Chapter 5

Randomized algorithms

A *randomized algorithm* is an algorithm that makes random choices during its execution. A randomized algorithm solves a deterministic problem and, whatever the random choices are, always runs in a finite time and outputs the correct solution to the problem. Therefore, only the path that the algorithm chooses to follow to reach the solution is random: the solution is always the same. Most randomized methods lead to conceptually simple algorithms, which often yield a better performance than their deterministic counterparts. This explains the success encountered by these methods and the important position they are granted in this book. The time and space used when running a randomized algorithm depend both on the input set and on the random choices. The performances of such an algorithm are thus analyzed on the average over all possible random choices made by the algorithm, yet in the worst case for the input. Randomization becomes interesting when this average complexity is smaller than the worst-case complexity of deterministic algorithms that solve the same problem.

The randomized algorithms described in this chapter, and more generally encountered in this book, use the randomized incremental method. The incremental resolution of a problem consists of two stages: first, the solution for a small subset of the data is computed, then the remaining input objects are inserted while the current solution is maintained. An incremental algorithm is said to be randomized if the data are inserted in a deliberately random order. For instance, sorting by insertion can be considered as a randomized incremental method: the current element, randomly chosen among the remaining ones, is inserted into the already sorted list of previously inserted elements.

Some incremental algorithms can work *on line*: they do not require prior knowledge of the whole set of data. Rather, these algorithms maintain the solution to the problem as the input data are successively inserted, without looking ahead at the objects that remain to be inserted. We refer to such algorithms as *on-line* or *semi-dynamic* algorithms. The order in which the data are inserted is imposed

on the algorithm, and such algorithms cannot properly be called randomized, as their behavior is purely deterministic. Nevertheless, we may be interested in the behavior of such an algorithm when the insertion order is assumed to be random. We may then speak of the *randomized analysis* of an on-line algorithm.

In the first section of this chapter, the randomized incremental method is sketched in the framework introduced in the previous chapter, with objects, regions, and conflicts. The underlying probabilistic model is made clear. At any step, a randomized incremental algorithm must detect the conflicts between the current object and the regions created previously. One way of detecting these conflicts is to use a *conflict graph*. Algorithms using a conflict graph must have a global knowledge of the input and are thus *off-line*. Another way is to use an *influence graph*. This structure leads to semi-dynamic algorithms and allows the objects to be inserted on-line. The conflict graph is described in section 5.2 and the influence graph in section 5.3. In both cases, the method is illustrated by an algorithm that builds the *vertical decomposition* of a set of line segments in the plane. This planar map was introduced in section 3.3, and in particular one can deduce from it all the intersecting pairs of segments. Both methods lead to a randomized algorithm that runs in time $O(n \log n + a)$ on the average, where a is the number of intersecting pairs of segments, and this is optimal. Finally, section 5.4 shows how both methods may be combined and, sometimes, lead to *accelerated* algorithms. For instance, we show how to decompose the set of line segments forming the boundary of a simple polygon in time $O(n \log^* n)$ on the average (provided that the order of the edges along the boundary of the polygon is also part of the input).

We give several randomized incremental algorithms, for instance to compute convex hulls (chapter 8), to solve linear programs (chapter 10), to compute the lower envelope of a set of segments in the plane (chapter 15) or of triangles in three-dimensional space (chapter 16), or even to compute the k-level of an arrangement of hyperplanes (chapter 14) or a cell in an arrangement of segments (chapter 15) or of triangles (chapter 16).

5.1 The randomized incremental method

The problem to be solved is formulated in terms of objects, regions, and conflicts in the general framework described in the previous chapter. The problem now becomes that of finding the set $\mathcal{F}_0(\mathcal{S})$ of regions defined and without conflict over a finite set $\mathcal{S}$ of objects. The notation used in this chapter is that defined in subsections 4.1.1 and 4.1.2.

The initial step in the incremental method builds the set $\mathcal{F}_0(\mathcal{R}_0)$ of regions that are defined and without conflict over a small subset $\mathcal{R}_0$ of $\mathcal{S}$. Each subsequent step consists of processing an object of $\mathcal{S} \setminus \mathcal{R}_0$. Let $\mathcal{R}$ be the set of already

processed objects and let us call *step* r the step during which we process the r-th object.

Let O be the object processed in step r. From the already computed set of regions defined and without conflict over $\mathcal{R}$, we compute in step r the set of regions defined and without conflict over $\mathcal{R} \cup \{O\}$.

- The regions of $\mathcal{F}_0(\mathcal{R})$ that do not belong to $\mathcal{F}_0(\mathcal{R} \cup \{O\})$ are exactly those regions in $\mathcal{F}_0(\mathcal{R})$ that conflict with O. These regions are said to be *killed* by O, and O is their *killer*.
- The regions of $\mathcal{F}_0(\mathcal{R} \cup \{O\})$ that do not belong to $\mathcal{F}_0(\mathcal{R})$ are exactly those regions $\mathcal{F}_0(\mathcal{R} \cup \{O\})$ that are determined by a subset of $\mathcal{R} \cup \{O\}$ that contains O. These regions are said to be *created* by O.

A region created by O during step r is said to be *created at step* r. The set of regions created by an incremental algorithm consists of all the regions created during the initial step or at one of the subsequent insertion steps.

The *chronological sequence* is the sequence of the objects of $\mathcal{S}$ in the order in which they are processed. The probabilistic assumption on which the randomized analysis of incremental algorithms relies is that the chronological sequence is any of the $n!$ possible permutations of the objects of $\mathcal{S}$ with uniform probability. As a consequence, the subset of objects $\mathcal{R}$ already processed at step r is a random r-sample of $\mathcal{S}$, and any subset of $\mathcal{S}$ is equally likely. The object O processed during step r is a random element of $\mathcal{S} \setminus \mathcal{R}$. Equivalently, it is any element of $\mathcal{R} \cup \{O\}$ with a uniform probability.

5.2 Off-line algorithms

5.2.1 The conflict graph

The first task of each step in an incremental algorithm is to detect the regions of $\mathcal{F}_0(\mathcal{R})$ that conflict with the object O to be processed at this step. These are the regions killed by O. To check all the regions in $\mathcal{F}_0(\mathcal{R})$ does not lead to efficient algorithms. Instead, in addition to the set $\mathcal{F}_0(\mathcal{R})$, we maintain a graph called the *conflict graph*. The conflict graph is a bipartite graph between the objects of $\mathcal{S} \setminus \mathcal{R}$ and the regions of $\mathcal{F}_0(\mathcal{R})$. Arcs belong to $\mathcal{F}_0(\mathcal{R}) \times (\mathcal{S} \setminus \mathcal{R})$, and are called the *conflict arcs*. There is a conflict arc (F, S) exactly for each region F of $\mathcal{F}_0(\mathcal{R})$ and each object S of $\mathcal{S} \setminus \mathcal{R}$ that conflicts with F.

The conflict graph allows us to find all the regions killed by an object O in time proportional to the number of those regions. Each step of the incremental algorithm must then update the conflict graph. The conflict arcs incident to the regions killed by O are removed and the new conflict arcs incident to the regions

created by O are found. The complexity of each incremental step is thus at least bounded from below by the number of regions that are killed or created during this step, and by the number of conflict arcs that are removed or added during this step.

Update condition 5.2.1 (for conflict graphs) *A randomized incremental algorithm that uses a conflict graph satisfies the* update condition *if, at each incremental step:*

1. *updating the set of regions defined and without conflict over the current subset can be carried out in time proportional to the number of regions killed or created during this step, and*

2. *updating the conflict graph can be carried out in time proportional to the number of conflict arcs added or removed during this step.*

Lemma 5.2.2 *Let S be a set of n objects and F a region determined by i objects of S that has j conflicts with the objects in S.*

1. *The probability p'^i_j that F be one of the regions created by a randomized incremental algorithm processing S is*

$$p'^i_j = \frac{i!j!}{(i+j)!}.$$

2. *The probability $p'^i_j(r)$ that F be one of the regions created by the algorithm during step r is*

$$p'^i_j(r) = \frac{i}{r} p^i_j(r).$$

In these expressions, $p^i_j(r)$ stands for the probability that a region F of $\mathcal{F}^i_j(S)$ be defined and without conflict over a random r-sample of S, as given in subsection 4.2.1.

If we replace $p^i_j(r)$ by its expression in lemma 4.2.1, we obtain (see also exercise 5.1) that the probabilities p'^i_j and $p'^i_j(r)$ satisfy the relation

$$p'^i_j = \sum_{r=1}^{n} p'^i_j(r).$$

Proof. A region F of $\mathcal{F}^i_j(S)$ is created by a randomized incremental algorithm if and only if, during some step of the algorithm, this region is defined and without conflict over the current subset. Such a situation occurs when the i objects that

determine F are processed before any of the j objects of $\mathcal{S}$ that conflict with F. Since all permutations of these objects are equally likely, this case happens with probability

$$\frac{i!j!}{(i+j)!},$$

proving the first part of the lemma. Let $\mathcal{R}$ be the set of objects processed in the steps preceding and including step r. For a region F to be created during step r, we first require that F be defined and without conflict over $\mathcal{R}$, which happens precisely with probability $p_j^i(r)$. If so, F is created at step r precisely if the object O processed during step r is one of the i objects of $\mathcal{R}$ that determine F. This happens with conditional probability i/r. □

Theorem 5.2.3 (Conflict graph) *Let $\mathcal{S}$ be a set of n objects, and consider a randomized incremental algorithm that uses a conflict graph to process $\mathcal{S}$.*

1. *The expected total number of regions created by the algorithm is*

$$O\left(\sum_{r=1}^{n} \frac{f_0(r,\mathcal{S})}{r}\right).$$

2. *The expected total number of conflict arcs added to the conflict graph by the algorithm is*

$$O\left(n\sum_{r=1}^{n} \frac{f_0(r,\mathcal{S})}{r^2}\right).$$

3. *If the algorithm satisfies the update condition, then its complexity (both in time and in space) is, on the average,*

$$O\left(n\sum_{r=1}^{n} \frac{f_0(r,\mathcal{S})}{r^2}\right).$$

In these expressions, $f_0(r,\mathcal{S})$ denotes the expected number of regions defined and without conflict over a random r-sample of $\mathcal{S}$.

Thus, if $f_0(r,\mathcal{S})$ behaves linearly with respect to r ($f_0(r,\mathcal{S}) = O(r)$), the total number of created regions is $O(n)$ on the average, the total number of conflict arcs is $O(n \log n)$ on the average, and the complexity of the algorithm is $O(n \log n)$ on the average. If the growth of $f_0(r,\mathcal{S})$ is super-linear with respect to r ($f_0(r,\mathcal{S}) = O(r^\alpha)$ for some $\alpha > 1$), then the total number of created regions is $O(n^\alpha)$ on the average, the total number of conflict arc is $O(n^\alpha)$ on the average, and the complexity of the algorithm is $O(n^\alpha)$ on the average.

Proof.

1. We obtain the expectation $v(\mathcal{S})$ of the total number of regions created by the algorithm by summing, over all regions F defined over $\mathcal{S}$, the probability that this region F be created by the algorithm:

$$v(\mathcal{S}) = \sum_{i=1}^{b} \sum_{j=0}^{n-i} |\mathcal{F}_j^i(\mathcal{S})|\, p'^i_j = \sum_{i=1}^{b} \sum_{j=0}^{n-i} \sum_{r=1}^{n} |\mathcal{F}_j^i(\mathcal{S})|\, \frac{i}{r}\, p_j^i(r).$$

In this expression, we recognize the expected number of regions defined and without conflict over a random r-sample of $\mathcal{S}$ (lemma 4.2.2), so we get

$$v(\mathcal{S}) = O\left(\sum_{r=1}^{n} \frac{f_0(r,\mathcal{S})}{r} \right).$$

2. Let $e(\mathcal{S})$ be the expected total number of arcs added to the conflict graph. To estimate $e(\mathcal{S})$, we note that if a region F in conflict with j objects of $\mathcal{S}$ is a region created by the algorithm, then it is adjacent to j conflict arcs in the graph. Therefore,

$$e(\mathcal{S}) = \sum_{i=1}^{b} \sum_{j=0}^{n-i} |\mathcal{F}_j^i(\mathcal{S})|\, j\, p'^i_j, = \sum_{i=1}^{b} \sum_{j=0}^{n-i} \sum_{r=1}^{n} |\mathcal{F}_j^i(\mathcal{S})|\, \frac{ij}{r}\, p_j^i(r).$$

Apart from the factor i/r, we recognize in this expression the moment of order 1 of a random r-sample (lemma 4.2.5). Applying corollary 4.2.7 to the moment theorem, we get

$$\begin{aligned} e(\mathcal{S}) &\leq b \sum_{r=1}^{n} \frac{m_1(r,\mathcal{S})}{r} \\ &= O\left(\sum_{r=1}^{n} \frac{n}{r^2} f_0(\lfloor r/2 \rfloor, \mathcal{S}) \right) = O\left(n \sum_{r=1}^{n} \frac{f_0(r,\mathcal{S})}{r^2} \right). \end{aligned}$$

3. A given region is killed or created at most once during the course of the algorithm and, likewise, a given conflict arc is added or removed at most once. A randomized incremental algorithm which satisfies the update condition thus has an average complexity of at most

$$v(\mathcal{S}) + e(\mathcal{S}) = O\left(n \sum_{r=1}^{n} \frac{f_0(r,\mathcal{S})}{r^2} \right). \qquad \square$$

Exercise 5.2 presents a non-amortized analysis of each step of an algorithm that uses a conflict graph.

5.2.2 An example: vertical decomposition of line segments

Let us discuss again the problem of finding all the a intersecting pairs of a set of n line segments in the plane. The space sweep algorithm presented in chapter 3 solves this problem in time $O((n+a)\log n)$, which is suboptimal because a could be as large as $\Omega(n^2)$. The complexity of the problem is $O(n\log n + a)$, and is matched by a deterministic algorithm (see section 3.5). The algorithm we present here is randomized incremental, and its expected complexity also matches the optimal bound. It is much simpler than the deterministic algorithm, and generalizes easily to the case of curved segments (see exercise 5.5).

This algorithm builds in fact the vertical decomposition of the set of segments $\mathcal{S}$, or more precisely its simplified decomposition $\mathcal{D}ec_s(\mathcal{S})$. This structure is discussed in detail in section 3.3. Let us simply recall that the vertical decomposition of $\mathcal{S}$ is induced by the segments of $\mathcal{S}$ and the vertical walls stemming from their endpoints and intersection points (see figure 3.2 or figure 5.4 below). The decomposition of the set $\mathcal{S}$ of segments with a intersecting pairs is of size $O(n+a)$, and can be used to report all the intersecting pairs of $\mathcal{S}$ in the same time bound $O(n+a)$.

To use the preceding framework, we define the set of objects $\mathcal{O}$ to be the set of all segments in the plane, the universe of regions $\mathcal{F}$ being the set of all possible trapezoids with parallel vertical sides, occasionally in some degenerate state such as triangles, semi-infinite trapezoids, vertical slabs, or half-planes bounded by a vertical line. A trapezoid F of $\mathcal{F}$ conflicts with a segment S if this segment intersects the interior of F. A trapezoid F is determined by a minimal set of segments $\mathcal{X}$, the decomposition of which includes F. Each region in the simplified decomposition $\mathcal{D}ec_s(\mathcal{S})$ is described using at most four segments of $\mathcal{S}$. Indeed, the floor and ceiling are each supported by a single segment in $\mathcal{S}$, and each vertical side consists of one or two walls stemming either from an endpoint of a segment, or from an intersection of two segments. In the former case, the vertical side consists either of one vertical wall stemming upward from an endpoint of the floor, or stemming downward from an endpoint of the ceiling, or of two vertical walls stemming in both directions from the endpoint of another segment. In the latter case, one of the intersecting segments is necessarily supporting the floor or ceiling of the trapezoid. In all cases, we see that four segments suffice to determine the trapezoid. Three may suffice if the trapezoid is a degenerate triangle. Partially infinite triangles may be determined by three, two, or even one segment. The cardinality of the set $\mathcal{X}$ that determines a region is thus at most four, and the constant b that upper-bounds the number of objects that determine a region is thus set to $b = 4$.

Therefore, the set of trapezoids in the vertical decomposition of the set $\mathcal{S}$ of segments is exactly the set of those segments defined and without conflict over $\mathcal{S}$. According to our notation, this set is denoted by $\mathcal{F}_0(\mathcal{S})$.

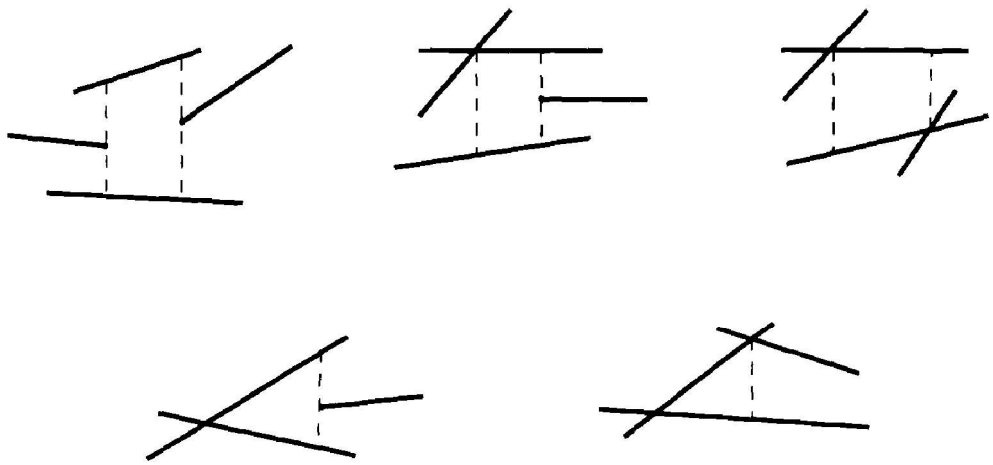

Figure 5.1. Instances of trapezoids in the simplified vertical decomposition.

The algorithm

According to the randomized incremental scheme, the algorithm processes the segments of $\mathcal{S}$ one by one in a random order. Again, we denote by $\mathcal{R}$ the set of the already processed segments at a given step r. The algorithm maintains the simplified vertical decomposition $\mathcal{D}ec_s(\mathcal{R})$ of this set of segments, as well as the corresponding conflict graph.

The decomposition $\mathcal{D}ec_s(\mathcal{R})$ is maintained in a data structure that encodes its simplified description. This structure includes the list of all trapezoids of $\mathcal{D}ec_s(\mathcal{R})$. Each trapezoid is described by the set of at most four segments that determine it, and by the at most six edges in the simplified decomposition (floor, ceiling, and at most four vertical walls) which bound it. The data structure also describes the vertical adjacency relationships in the decomposition, that is the pairs of trapezoids whose boundaries share a vertical wall. Such an adjacency is stored in a bidirectional pointer linking both trapezoids. Note that each trapezoid is vertically adjacent to at most four trapezoids.

The conflict graph has a conflict arc for each pair (F, S) of a trapezoid F of $\mathcal{D}ec_s(\mathcal{R})$ and a segment S of $\mathcal{S} \setminus \mathcal{R}$ that intersects the interior of F. The conflict graph is implemented by a system of interconnected lists.

- For each segment S of $\mathcal{S} \setminus \mathcal{R}$, the data structure stores a list $\mathcal{L}(S)$ representing the set of trapezoids of $\mathcal{D}ec_s(\mathcal{R})$ intersected by S. The list $\mathcal{L}(S)$ is ordered according to which trapezoids are encountered as we slide along S from left to right.

- For each trapezoid F of $\mathcal{D}ec_s(\mathcal{R})$, the algorithm maintains the list $\mathcal{L}'(F)$ of the segments in $\mathcal{S} \setminus \mathcal{R}$ that conflict with F.

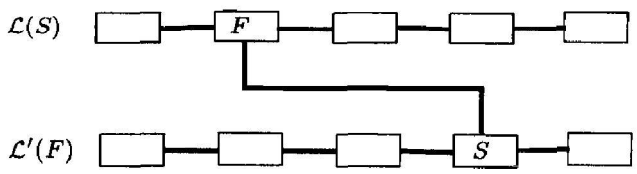

Figure 5.2. Representation of the conflict graph used to build the vertical decomposition of a set of line segments.

- The node that stores a segment S in the list $\mathcal{L}'(F)$ of a trapezoid F is interconnected via a bidirectional pointer with the node that stores the trapezoid F in the list $\mathcal{L}(S)$ of the segment S (see figure 5.2).

In the *initial step*, the algorithm builds the decomposition $\mathcal{D}ec_s(\mathcal{R})$ for a subset $\mathcal{R}$ of $\mathcal{S}$ that contains only a single segment. This decomposition consists of four trapezoids. It also initializes the lists that represent the conflict graph. The initial decomposition is built in constant time, and the initial conflict graph in linear time.

The *current step* processes a new segment S of $\mathcal{S} \setminus \mathcal{R}$: it updates the decomposition and the conflict graph accordingly.

Updating the decomposition. The conflict graph gives the list $\mathcal{L}(S)$ of all the trapezoids of $\mathcal{D}ec_s(\mathcal{R})$ that are intersected by S. Each trapezoid is split into at most four *subregions* by the segment S, the walls stemming from the endpoints of S, and the walls stemming from the intersection points between S and the other segments in $\mathcal{R}$ (see figure 5.3).

These subregions are not necessarily trapezoids of $\mathcal{D}ec_s(\mathcal{R} \cup \{S\})$. Indeed, S intersects some vertical walls of $\mathcal{D}ec_s(\mathcal{R})$, and any such wall must be shortened: the portion of this wall that contains no endpoint or intersection point must be removed from the decomposition, and the two subregions that share this portion of the wall must be joined into a new trapezoid of $\mathcal{D}ec_s(\mathcal{R} \cup \{S\})$ (see figure 5.4). Thus, any trapezoid of $\mathcal{D}ec_s(\mathcal{R} \cup \{S\})$ created by S is either a subregion, or the union of a maximal subset of subregions that can be ordered so that two consecutive subregions share a portion of a wall to be removed. The vertical adjacency relationships in the decomposition that concern trapezoids created by S can be inferred from the vertical adjacency relationships between the subregions and from those between the trapezoids of $\mathcal{D}ec_s(\mathcal{R})$ that conflict with S.

Updating the data structure that represents the decomposition $\mathcal{D}ec_s(\mathcal{R})$ can therefore be carried out in time linear in the number of trapezoids conflicting with S.

Updating the conflict graph. When a trapezoid F is split into subregions F_i ($i \leq 4$), the list $\mathcal{L}'(F)$ of segments that conflict with F is traversed linearly, and a conflict list $\mathcal{L}'(F_i)$ is set up for each of the subregions F_i. During this

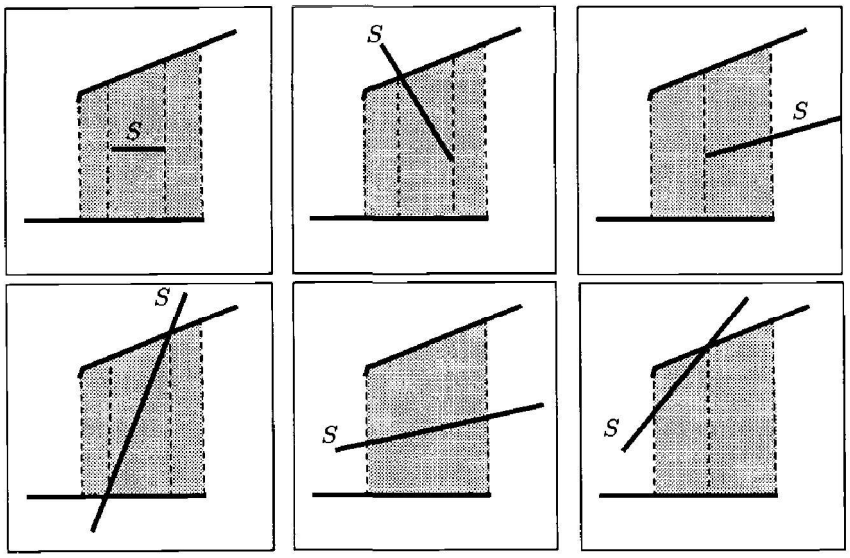

Figure 5.3. Decomposing a set of segments: splitting a trapezoid into at most four new trapezoids.

traversal, the list $\mathcal{L}(S')$ of each segment S' in $\mathcal{L}'(F)$ is updated as follows: each node pointing to F in such a list is replaced by the sequence of those subregions F_i that intersect S', in the left-to-right order along S'.

Consider now a sequence $F_1, F_2, \ldots, F_k$ of subregions that have to be joined to yield a trapezoid F' of $\mathcal{D}ec_s(\mathcal{R} \cup \{S\})$ created by S. We assume that the subregions are encountered in this order along S. To build the list $\mathcal{L}'(F')$, we must merge the lists $\mathcal{L}'(F_i)$ while at the same time removing redundant elements. To do this, we traverse successively each of the lists $\mathcal{L}'(F_i)$. For each segment S' that we encounter in $\mathcal{L}'(F_i)$, we obtain the entry corresponding to F_i in the list $\mathcal{L}(S')$ by following the bidirectional pointer in the entry corresponding to S' in the list $\mathcal{L}'(F_i)$. The subregions that conflict with S' and that have to be joined are consecutive in the list $\mathcal{L}(S')$. The nodes that correspond to these regions are removed from the list $\mathcal{L}(S')$, and for each entry F_j removed from the list $\mathcal{L}(S')$, the corresponding entry for S' in $\mathcal{L}'(F_j)$ is also removed. (This process is illustrated in figure 5.5.) In this fashion, we merge the conflict lists of a set of adjacent subregions while visiting each node of the conflict lists of these subregions once and only once. Similarly, the corresponding nodes in the conflict lists of the segments are visited once and only once. This ensures that the time taken to update the conflict graph is linear in the number of arcs of the graph that have to be removed.

Analysis of the algorithm

The preceding discussion shows that the algorithm which computes the vertical decomposition of a set of line segments using a conflict graph obeys the update

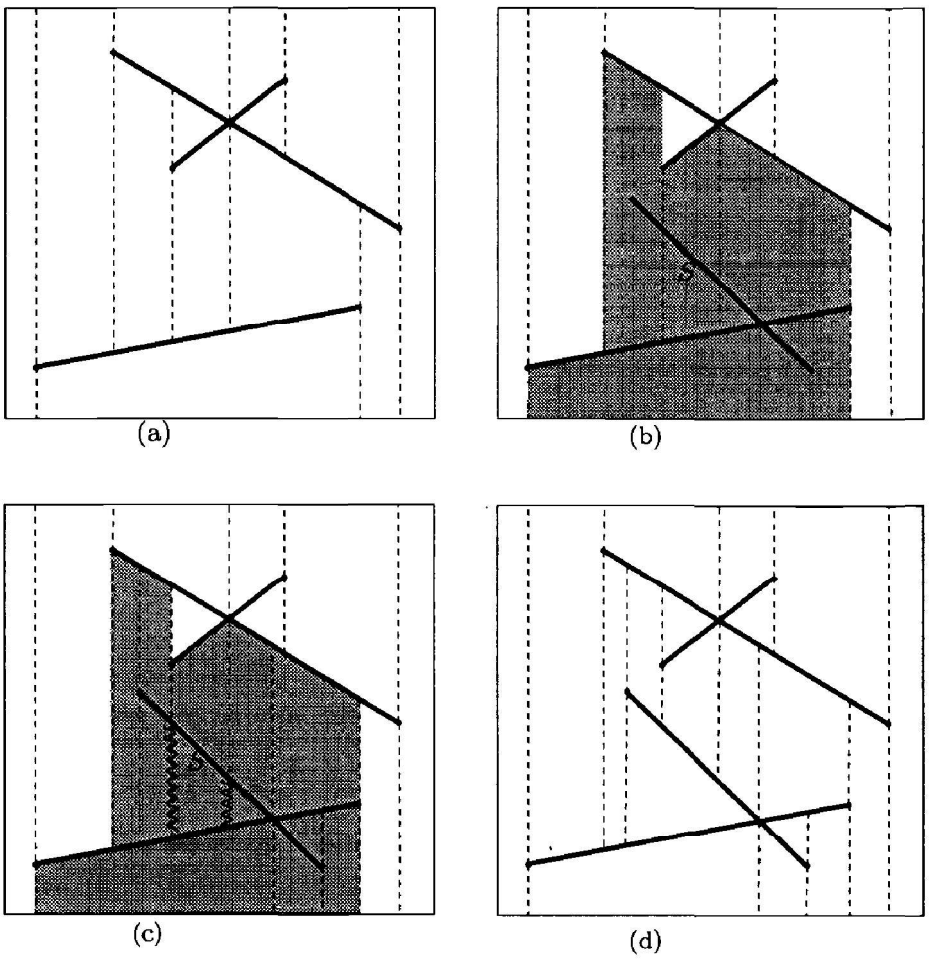

Figure 5.4. Decomposing a set of segments: the incremental construction
(a) The decomposition before inserting segment S.
(b) Shaded, the trapezoids that conflict with segment S.
(c) Splitting those trapezoids. Wavy lines indicate the portions of walls to be removed.
(d) The decomposition after inserting segment S.

condition 5.2.1. We may therefore quote theorem 5.2.3 to show that the average running time of the algorithm, given a set of n segments with a intersection points, is

$$O\left(n \sum_{1 \le r \le n} \frac{f_0(r, \mathcal{S})}{r^2}\right).$$

Here, $f_0(r, \mathcal{S})$ is the expected number of trapezoids in the vertical decomposition of a random r-sample of $\mathcal{S}$. The following lemma estimates this number.

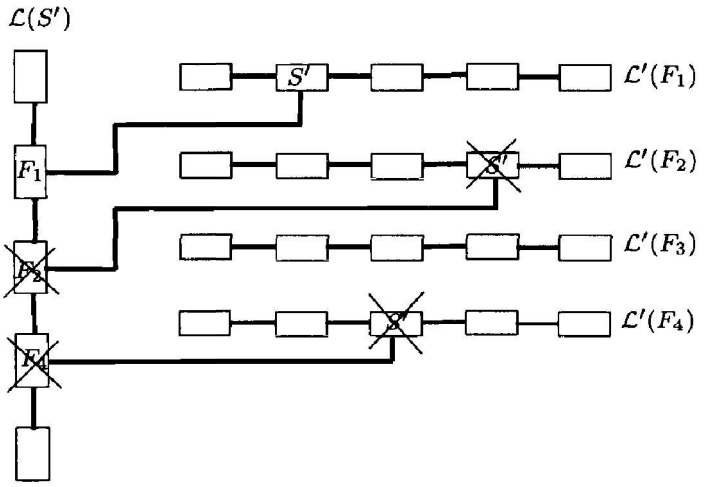

Figure 5.5. Decomposing a set of line segments: merging the conflict lists.

Lemma 5.2.4 *Let $\mathcal{S}$ be a set of n segments, a pairs of which have a non-empty intersection. The expected number $f_0(r, \mathcal{S})$ of trapezoids in the vertical decomposition of a random r-sample of $\mathcal{S}$ is $O(r + ar^2/n^2)$.*

Proof. Let $\mathcal{R}$ be a subset of r segments in $\mathcal{S}$, and denote by $a(\mathcal{R})$ the number of intersecting pairs of segments in $\mathcal{R}$. The number of regions in the vertical decomposition of $\mathcal{R}$ is therefore $O(r + a(\mathcal{R}))$. If $\mathcal{R}$ is a random r-sample of $\mathcal{S}$, however, the expected number $a(\mathcal{R})$ of intersections is $\frac{ar(r-1)}{n(n-1)}$. Indeed, an intersection point P between two segments of $\mathcal{S}$ is an intersection point of two segments of $\mathcal{R}$ if the two segments of $\mathcal{S}$ that intersect at P belong to $\mathcal{R}$, which happens with probability

$$\binom{n-2}{r-2} \Big/ \binom{n}{r}.$$

□

Therefore, we immediately derive the following theorem.

Theorem 5.2.5 *The vertical decomposition of a set $\mathcal{S}$ of n line segments in the plane with a intersecting pairs can be obtained by a randomized incremental algorithm that uses a conflict graph, in expected time $O(n \log n + a)$.*

Remark 1. The algorithm builds only the *simplified* representation $\mathcal{D}ec_s(\mathcal{S})$ of the decomposition. From this representation, however, we can derive the complete representation $\mathcal{D}ec(\mathcal{S})$ of the decomposition in $O(n + a)$ time. Exercise 5.3 shows that the complete representation can be directly computed by a variant of the preceding algorithm, while still using no more than $O(n \log n + a)$ running time.

Remark 2. The expected storage of the algorithm is $O(n \log n + a)$. In the variant mentioned in the above remark, it is possible to simplify the conflict graph: for each segment, we retain only a single conflict arc, for instance the conflict with the trapezoid which contains the left endpoint of the segment. We can still update the conflict graph in linear time, therefore the average running time is unchanged and still $O(n \log n + a)$, but the expected storage is lowered to $O(n + a)$ (see exercise 5.4).

5.3 On-line algorithms

Algorithms that use a conflict graph are incremental but static, that is, they require initial knowledge of all the segments to be inserted. In contrast, on-line (or semi-dynamic) algorithms maintain the solution to the problem as the input objects are inserted, with no preliminary knowledge of the input data. A possible way to transform an algorithm that uses a conflict graph into an on-line algorithm is to replace the conflict graph by a different kind of structure that can detect conflicts between *any* object and the regions defined and without conflict over the current set of objects. The *influence graph* is such a structure.

5.3.1 The influence graph

The influence graph is a structure that stores the history of the incremental construction and depends on the order in which the objects have been processed by the algorithm. This graph represents the regions created by the algorithm during the incremental construction, and can be used to detect the conflicts between these regions and a new object. When the algorithm uses a conflict graph, the set of data is known in advance, and the algorithm may then compute the objects in $\mathcal{S}$ that conflict with a given region. However, an on-line algorithm does not assume any knowledge of the objects to be processed. Thus it must be able to describe the entire domain of influence of a region which, as we recall, is the subset of all the objects in the universe that conflict with this region.

The influence graph is a directed, acyclic, and connected graph. It possesses a single root, and its nodes correspond to the regions created by the algorithm during its entire history. Therefore, a node corresponds to a region that was defined and without conflict over the current set of objects at some point during the execution of the algorithm. Properly speaking, this graph is not a tree: a node might have several parents. Nevertheless, the terminology of trees will be quite useful for describing it. In particular, a *leaf* is a node that has no children. The influence graph must possess two essential properties.

Property 5.3.1 *1. At each step of the algorithm, a region defined and without conflict over the current subset is associated with a leaf of the influence graph.*

2. The domain of influence of a region associated with a node of the influence graph is contained in the union of the domains of influence of the regions associated with the parents of that node.

To shorten and simplify the terminology, we attach qualifiers normally used for a node to its corresponding region and vice versa. This allows us to speak of the domain of influence of a node instead of the domain of influence of its associated region. Likewise, a region has children or parents. This slight abuse in the terminology should not create any confusion.

The algorithm

The algorithm is incremental and maintains the set $\mathcal{F}_0(\mathcal{R})$ of regions defined and without conflict over the current set $\mathcal{R}$, together with the influence graph corresponding to the chronological sequence of objects in $\mathcal{R}$.

The *initial step* processes a small set of r_0 objects. For instance, r_0 can be the minimal number of objects needed to determine a region. The algorithm computes the regions defined and without conflict over the set $\mathcal{R}_0$ of these r_0 objects. The influence graph is initialized by creating a root node, corresponding to a fictitious region whose influence domain is the universe $\mathcal{O}$ of objects in its entirety. A node whose parent is the root is created for each of the regions of $\mathcal{F}(\mathcal{R}_0)$.

In the *current step*, the object O is added to $\mathcal{R}$. The work can be divided into two phases: we first locate O and then update the data structures.

Locating. In this phase, we must find all the regions in $\mathcal{F}_0(\mathcal{R})$ that conflict with the new object O. Starting from the root of the influence graph, we recursively visit all the nodes that conflict with O, and their children. The regions of $\mathcal{F}_0(\mathcal{R})$ that conflict with O are said to be *killed* by O.

Updating. We now have to update the data structure that represents the set of those regions defined and without conflict over the current subset of objects ($\mathcal{F}_0(\mathcal{R})$ becomes $\mathcal{F}_0(\mathcal{R} \cup \{O\})$). We also have to update the influence graph. A leaf of the influence graph is created for each of the regions in $\mathcal{F}_0(\mathcal{R} \cup \{O\}) \setminus \mathcal{F}_0(\mathcal{R})$. These are the regions *created* by O. Each of these leaves is linked to enough parents to satisfy property 2 of the influence graph. We never remove a node from the graph.

The details of the implementation of these steps naturally depend on the problem. Typically, the set of regions created by O can be derived from the set of regions killed by O, and the parents of the leaves corresponding to created regions

may be chosen among the nodes corresponding to regions killed by O. In this case, the information needed to carry out the update is gathered in the locating phase. In some cases, we may need to know some extra information, such as adjacency relationships between regions. The influence graph is then augmented with the required information.

Randomized analysis of the influence graph

A randomized on-line algorithm is not a randomized algorithm properly speaking. Indeed, the order in which the data are processed is imposed on the algorithm, and the algorithm makes no random choices whatsoever. The algorithm is therefore perfectly deterministic. Nevertheless, we can analyze the average performance of such an algorithm (in running time or storage) under the assumption that all inputs are equally likely or, more precisely, that any permutation of the input data is equally likely. The performances of the algorithm are valid for any input, and in particular no assumption is made on the distribution of the input. In this case, the analysis of the algorithm is said to be a *randomized analysis.* The randomized analysis of an on-line algorithm which has currently processed a set $\mathcal{S}$ of n objects assumes that the chronological sequence Σ, made up from the objects of $\mathcal{S}$ in the processing order, is a random sequence.This means that all $n!$ permutations of $\mathcal{S}$ are equally likely to occur in Σ. At any step, the current subset of objects already processed by the algorithm is thus a random sample of $\mathcal{S}$.

Each node of the influence graph corresponds to a region created at some step of the algorithm. Such a region was, at this step, defined and without conflict over the current subset of objects. The set of those regions created by an incremental algorithm depends only on the order in which the objects are inserted. In particular, it does not depend on whether the incremental algorithm uses a conflict graph or an influence graph. The following theorem is thus an immediate consequence of theorem 5.2.3.

Theorem 5.3.2 *Let an on-line algorithm use an influence graph to process a set $\mathcal{S}$ of n objects. The expected number $v(\mathcal{S})$ of nodes in this influence graph is*

$$O\left(\sum_{r=1}^{n} \frac{f_0(r,\mathcal{S})}{r}\right).$$

In this expression, $f_0(r,\mathcal{S})$ denotes the expected number of regions defined and without conflict over a random r-sample of $\mathcal{S}$.

To carry the analysis further, we must also be able to bound the number of arcs in the influence graph, since this number gives the time and storage taken

to update the set of regions without conflict and the influence graph itself, as is done in the second phase of each insertion step of the algorithm. We also need a special assumption to control the complexity of testing whether there is a conflict between an object and a region.[1] The triple update condition stated below is actually satisfied by a large class of practical problems.

Update condition 5.3.3 (for influence graphs) *An on-line algorithm that uses an influence graph satisfies the* update condition *if:*

1. *the existence of a conflict between a given region and a given object can be tested in constant time.*
2. *the number of children of each node of the influence graph is bounded by a constant, and*
3. *the parents of a node created by an object O are nodes that are killed by O, and updating the influence graph takes time linear in the number of nodes killed or created at each step.*

Theorem 5.3.4 (Influence graph) *Consider an on-line algorithm that uses an influence graph to process a set S of n objects. If the algorithm satisfies the update condition 5.3.3, then:*

1. *The expected storage used by the algorithm to process the n objects is*
$$O\left(\sum_{r=1}^{n} \frac{f_0(r,\mathcal{S})}{r}\right).$$
2. *The expected time complexity of the algorithm is*
$$O\left(n\sum_{r=1}^{n} \frac{f_0(r,\mathcal{S})}{r^2}\right).$$
3. *The expected time complexity of the locating phase at step k is*
$$O\left(\sum_{r=1}^{k-1} \frac{f_0(r,\mathcal{S})}{r^2}\right).$$
4. *The expected time complexity of the updating phase at step k is*
$$O\left(\frac{f_0(k,\mathcal{S})}{k} + \frac{f_0(\lfloor (k-1)/2 \rfloor,\mathcal{S})}{k-1}\right).$$

[1] Note that such an assumption is implicitly contained in the update condition 5.2.1 when the algorithm uses a conflict graph.

As always, $f_0(r, \mathcal{S})$ denotes the expected number of regions defined and without conflict over a random r-sample of $\mathcal{S}$.

Thus, the expected time complexity of an on-line algorithm that uses an influence graph is identical to that of a similar incremental algorithm that uses a conflict graph, as long as the respective update conditions are satisfied.

If $f_0(r, \mathcal{S})$ behaves linearly with respect to r ($f_0(r, \mathcal{S}) = O(r)$), the complexity of the algorithm is $O(n \log n)$ on the average, and the expected storage is $O(n)$. Introducing the n-th object takes time $O(\log n)$ for the locating phase, and constant time for updating the data structure and the influence graph.

If the growth of $f_0(r, \mathcal{S})$ is super-linear with respect to r ($f_0(r, \mathcal{S}) = O(r^\alpha)$ for some $\alpha > 1$), the expected storage is $O(n^\alpha)$. Introducing the n-th object takes time $O(n^{\alpha-1})$ for the locating and updating phases.

Proof.

1. The upper bound on the expected storage is a direct consequence of theorem 5.3.2, which bounds the number of nodes in the influence graph, and of the second clause in the update condition, which bounds the number of children of each node.

2. The contribution to the running time complexity of the updating phases is proportional to the number of regions created, because of the third clause of the update condition. From theorem 5.2.3, we know that this number is

$$O\left(\sum_{r=1}^{n} \frac{f_0(r, \mathcal{S})}{r}\right).$$

We still must evaluate the cost of the locating phases. From the first clause of the update condition, we derive that the complexity of locating an object O is proportional to the number of nodes visited to locate O. If every child has a constant number of descendants (second clause in the update condition), however, the number of nodes visited during the locating phase of O is at most proportional to the nodes of the influence graph that conflict with O. The overall cost of the locating phases is therefore proportional to the total number of conflicts detected during the algorithm.

Let F be a region of $\mathcal{F}_j^i(\mathcal{S})$. If this region is created at some step of the algorithm, the corresponding node in the influence graph will be visited j times in the subsequent steps, and this happens each time we insert one of the j objects that conflict with F. For a given permutation of the input, an algorithm that uses an influence graph will not only create the same regions as another that uses a conflict graph, but will also detect a conflict with a given region as many times as there are conflict arcs incident to this region in the conflict graph.

As a consequence, the total expected complexity of the locating phases is proportional to the expected number of conflict arcs created in the conflict graph, and is given by theorem 5.2.3.

3. Finally, we can give a non-amortized analysis of each incremental step of an on-line algorithm. At step k, the locating phase takes time proportional to the number of nodes in the influence graph that conflict with O_k, the object introduced in this step. The average complexity of this locating phase during step k is thus proportional to $w(k, \mathcal{S})$, the expected number of nodes in the influence graph that conflict with O_k. From lemma 5.2.2, we know that a region F in $\mathcal{F}_j^i(\mathcal{S})$ is created at step r with a probability $p_j'^i(r) = \frac{i}{r} p_j^i(r)$. The conditional probability that this region conflict with O_k, knowing that F is created prior to step k, is $j/(n-r)$. Consequently,

$$w(k, \mathcal{S}) = \sum_{i=1}^{b} \sum_{j=0}^{n-i} |\mathcal{F}_j^i(\mathcal{S})| \sum_{r=1}^{k-1} \frac{i}{r} p_j^i(r) \frac{j}{n-r}.$$

If we recognize the expression for the first order moment of a random r-sample of $\mathcal{S}$ given in lemma 4.2.5, and bound the sum above by using corollary 4.2.7 to the moment theorem, we obtain

$$\begin{aligned} w(k, \mathcal{S}) &= \sum_{i=1}^{b} \sum_{r=1}^{k-1} \frac{i}{r(n-r)} m_1(r, \mathcal{S}) \\ &= O\left(\sum_{r=1}^{k-1} \frac{f_0(\lfloor r/2 \rfloor, \mathcal{S})}{r^2} \right) = O\left(\sum_{r=1}^{k-1} \frac{f_0(r, \mathcal{S})}{r^2} \right). \end{aligned}$$

4. Now, updating the data structure and the influence graph at step k takes time proportional to the number of nodes created or killed by O_k. Let $v(k, \mathcal{S})$ be the expected number of regions created at step k. From lemma 5.2.2, we derive

$$\begin{aligned} v(k, \mathcal{S}) &= \sum_{i=1}^{b} \sum_{j=0}^{n-i} |\mathcal{F}_j^i(\mathcal{S})| \frac{i}{k} p_j^i(k) \\ &= \frac{f_0(k, \mathcal{S})}{k}. \end{aligned}$$

Let now $v'(k, \mathcal{S})$ be the expected number of regions killed at step k. We denote by $\mathcal{S}_{k-1}$ the current subset immediately prior to step k. A region F in $\mathcal{F}_j^i(\mathcal{S})$ is a region killed at step k if it is a region of $\mathcal{F}_0(\mathcal{S}_{k-1})$ that conflicts with O_k, which happens with probability

$$p_j^i(k-1) \frac{j}{n-k+1}.$$

Again, using lemma 4.2.5 and corollary 4.2.7, we get

$$\begin{aligned} v'(k,S) &= \sum_{i=1}^{b}\sum_{j=0}^{n-i} |\mathcal{F}_j^i(S)|\, p_j^i(k-1)\, \frac{j}{n-k+1} \\ &= \frac{m_1(k-1,S)}{k-1} \\ &= O\left(\frac{f_0(\lfloor (k-1)/2 \rfloor, S)}{k-1}\right). \end{aligned}$$

This completes the proof of theorem 5.3.4. □

Some remarks on the update condition

The update condition 5.3.3 is not mandatory and it is often possible to analyze an on-line algorithm that does not satisfy all of its clauses.

1. For instance, if the first clause is not satisfied, the cost of testing the conflicts may be added to the analysis. If this cost can be bounded, this bound appears as a multiplicative factor in the cost of the locating phases.

2. The analyses of on-line algorithms developed above and in the remainder of this section are still valid for less restrictive statements of the third clause. We may assume only that the cost of the update phase is proportional to the number of regions created or killed. We have preferred, however, to assume that the parents of nodes created by some object are killed by the same object. This assumption is satisfied by most of the algorithms given in this book, and it greatly simplifies the analysis of dynamic on-line algorithms given in the next chapter.

3. Lastly, the second clause can also be relaxed. Indeed, in order to bound the space needed to store the influence graph, it suffices to bound the total number of arcs in the entire graph and not necessarily the out-degree of each node. We may then generalize the analysis of the locating phase by using the notion of a biregion (see exercise 5.7). In particular, such an analysis applies to the case when the number of parents of a node is bounded, but not the number of children. We illustrate this situation in the case of the on-line computation of convex hulls (see exercise 8.5).

5.3.2 An example: vertical decomposition of line segments

Again, we discuss the problem of building the vertical decomposition of a set of line segments in the plane, and this time we show how to compute it on-line, using an influence graph. Each time a segment is inserted, the algorithm updates the decomposition of the current set of segments, called the *current*

decomposition for short. The notions of objects, regions, and conflicts are defined as in subsection 5.2.2.

The trapezoids in the current decomposition are the regions defined and without conflict over the current set of segments and are linked to the corresponding nodes in the influence graph. An internal node of this graph is associated with a trapezoid which was in the current decomposition at some previous step of the algorithm. In addition to the set of pointers that take care of the parent–child relationships between the nodes, each node contains the following information:

- A description of the corresponding trapezoid and a list of the (at most four) segments that determine it.
- At most four pointers for the adjacency relationships through the vertical walls. As long as the node is a leaf of the influence graph, the corresponding trapezoid F belongs to the current decomposition and is adjacent to at most four leaves in the graph, each of which shares a vertical wall with F. When the node corresponding to F becomes an internal node, these pointers are not modified any more.

Therefore, a description of the simplified decomposition can be extracted from the information stored at the leaves of the graph.

At each step, the new segment S is located in the influence graph, yielding the list $\mathcal{L}(S)$ of trapezoids that it intersects. Each of these trapezoids is subdivided into at most four subregions by S, by the walls stemming from the endpoints of S, and by the walls stemming from the intersection points between S and some previously inserted segment. In the influence graph, a temporary node is created for these subregions. For each trapezoid F in the current decomposition that is intersected by S, we create links to the temporary nodes for the subregions F_i inside F, and they become the children of F in the influence graph (see figure 5.3). The list $\mathcal{L}(S)$ is not sorted, yet the vertical adjacency pointers allow a traversal of the decomposition in the left-to-right order along S. This allows us to set the vertical adjacencies of the subregions, and to identify which walls have to be removed and which subregions have to be joined to obtain the simplified decomposition $\mathcal{D}ec_s(\mathcal{R} \cup \{S\})$ after the insertion of S. We replace all the temporary nodes that correspond to subregions to be joined by a single node that inherits all the parents of the subregions. In this fashion, a leaf of the graph is created for each trapezoid F created by S, and is linked to all the trapezoids F' in $\mathcal{L}(S)$ which intersect the interior of F (see figures 5.6, 5.7, and 5.8). Properties 1 and 2 of the influence graph are thus maintained from step to step. Vertical adjacency relationships between trapezoids created by S can be derived from those of the subregions. This completes the description of the update phase in an incremental step.

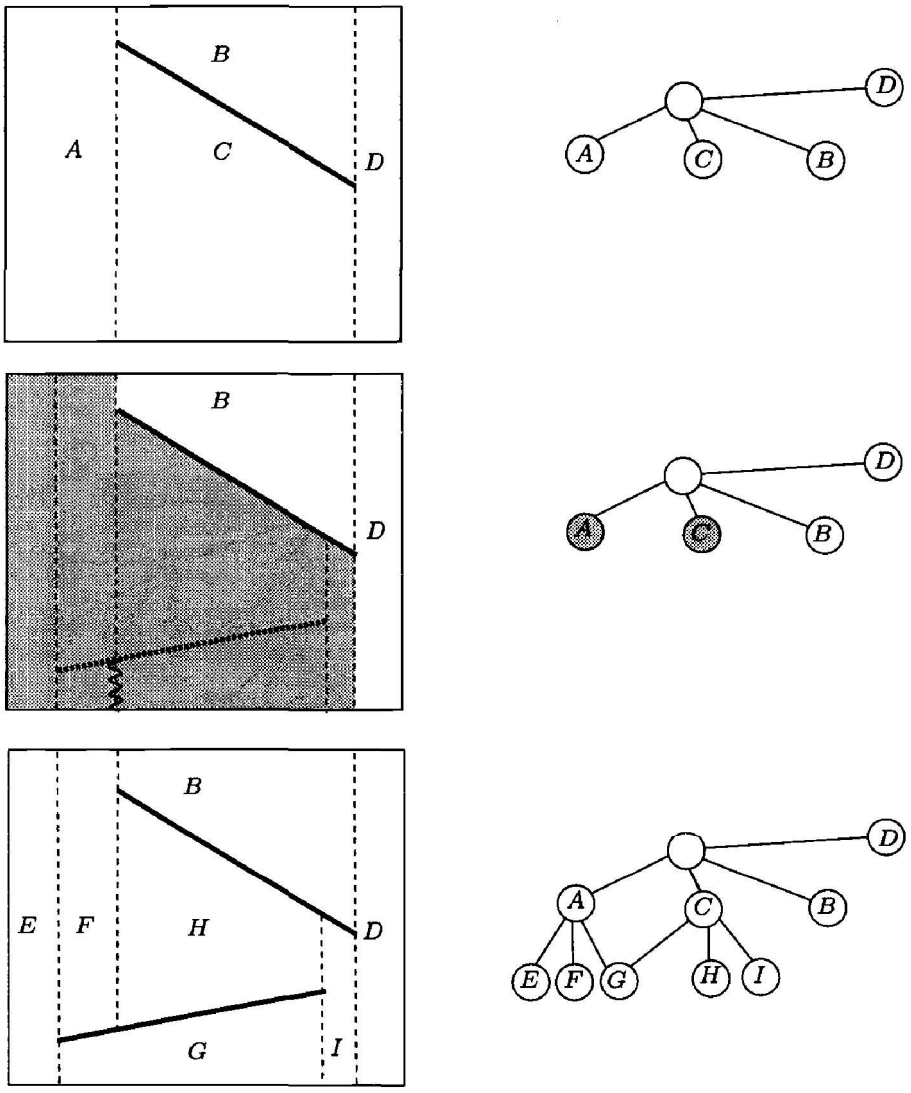

Figure 5.6. Vertical decomposition and influence graph (1).

Each internal node of the influence graph has at most four children, and the running time needed to carry out all the operations described in the previous paragraph is clearly proportional to the number of trapezoids in $\mathcal{L}(S)$. The update condition is therefore satisfied.

From lemma 5.2.4, we know that the expected number of trapezoids in the vertical decomposition of a random r-sample is $O(n + ar^2/n^2)$, if n is the number of segments in S and a is the number of intersecting pairs of segments. Theorem 5.3.4 therefore shows that the on-line algorithm just described has an expected complexity of $O(n \log n + a)$ and uses expected storage $O(n + a)$. The average complexity of the n-th insertion is $O(\log n + a/n)$.

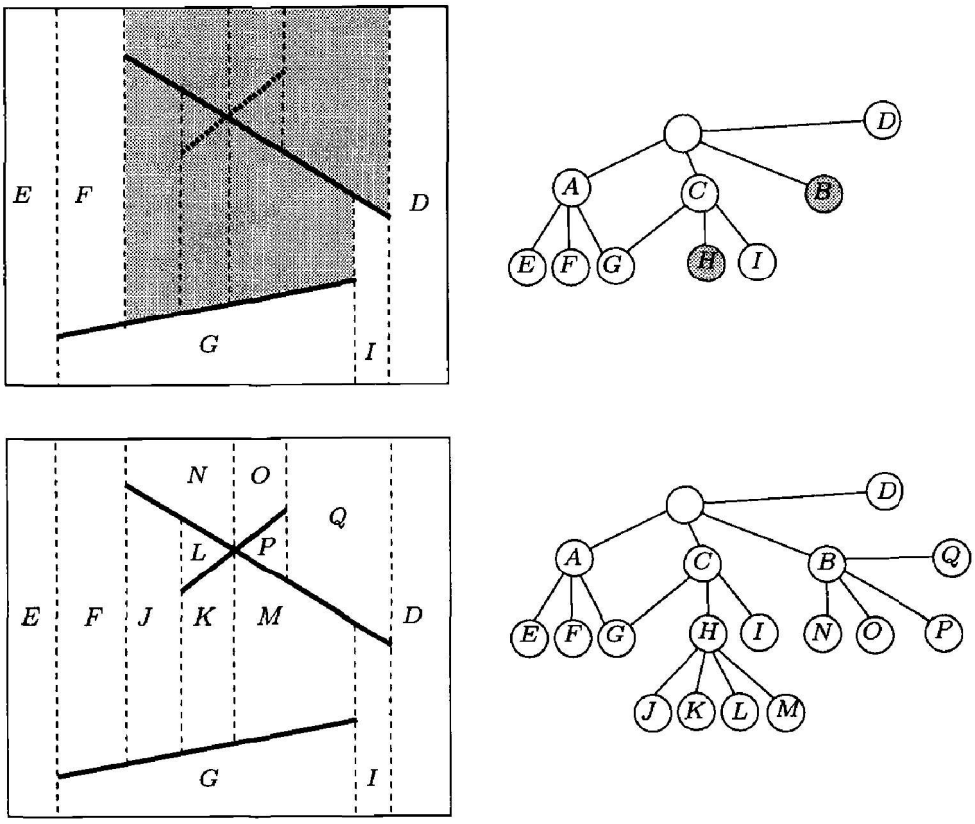

Figure 5.7. Vertical decomposition and influence graph (2).

5.4 Accelerated incremental algorithms

One of the problems encountered in solving a problem using the incremental method, that is, in identifying the set $\mathcal{F}_0(\mathcal{S})$ of those regions defined and without conflict over a given set $\mathcal{S}$ of objects, is the detection of conflicts between a new object and a region defined and without conflict over the current subset. Algorithms that use a conflict graph are static, as opposed to on-line algorithms which use an influence graph. In this section, which may be skipped in a first reading, we show how to combine both data structures to transform an on-line algorithm into a static one that has a lower asymptotic average complexity.

5.4.1 The general method

Theorem 5.3.4 on the influence graph shows that, if the expected number of regions defined and without conflict over $\mathcal{S}$ is $O(r)$, then the complexity of any algorithm that uses an influence graph is dominated by the cost of the locating phases in the incremental steps. (This cost is $O(n \log n)$, whereas the cost of the updates is only $O(n)$.) The idea is to lower the complexity of the locating phase

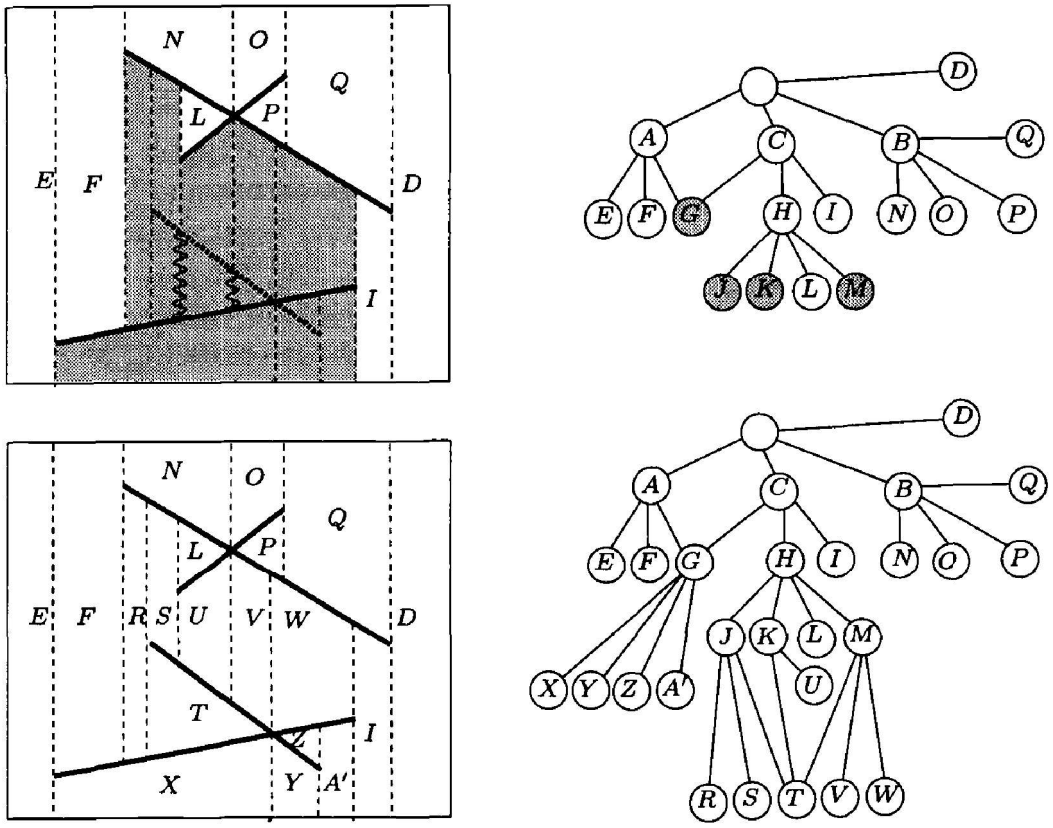

Figure 5.8. Vertical decomposition and influence graph (3).

by using a conflict graph, in addition to the influence graph. We cannot afford to maintain this conflict graph at every step, however, so we update it only at certain steps of the algorithm.

Let $\mathcal{S}_k$ be the current subset immediately after step k. The *conflict graph at step k* is the conflict graph that stores all the conflicts between the regions in $\mathcal{F}_0(\mathcal{S}_k)$ and the objects in $\mathcal{S} \setminus \mathcal{S}_k$. The following theorem shows how to use a knowledge of this conflict graph at step k to speed up the subsequent locating phases.

Theorem 5.4.1 *If an on-line algorithm satisfies the update condition, a knowledge of the conflict graph at step k can be used to perform the locating phase in a subsequent step l, with an average complexity of*

$$O\left(\sum_{r=k+1}^{l-1} \frac{f_0(\lfloor r/2 \rfloor, \mathcal{S})}{r^2}\right).$$

In particular, if $f_0(r, \mathcal{S})$ is $O(r)$, the cost of a locating phase at step l is $O\left(\log(\frac{l}{k})\right)$ on the average.

Proof. The conflict graph at step k can be augmented, for each object O in $\mathcal{S} \setminus \mathcal{S}_k$, by a list of pointers to the nodes of the influence graph which correspond to a region of $\mathcal{F}_0(\mathcal{S}_k)$ that conflicts with O. In order to locate the object O_l at step l, the algorithm may start to traverse the influence graph not from the root, but from the nodes of the influence graph which correspond to a region of $\mathcal{F}_0(\mathcal{S}_k)$ that conflicts with O_l. If the update condition is satisfied, the number of children of each node is bounded by a constant, and the number of nodes visited is proportional to the number of regions F created between steps $k+1$ and $l-1$ that conflict with O_l. A region F in $\mathcal{F}_j^i(\mathcal{S})$ is created at step r with probability $\frac{i}{r} p_j^i(r)$. Given this, the conditional probability that F conflicts with a given object O_l is $\frac{j}{n-r}$. The expected number $w(l, \mathcal{S})$ of nodes visited while locating O_l is thus

$$w(l,\mathcal{S}) = O\left(\sum_{r=k+1}^{l-1} \sum_{i=1}^{b} \sum_{j=0}^{n-b} |\mathcal{F}_j^i(\mathcal{S})|\, p_j^i(r)\, \frac{i}{r}\, \frac{j}{n-r}\right).$$

In this expression we recognize the first order moment (lemma 4.2.5). Using corollary 4.2.7 to the moment theorem, we obtain

$$w(l,\mathcal{S}) = O\left(\sum_{r=k+1}^{l-1} \frac{b}{r(n-r)}\, m_1(\mathcal{R},\mathcal{S})\right) = O\left(\sum_{r=k+1}^{l-1} \frac{f_0(\lfloor r/2 \rfloor, \mathcal{S})}{r^2}\right). \qquad \square$$

The accelerated algorithm proceeds as follows. At each step, the algorithm updates the influence graph. At certain steps, numbered $n_1, n_2, \ldots, n_k, \ldots$, the algorithm builds the conflict graph between the regions in $\mathcal{F}_0(\mathcal{S}_{n_k})$ and the objects in $\mathcal{S} \setminus \mathcal{S}_{n_k}$. This conflict graph is used to speed up the locating phases in the steps whose numbers range from $(n_k + 1)$ to n_{k+1}.

Of course, this design is useful only if the computation of the conflict graph at steps $n_1, n_2, \ldots, n_k, \ldots$ is not too unwieldy. The following theorem presents a general situation when an on-line algorithm that runs in expected time $O(n \log n)$ can be speeded up into a static algorithm that runs in expected time $O(n \log^* n)$.

Theorem 5.4.2 *Let $\mathcal{S}$ be a set of n objects, and $\mathcal{R}$ be a random r-sample of $\mathcal{S}$. Suppose that the expected number $f_0(r, \mathcal{S})$ of regions defined and without conflict over $\mathcal{R}$ is $O(r)$. If the conflict graph at any step k can be built in expected time $O(n)$, then the randomized accelerated algorithm runs in expected time $O(n \log^* n)$.*

Proof. The conflict graph is computed at steps $n_k = \left\lfloor n / \log^{(k)} n \right\rfloor$, for $k = 1, \ldots, \log^* n$. The conflict graph is therefore computed $\log^* n$ times overall, accounting

for an expected complexity of $O(n \log^* n)$. The locating phases, between step n_k and step n_{k+1}, have a total average complexity of

$$\begin{aligned}\sum_{l=n_k+1}^{n_{k+1}} O\left(\log\left(\frac{l}{n_k}\right)\right) &= \sum_{l=n_k+1}^{n_{k+1}} O\left(\log\left(\frac{l}{n}\log^{(k)} n\right)\right) \\ &= (n_{k+1} - n_k) O\left(\log\log^{(k)} n\right) \\ &= O(n).\end{aligned}$$

The total contribution of the locating phases to the running time is therefore, on the average, $O(n \log^* n)$. This fact combined with theorem 5.3.4 proves that the average complexity of the accelerated algorithm is $O(n \log^* n)$. □

5.4.2 An example: vertical decomposition of a polygon

The vertical decomposition of a polygon $\mathcal{P}$, which we denote by $\mathcal{D}ec(\mathcal{P})$, is the decomposition $\mathcal{D}ec(\mathcal{S}_\mathcal{P})$ of the set of line segments $\mathcal{S}_\mathcal{P}$ that are the edges on the boundary of the polygon $\mathcal{P}$. The decomposition of a simple polygon is a very interesting structure since we can derive from it a triangulation of the polygon, as we explain in chapter 12. The previous method helps to compute the decomposition of a polygon with n sides in expected time $O(n \log^* n)$, which therefore leads to a randomized algorithm of better average complexity than most of its deterministic counterparts.

The algorithm processes the segments in $\mathcal{S}_\mathcal{P}$ in a random order, builds an influence graph as described in subsection 5.3.2, and maintains the simplified decomposition of the current set of edges. In accordance with the preceding idea, it computes a conflict graph at steps $n_k = \left\lfloor n/\log^{(k)} n \right\rfloor$, $k = 1, \ldots, \log^* n$, which is then used to locate the subsequent edges between step n_k and step n_{k+1}.

The segments in $\mathcal{S}_\mathcal{P}$ may intersect only at their endpoints. Lemma 5.2.4 shows that the number $f_0(r, \mathcal{S}_\mathcal{P})$ of trapezoids in the decomposition of a random r-sample of $\mathcal{S}_\mathcal{P}$ is $O(r)$.

The conflict graph at step n_k can be computed using the following method in expected time $O(n)$. Let $\mathcal{D}ec_s(\mathcal{S}(n_k))$ be the current decomposition immediately after step n_k. A simplified decomposition will suffice for our purposes. We first begin by locating a given vertex of the polygon in the decomposition $\mathcal{D}ec_s(\mathcal{S}(n_k))$, using brute force and $O(n)$ operations. We then follow the boundary of the polygon, reporting all the conflicts between the edges and the trapezoids of the decomposition $\mathcal{D}ec_s(\mathcal{S}(n_k))$. Immediately after step n_k, an edge has either been inserted already, or it conflicts with some trapezoids in the decomposition $\mathcal{D}ec_s(\mathcal{S}(n_k))$. In the former case, it has been split into possibly many edges of

this decomposition, and the total complexity of following these edges is dominated by the size $O(n_k) = O(n)$ of this decomposition. In the latter case, the cost of following these edges is proportional to the number of conflicts between these edges and the trapezoid of the decompositions $\mathcal{D}ec_s(\mathcal{S}(n_k))$.[2] From theorem 4.2.6 and its corollary 4.2.7, the expected number of conflicts reported at step n_k is exactly the first order moment of the current subset of edges at step n_k. From corollary 4.2.7, this number is $O(f_0(\lfloor n_k/2 \rfloor, \mathcal{S}))$, which is $O(n)$ for non-intersecting segments, as is the case for the edges of a polygon.

The hypotheses of theorem 5.4.2 are thus satisfied, which yields:

Theorem 5.4.3 *A randomized incremental algorithm can build the vertical decomposition of a simple polygon with n edges in expected time $O(n \log^* n)$.*

Remark. The algorithm relies on two facts: the edges are connected, and do not intersect except possibly at common endpoints. The same algorithm therefore works as well in the more general cases of a polygonal line, or a connected set of segments whose pairwise interiors are disjoint.

5.5 Exercises

Exercise 5.1 (Probabilities) Prove that

$$\sum_{r=i}^{n-j} \frac{i}{r} \frac{\binom{n-i-j}{r-i}}{\binom{n}{r}} = \frac{i!j!}{(i+j)!}.$$

Then show that the probabilities p'^i_j and $p'^i_j(r)$ defined in section 5.2 satisfy the following relation:

$$p'^i_j = \sum_{r=1}^{n} p'^i_j(r).$$

Exercise 5.2 (Non-amortized analysis of the conflict graph) Consider a randomized incremental algorithm that processes a set $\mathcal{S}$ of n objects by using a conflict graph. Show that if the update condition is satisfied, then the complexity of step k is on the average

$$O\left(\frac{f_0(k,\mathcal{S})}{k} + \frac{n}{k^2} f_0\left(\left\lfloor \frac{k}{2} \right\rfloor, \mathcal{S}\right) + \frac{n}{(k-1)^2} f_0\left(\left\lfloor \frac{k-1}{2} \right\rfloor, \mathcal{S}\right)\right).$$

[2] We may recall that each trapezoid of the decomposition is adjacent to at most four trapezoids through vertical walls and these adjacencies are encoded through additional pointers in the influence graph. Since an edge may not intersect a floor or ceiling, we can trace its conflicts in the current decomposition using constant time per conflict.

Hint: The expected number of regions killed or created during step k of a randomized incremental algorithm is estimated in the proof of theorem 5.3.4. It remains to estimate the expected number of conflict arcs added or removed during step k.

Exercise 5.3 (Complete description of the vertical decomposition) Let $\mathcal{S}$ be a set of line segments in the plane, with a intersecting pairs, and let $\mathcal{R}$ be a random r-sample of $\mathcal{S}$. We consider the complete description of the decomposition $\mathcal{D}ec(\mathcal{R})$ of $\mathcal{R}$, and particularly the expected number of edges on the boundaries of the trapezoids of $\mathcal{D}ec(\mathcal{R})$ which are cut by a random segment of $\mathcal{S} \setminus \mathcal{R}$. Show that this expectation is $O(1 + ar/n^2)$.

From this, show that the randomized incremental algorithm described in subsection 5.2.3 can be slightly modified to compute, within the same complexity bounds, a complete description of the decomposition of the line segments, which in particular includes all the adjacency relationships between the trapezoids.

Hint: We may redefine the notions of regions and conflicts as follows: A region defined on $\mathcal{R}$ is a *paddle* with two components, a trapezoid F in the decomposition $\mathcal{D}ec(\mathcal{R})$, and a wall butting on the floor or ceiling of F. A paddle is determined by at most six segments. It conflicts with a segment if the interior of the trapezoid intersects the segment. The problem is now to find an upper bound on the number of paddles defined and without conflict with a segment of $\mathcal{S} \setminus \mathcal{R}$.

Exercise 5.4 (Storage) Consider the incremental algorithm that uses a conflict graph as in subsection 5.2.2 in order to compute the decomposition of a set $\mathcal{S}$ of n segments. Show that if a is the number of intersecting pairs, the storage needed by the algorithm at step k is, on the average,

$$m_1(k, \mathcal{S}) = O\left(n + a\frac{k}{n}\right).$$

Using the result of the previous exercise, show that we may reduce the storage to $O(n)$ by storing only one conflict for each non-inserted segment, say with the trapezoid that contains its left endpoint, without affecting the running time of the algorithm.

Exercise 5.5 (Decomposing a set of curves) Show how to generalize the notion of a decomposition for a set of curves supported by algebraic curves of bounded degree. Two such curves intersect at only a constant number of points, which we assume may be computed in constant time. Show that both algorithms given in subsections 5.2.2 and 5.3.2 may be extended to build the decomposition of a set of such curves.

Hint: Do not forget to trace walls from each point where the curves have a vertical tangent.

Exercise 5.6 (Backward analysis) Backward analysis (see also exercises 4.1 and 4.2) gives an alternative proof of the results of this chapter without using the explicit expressions for $p_j^i(r)$ and $p_j'^i(r)$.

For instance, we show how backward analysis can be used to estimate the number $v(k, \mathcal{S})$ of regions created at step k by an incremental algorithm. Note that if $\mathcal{S}_k$ is the current subset immediately after inserting object O_k at step k, the regions created by O_k during this step are the regions of $\mathcal{F}_0(\mathcal{S}_k)$ determined by a subset of $\mathcal{S}_k$ that contains O_k. Since O_k, which has chronological rank k, may be any of the objects in $\mathcal{S}_k$ with uniform probability $\frac{1}{k}$, a region of $\mathcal{F}_0(\mathcal{S}_k)$ is created at step k with probability at most b/k. Therefore, $v(k, \mathcal{S})$ is at most the expectation of $\frac{b}{k}|\mathcal{F}_0(\mathcal{S}_k)|$ over all possible $\mathcal{S}_k$.

Similarly, a region that is killed during step k is a region of $\mathcal{F}_1(\mathcal{S}_k)$ that conflicts with O_k. Any region of $\mathcal{F}_1(\mathcal{S}_k)$ conflicts with O_k with probability $1/k$. The expected number $v'(k, \mathcal{S})$ of regions killed during step k is therefore at most the expectation of $\frac{1}{k}|\mathcal{F}_1(\mathcal{S}_k)|$ over all possible $\mathcal{S}_k$.

It is possible to compute in this fashion the expected numbers $v(k, \mathcal{S})$ and $v'(k, \mathcal{S})$ of regions created or killed during step k. Show how to use backward analysis to prove the other results in this chapter, for instance, to bound the number of conflict arcs that are added to or removed from the conflict graph, or to bound the number of conflicts detected during a locating phase by an algorithm that uses an influence graph.

Exercise 5.7 (Biregions) The notion of *biregion* introduced in this exercise can be used to analyze the average complexity of some algorithms that use an influence graph, but do not satisfy the update condition 5.3.3. A *biregion* is pair of regions which can have a parent–child relationship in the influence graph for at least one permutation of the data. A biregion is determined by a set of at most $2b$ objects, those that determine the parent region and those that determine the child region. Exactly one of the objects that determine the child region conflicts with the parent region. We can extend the notion of conflict to biregions in the following way: an object conflicts with a biregion if it conflicts with at least one of its two regions and does not belong to the set of objects that determine the biregion. A biregion can then be considered as a region in the framework described in chapter 4.

1. Let $\mathcal{S}$ be a set of n objects. Show that a biregion, determined by i objects of $\mathcal{S}$ and in conflict with j objects of $\mathcal{S}$, is defined and with k conflicts over a random r-sample of $\mathcal{S}$ with the probability $p^i_{j,k}(r)$ given by lemma 4.2.1.

2. From this, extend both the sampling theorem and the moment theorem to the case of biregions.

3. In essence, the difference between biregions and regions resides in the following fact. Let *FF* be a biregion determined by i objects and conflicting with j objects of $\mathcal{S}$. For *FF* to correspond to an arc in the influence graph built for $\mathcal{S}$, it is not enough that the i objects that determine *FF* be inserted before any of the j objects that conflict with *FF*; it must also be the case that the i objects that determine i be processed in a certain order. This order has to meet several criteria. These criteria depend on the algorithm. At the very least, one of the objects that determine the child region, more precisely the one that conflicts with the parent region, must be inserted after all the objects that determine the parent region.

Show that the probability that *FF* correspond to an arc of the influence graph is $\alpha {p'}^i_j$, where ${p'}^i_j$ is given in lemma 5.2.2, and α is a constant that satisfies $\frac{1}{(2b)!} \leq \alpha \leq \frac{1}{i}$ and that depends only on the particular criteria that the insertion order has to meet. Then

show that the probability that the biregion FF correspond to an arc of the influence graph that is created at step r is $\alpha \frac{i}{r} p_j^i(r)$, where $p_j^i(r)$ is defined in subsection 4.2.1.

4. Our goal is now to give a randomized analysis of an on-line algorithm that uses an influence graph in which a node can have arbitrarily many children. We thus forget about the second clause in the update condition 5.3.3, and relax the third one by assuming that the parents of a region created by O are either killed by O, or still have no conflict after the insertion of O. In this way, regions defined and without conflict with the current subset may not be leaves of the influence graph, but could have many children before they are killed. The complexity of the update phase is still assumed to take time proportional to the number of arcs added to or removed from the influence graph. For instance, the algorithm that computes convex hulls described in exercise 8.5 meets these conditions.

Let $ff_0(r, \mathcal{S})$ stand for the expected number of biregions defined and without conflict over a random r-sample of $\mathcal{S}$. Show that the number of arcs in the influence graph built for $\mathcal{S}$ is, on the average,

$$\Theta\left(\sum_{r=1}^{n} \frac{ff_0(r, \mathcal{S})}{r}\right).$$

Show that the cost of the locating phases is

$$O\left(n \sum_{r=1}^{n} \frac{ff_0(r, \mathcal{S})}{r^2}\right).$$

5. Assume now that the influence graph built for a random r-sample of $\mathcal{S}$ has an expected number of arcs at most $g(r, \mathcal{S})$, where g is a known function. For instance, when each node of the influence graph has at most a bounded number of children, we may choose

$$g(r, \mathcal{S}) = O\left(\sum_{j=1}^{n} \frac{f_0(j, \mathcal{S})}{j}\right).$$

Show that the n-th incremental step of the on-line algorithm has an average complexity of

$$O\left(\frac{g(n, \mathcal{S})}{n} + \sum_{r=1}^{n} \frac{g(r, \mathcal{S})}{r^2}\right).$$

Exercise 5.8 (Decomposing a polygon) This exercise presents another randomized algorithm that builds the vertical decomposition of a simple polygon with n edges in expected time $O(n \log^* n)$.

The algorithm is incremental but inserts a number of edges of the polygon at a time. Let $\mathcal{P}$ be a polygon, and $\mathcal{S}$ the set of its n edges. Assume that the segments in $\mathcal{S}$ are ordered in a random order, and let $\mathcal{S}_i$ be the subset containing the first r_i segments of $\mathcal{S}$, with $r_i = \left\lfloor n / \left\lceil \log^{(i)} n \right\rceil \right\rfloor$. The subset $\mathcal{S}_i$ is thus a random r_i-sample of $\mathcal{S}$, and

$$\mathcal{S}_1 \subset \mathcal{S}_2 \subset \ldots \subset \mathcal{S}_{\log^* n} = \mathcal{S}.$$

The algorithm computes a simplified description of the decomposition $\mathcal{D}ec_s(\mathcal{P})$, using $\log^* n$ steps. Step i computes the decomposition $\mathcal{D}ec_s(\mathcal{S}_i)$ from $\mathcal{D}ec_s(\mathcal{S}_{i-1})$.

In the initial step, we build $\mathcal{D}ec_s(\mathcal{S}_1)$ using the plane sweep algorithm of subsection 3.2.2, in time $O(r_1 \log r_1)$. (Any algorithm that runs in time $O(r_1 \log r_1)$ would do.)

In a subsequent step i, $i > 1$:

1. We locate the segments of $\mathcal{S}$ in $\mathcal{D}ec_s(\mathcal{S}_{i-1})$. In other words, for each region F in $\mathcal{D}ec_s(\mathcal{S}_{i-1})$, we compute the set $\mathcal{S}(F)$ of segments in $\mathcal{S}$ which intersect F.
2. For each region F of $\mathcal{D}ec_s(\mathcal{S}_{i-1})$, we compute the decomposition of $\mathcal{S}(F) \cup \mathcal{S}_i$, and the portion of it that lies inside F. To do this, we simply compute the total decomposition $\mathcal{D}ec_s(\mathcal{S}(F) \cup \mathcal{S}_i)$, using the plane sweep algorithm of subsection 3.2.2. (Again, any algorithm that runs in time $O(m \log m)$ for m segments would do.)
3. We obtain $\mathcal{D}ec_s(\mathcal{S}_i)$ by putting together all the portions $\mathcal{D}ec_s(\mathcal{S}(F) \cup \mathcal{S}_i) \cap F$ inside the trapezoids F of $\mathcal{D}ec_s(\mathcal{S}_{i-1})$, and merging the regions that share a wall of $\mathcal{D}ec_s(\mathcal{S}_{i-1})$ which disappears in $\mathcal{D}ec_s(\mathcal{S}_i)$.

Show that all three phases 1, 2, and 3 can be performed using $O(n)$ operations. To analyze phase 2, note that $\mathcal{S}_{i-1}$ is a random r_{i-1}-sample of $\mathcal{S}_i$, then use the extension of the moment theorem given in exercise 4.3 for the function $g(x) = x \log x$.

Exercise 5.9 (Querying the influence graph) The influence graph built by an on-line algorithm can be used to answer conflict queries on a set of objects. For instance, the influence graph built for a vertical decomposition can answer location queries for a point inside this decomposition. Show that, if n segments are stored in the influence graph, answering a given location query takes time $O(\log n)$, on the average over all possible insertion orders of the n segments into the influence graph. More generally, show that the same time bound holds for any conflict query which, on any subset $\mathcal{R}$ of objects, answers with a single region of $\mathcal{F}_0(\mathcal{R})$.

5.6 Bibliographical notes

The first non-trivial (that is, sub-quadratic) algorithm that computes all the intersecting pairs in a set of segments in the plane is that of Bentley and Ottmann [23], which uses a plane sweep method. This algorithm, described in chapter 3, computes all a intersecting pairs in $O((n+a) \log n)$, which falls short of being optimal. About ten years later, Chazelle and Edelsbrunner proposed in [48, 49] a deterministic algorithm that runs in optimal time $O(n \log n + a)$ to compute all the a intersections. The description and implementation of their algorithm is rather complicated, however. At about the same time, Clarkson and Shor [71] and independently Mulmuley [171, 173] proposed randomized incremental algorithms for the same problem that have an optimal average complexity.

The algorithm by Clarkson and Shor that uses a conflict graph is described in section 5.2 in this chapter. In the same paper [71], they also set up the formalism of objects, regions, and conflicts, and introduce the conflict graph in these terms; they give other algorithms that use the conflict graph (computing the intersection of n half-spaces, the diameter of a point set in 3 dimensions), and show how to compute a complete description of the decomposition of line segments and how to lower the storage requirements of

their algorithm (see exercises 5.3 and 5.4). Mulmuley's algorithm is very similar to that of Clarkson and Shor, yet its analysis is based on probabilistic games and combinatorial series, and is much less immediate.

The influence graph was first introduced in a paper by Boissonnat and Teillaud [31, 32] where it was called the *Delaunay tree*, and was used there to compute on-line the Delaunay triangulation of a set of points. Guibas, Knuth, and Sharir [117] proposed a similar algorithm to solve the same problem. How to use the influence graph in an abstract setting is described by Boissonnat, Devillers, Schott, Teillaud, and Yvinec in [28] and applied to other problems, especially to compute convex hulls or to decompose a set of segments in the plane. The method was later used to solve numerous other problems. The influence graph is sometimes called the *history* of the incremental construction.

The accelerated algorithm that builds the vertical decomposition of a simple polygon is due to Seidel [204]. This method was subsequently extended to solve other problems by Devillers [80], for instance to compute the Voronoi diagram of a polygonal line or of a closed simple polygon (see section 19.2). The algorithm described in exercise 5.8 that computes the decomposition of a polygon in time $O(n \log^* n)$ is due to Clarkson, Cole and Tarjan [69].

The method called *backward analysis* used in exercise 5.6 was first used by Chew in [59] to analyze an algorithm that computes the Voronoi diagram of a convex polygon (see exercise 19.4). It was subsequently used in a systematic fashion by Seidel in [203] and Devillers in [80].

Mehlhorn, Sharir, and Welzl [167, 168] gave a finer analysis of randomized incremental analysis by bounding the probability that the algorithm exceeds its expected performances by more than a constant multiplicative factor.

Randomized incremental algorithms proved very efficient in solving many geometric problems. The basic methods (using the influence or the conflict graphs) or one of their many variants inspired much work by several researchers such as Mulmuley [172, 174], Mehlhorn, Meiser and Ó'Dúnlaing [164], Seidel [205], Clarkson and Shor [71], and Aurenhammer and Schwarzkopf [18].

There is a class of randomized algorithms which work not by the incremental method, but rather by the divide-and-conquer paradigm. The subdividing step is achieved using a sample of the objects to process. Randomization is used for choosing the sample, and the method can be proved efficient using the probabilistic theorems given in exercises 4.5 and 4.6. Randomized divide-and-conquer is mainly used for building hierarchical data structures that support repeated range queries. Typically, these queries can be expressed in terms of locating a point in the arrangement of a collection of hyperplanes, simplices, or other geometric objects. In a dual situation, the data set is a set of points and the queries ask for those points which lie in a given region (half-space, simplex, ...). Haussler and Welzl [123] spurred new interest in the field with their notion of an ϵ-net. Later, Matoušek introduced the related notion of ϵ-approximations [150]. Chazelle and Friedman [53] showed how to compute these objects in a deterministic fashion using the method of conditional probabilities. The resulting deterministic method is called a *derandomization* of the randomized divide-and-conquer method. This method was then widely used, for instance by Matoušek [150, 151, 152, 153, 154, 155], Matoušek and Schwarzkopf [156], or Agarwal and Matoušek [4]. In his thesis [35], Brönnimann

studies the derandomization of geometric algorithms and the related concept of the Vapnik–Chervonenkis dimension. Randomized divide-and-conquer is also used by Clarkson, Tarjan, and Van Wyk in [65] to build the vertical decomposition of a simple polygon.

Last but not least, the book by Mulmuley [177] is entirely devoted to randomized geometric algorithms, and serves as a very comprehensive reference on the topic.