1] These events can be taken care of collectively by removing all old line segments and adding all new line segments at once. While this is easy to do, it requires a slightly more complicated implementation than the original two events. Alternatively, the line segments can be processed in sequence. In this case, pairs are generated that belong to the transitive closure of $\prec_s$ but not to $\prec_s$ itself. Since these pairs are accounted for by the $3n$ upper bound argument, they do not cause any harm.

main idea is to store each list, L, in pieces and to represent each piece by a sorted tree. The reason for the pieces is obvious from figure 2.4. Before the binary search in $L(\kappa)$, the algorithm performs a binary search in $L(\mathrm{p}(\kappa))$

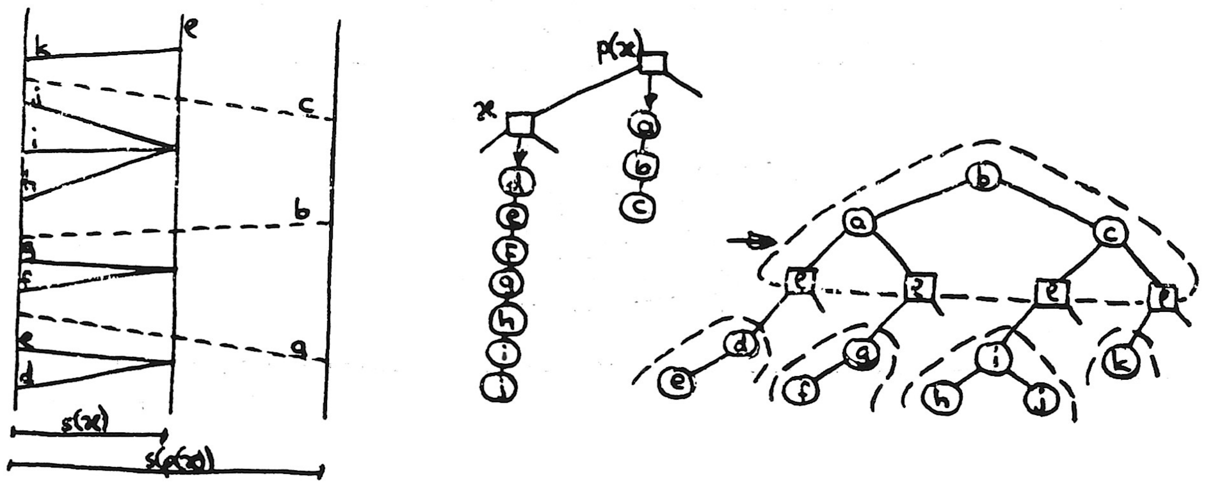

Figure 2.4: The slab of a node κ is decomposed by the line segments in $L(\kappa)$ and also by the line segments in $L(\mathrm{p}(\kappa))$. Assuming a, b, c fall into the same gap defined by $\mathrm{p}(\mathrm{p}(\kappa))$, the tree portions on the right illustrate the original segment tree, and the modified tree, T. Circles indicate nodes storing line segments, and squares indicate nodes storing vertical lines.

and identifies two line segments between which the query point, x, lies. The two line segments also identify a subsequence of $L(\kappa)$ that need to be searched. Accordingly, decompose $L(\kappa)$ into several lists, each associated with a gap between two line segments in $L(\mathrm{p}(\kappa))$, or between $-\infty$ and the lowest or $+\infty$ and the highest line segment in $L(\mathrm{p}(\kappa))$. Store each (new) list in a binary tree that supports binary search. We label the lists somehow and denote the tree that stores the ith list by T_i. Each internal node of T_i stores a line segment, and each leaf, λ, corresponds to a gap. For each gap there are two trees storing pieces of $L(\mathrm{l}(\kappa))$ and $L(\mathrm{r}(\kappa))$. The roots of these trees are made the left and right children of λ.

Now we have a binary tree T, composed of many small trees, the T_i, with nodes of two types. Each interior node of a T_i stores a line segment, and the search branches depending on whether the query point, x, lies below or above this line segment. Each leaf of a T_i stores a vertical line, and the search branches depending on whether x lies to the left or the right of this line.

Weighted binary search trees. So far, the query time did not necessarily improve yet. It does improve to $O(\log n)$ if weighted binary search trees are used for the T_i. Suppose the k leaves of a sorted binary tree, T_i, have positive real weights, $w(\lambda_1), w(\lambda_2), \ldots, w(\lambda_k)$. Define $w(T_i) = \sum_{j=1}^{k} w(\lambda_j)$. Then T_i is a *weighted binary search tree* for the assigned weights if there is a constant c so that the depth of λ_j is at most $c + \log_2 \frac{w(T_i)}{w(\lambda_j)}$, for all j. Such a tree can be constructed in time $O(k)$, see [6, chapter III.4].

We set the weight of a leaf λ of any T_i equal to the number of descendents it has in T. If λ is a leaf of T then it has one descendent, namely itself, and $w(\lambda) = 1$. Recall that the total number of non-leaf nodes of all T_i is no more than $(2 + 2\log_2 n) \cdot n$, see property (iv) in section 1. Each T_i has one more leaf than non-leaf, and there are at most $4n + 3$ such trees. The total number of nodes and an upper bound on the largest weight is therefore $4n\log_2 n + 8n + 3$. A search path goes through $h \leq \log_2 n + 3$ trees T_i. Let the trees along the path be indexed as T_1 through T_h and let λ_j be the leaf in T_j that lies on the path. Note that λ_{j-1} is the parent of the root of T_j and $w(T_j) \leq w(\lambda_{j-1})$. The length of the path is therefore at most

$$\sum_{j=1}^{h}\left(c + \log_2 \frac{w(T_j)}{w(\lambda_j)}\right) = c \cdot h + \log_2 w(T_1) - \log_2 w(\lambda_h) - \sum_{j=1}^{h-1}[w(\lambda_j) - w(T_{j+1})]$$

$$\begin{aligned} &\leq c(\log_2 n + 3) + \log_2(4n \log_2 n + 8n + 3) \\ &\leq (c+1)\log_2 n + O(\log\log n). \end{aligned}$$

Summary. The planar point location problem can be solved with the (modified) segment tree, which takes $O(n \log n)$ storage and construction time, so that a query can be answered in time $O(\log n)$.

Homework exercises

2.1 Prove that relations $\prec$ and $\prec_s$ defined above are acyclic.
(Remark. Acyclicity is essential because otherwise the transitive closure and linear extension would not exist.)

2.2 Give an algorithm that takes time $O(n \log n)$ to partition the lists L into the pieces required to improve the time for a point location query to $O(\log n)$.
(Hint. Use the lists H described in exercise 1.1.)

References

[1] T. H. Cormen, C. E. Leiserson and R. L. Rivest. *Introduction to Algorithms.* MIT Press, Cambridge, 1990.

[2] H. Edelsbrunner, L. J. Guibas and J. Stolfi. Optimal point location in a monotone subdivision. *SIAM J. Comput.* **15** (1986), 317–340.

[3] D. G. Kirkpatrick. Optimal search in planar subdivisions. *SIAM J. Comput.* **12** (1983), 28–35.

[4] D. E. Knuth. *Fundamental Algorithms. The Art of Computer Programming, Vol. 1.* Addison-Wesley, Reading, 1968.

[5] R. J. Lipton and R. E. Tarjan. Applications of a planar separator theorem. *In* "Proc. 18th IEEE Ann. Sympos. Found. Comput. Sci. 1977", 162–170.

[6] K. Mehlhorn. *Data Structures and Algorithms 1: Sorting and Searching.* Springer-Verlag, Heidelberg, 1984.

[7] F. P. Preparata. A new approach to planar point location. *SIAM J. Comput.* **10** (1981), 473–482.