16 Simplex Range Searching

Windowing Revisited

In Chapter 2 we saw that geographic information systems often store a map in a number of separate layers. Each layer represents a *theme* from the map, that is, a specific type of feature such as roads or cities. Distinguishing layers makes it easy for the user to concentrate her attention on a specific feature. Sometimes one is not interested in all the features of a given type, but only in the ones lying inside a certain region. Chapter 10 contains an example of this: from a road map of the whole of the U.S.A. we wanted to select the part lying inside a much smaller region. There the query region, or *window*, was rectangular, but it is easy to imagine situations where the region has a different shape. Suppose that

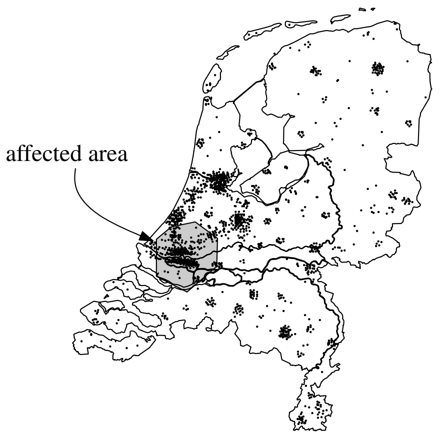


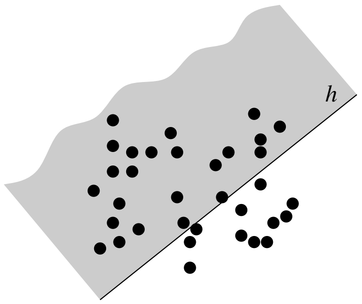
Figure 16.1
Population density of the Netherlands

we have a map layer whose theme is population density. The density is shown on the map by plotting a point for every 5,000 people, say. An example of such a map is given in Figure 16.1. If we want to estimate the impact of building, say, a new airport at a given location, it is useful to know how many people live in the affected area. In geometric terms we have a set of points in the plane and we want to count the points lying inside a query region (for instance, the region within which the noise of planes exceeds a certain level).

In Chapter 5, where we studied database queries, we developed a data structure to report the points inside an axis-parallel query rectangle. The area affected by the airport, however, is determined by the dominating direction of the wind, and unlikely to be rectangular, so the data structures from Chapter 5 are of little help here. We need to develop a data structure that can handle more general query ranges.

16.1 Partition Trees

Given a set of points in the plane, we want to count the points lying inside a query region. (Here and in the following we follow the convention of using the expression "to count the points" in the sense of "to report the number of points", not in the sense of enumerating the points.) Let's assume that the query region is a simple polygon; if it is not, we can always approximate it. To simplify the query answering algorithm we first triangulate the query region, that is, we decompose it into triangles. Chapter 3 describes how to do this. After we have triangulated the region, we query with each of the resulting triangles. The set of points lying inside the region is just the union of the sets of points inside the triangles. When counting the points we have to be a bit careful with points lying on the common boundary of two triangles, but that is not difficult to take care of.



We have arrived at the *triangular range searching problem*: given a set S of n points in the plane, count the points from S lying in a query triangle t. Let's first study a slightly simpler version of this problem, where the query triangle degenerates into a half-plane. What should a data structure for half-plane range queries look like? As a warm-up, let's look at the one-dimensional problem first: in the one-dimensional version of the problem we are given a set of n points on the real line, and we want to count the points in a query half-line (that is, the points lying on a specified side of a query point). Using a balanced binary search tree where every node also stores the number of points in its subtree, we can answer such queries in $O(\log n)$ time. How can we generalize this to a 2-dimensional setting? To answer this question we must first interpret a balanced search tree geometrically. Each node of the tree contains a key—the coordinate of a point—that is used to split the point set into the sets that are stored in the left and right subtrees. Similarly, we can consider this key value to split the real line into two pieces. In this way, every node of the tree corresponds to a region on the line—the root to the whole line, the two children of the root to two half-lines, and so on. For any query half-line and any node, the region of one child of the node is either completely contained in the half-line or completely disjoint from it. All points in that region are in the half-line, or none of them is. Hence, we only have to search recursively in the other subtree of the node. This is shown in Figure 16.2. The points are drawn below the tree as black dots; the two regions of the real line corresponding to the two subtrees are also indicated. The query half-line is the grey region. The region corresponding to the subtree rooted at the black node is completely inside the query half-line. Hence, we only have to recurse on the right subtree.

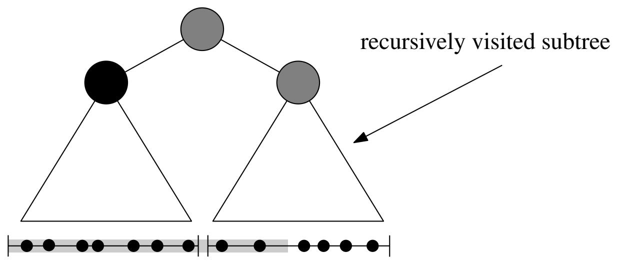


Figure 16.2
Answering a half-line query with a binary tree

To generalize this to two dimensions we could try to partition the plane into two regions, such that for any query half-plane there is one region that is either completely contained in the half-plane or completely disjoint from it. Unfortunately, such a partitioning does not exist, so we need a further generalization: instead of partitioning into two regions, we must partition into more regions. The partitioning should be such that for any query half-plane we have to search recursively in only few of the regions.

We now give a formal definition of the type of partitioning we need. A *simplicial partition* for a set S of n points in the plane is a collection $\Psi(S) := \{(S_1,t_1),\ldots,(S_r,t_r)\}$, where the S_i's are disjoint subsets of S whose union is S, and t_i is a triangle containing S_i. The subsets S_i are called *classes*. We do not require the triangles to be disjoint, so a point of S may lie in more than one triangle. Still, such a point is a member of only one class. We call r, the number of triangles, the *size* of $\Psi(S)$. Figure 16.3 gives an example of a simplicial partition of size five; different shades of grey are used to indicate the different classes. We say that a line ℓ crosses a triangle t_i if ℓ intersects the interior

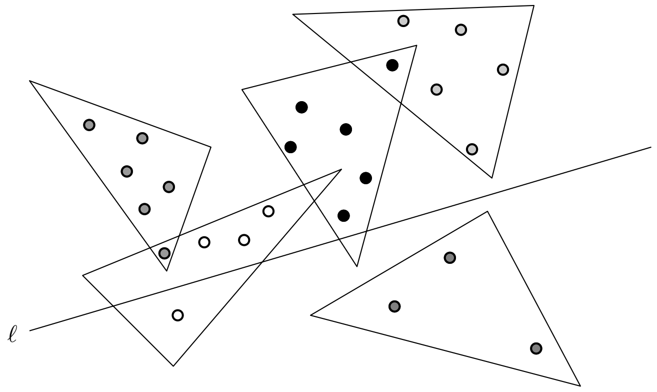


Figure 16.3
A fine simplicial partition

of t_i. When the point set S is not in general position we sometimes need to use (relatively open) segments as well as triangles in the simplicial partition. A line is defined to cross a segment when it intersects its relative interior but does not contain it. The *crossing number* of a line ℓ with respect to $\Psi(S)$ is the number of triangles of $\Psi(S)$ crossed by ℓ. Thus the crossing number of the line ℓ in Figure 16.3 is two. The crossing number of $\Psi(S)$ is the maximum crossing number over all possible lines ℓ. In Figure 16.3 you can find lines that intersect four triangles, but no line intersects all five. Finally, we say that a simplicial partition is *fine* if $|S_i| \leqslant 2n/r$ for every $1 \leqslant i \leqslant r$. In other words, in fine simplicial partitions none of the classes contains more than twice the average number of points of the classes.

Now that we have formalized the notion of partition, let's see how we can use such a partition to answer half-plane range queries. Let h be the query half-plane. If a triangle t_i of the partition is not crossed by the bounding line of h, then its class S_i either lies completely in h, or it is completely disjoint from h. This means we only have to recurse on the classes S_i for which t_i is crossed by the bounding line of h. For example, if in Figure 16.3 we queried with ℓ^+, the half-plane lying above ℓ, we would have to recurse on two of the five classes. The efficiency of the query answering process therefore depends on the crossing number of the simplicial partition: the lower the crossing number, the better the query time. The following theorem states that it is always possible to find a simplicial partition with crossing number $O(\sqrt{r})$; later we shall see what this implies for the query time.

Theorem 16.1 *For any set S of n points in the plane, and any parameter r with $1 \leqslant r \leqslant n$, a fine simplicial partition of size r and crossing number $O(\sqrt{r})$ exists. Moreover, for any $\varepsilon > 0$ such a simplicial partition can be constructed in time $O(n^{1+\varepsilon})$.*

It seems a bit strange to claim a construction time of $O(n^{1.1})$ or $O(n^{1.01})$ or with the exponent even closer to 1. Still, no matter how small ε is, as long as it is a positive constant, the bound in the theorem can be attained. But a better upper bound like $O(n)$ or $O(n \log n)$ isn't claimed in the theorem.

Section 16.4 gives pointers to the literature where a proof of this theorem can be found. We shall take the theorem for granted, and concentrate on how to use it in the design of an efficient data structure for half-plane range queries. The data structure we'll obtain is called a *partition tree*. Probably you can already

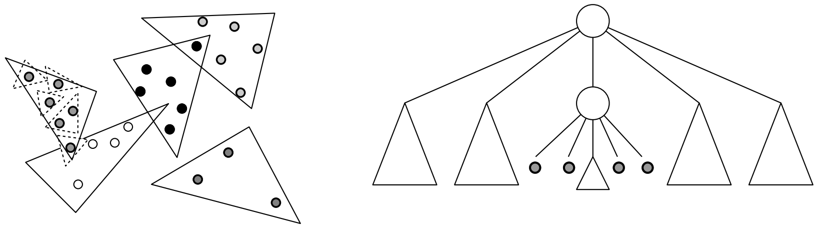


Figure 16.4
A simplicial partition and the corresponding tree

guess what such a partition tree looks like: it's a tree whose root has r children, each being the root of a recursively defined partition tree for one of the classes in a simplicial partition. There is no specific order on the children; it happens to be irrelevant. Figure 16.4 shows a simplicial partition and the corresponding tree. The dotted triangles form the partition computed recursively for the class corresponding to the middle child of the root; the five "subclasses" are stored in five subtrees below the middle child. Depending on the application, we also store some extra information about the classes. The basic structure of a partition tree is thus as follows:

- If S contains only one point, p, the partition tree consists of a single leaf where p is stored explicitly. The set S is the *canonical subset* of the leaf.
- Otherwise, the structure is a tree $\mathcal{T}$ of branching degree r, where r is a sufficiently large constant. (Below we shall see how r should be chosen.) The children of the root of the tree are in one-to-one correspondence with the triangles of a fine simplicial partition of size r for the set S. The triangle of the partition corresponding to child ν is denoted by $t(\nu)$. The corresponding class in S is called the *canonical subset* of ν; it is denoted $S(\nu)$. The child ν is the root of a recursively defined partition tree $\mathcal{T}_\nu$ on the set $S(\nu)$.
- With each child ν we store the triangle $t(\nu)$. We also store information about the subset $S(\nu)$; for half-plane range counting this information is the cardinality of $S(\nu)$, but for other applications we may want to store other information.

We can now describe the query algorithm for counting the number of points from S in a query half-plane h. The algorithm returns a set Υ of nodes from the partition tree $\mathcal{T}$, called the *selected nodes*, such that the subset of points from S lying in h is the disjoint union of the canonical subsets of the nodes in Υ. In other words, Υ is a set of nodes whose canonical subsets are disjoint, and such that

$$S \cap h \;=\; \bigcup_{\nu \in \Upsilon} S(\nu).$$

The selected nodes are exactly the nodes ν with the property: $t(\nu) \subset h$ (or, in case ν is a leaf, the point stored at ν lies in h) and there is no ancestor μ of ν such that $t(\mu) \subset h$. The number of points in h can be computed by summing the cardinalities of the selected canonical subsets.

Algorithm SELECTINHALFPLANE($h, \mathcal{T}$)
Input. A query half-plane h and a partition tree or subtree of it.
Output. A set of canonical nodes for all points in the tree that lie in h.

```
1.  $\Upsilon \leftarrow \emptyset$
2.  **if** $\mathcal{T}$ consists of a single leaf $\mu$
3.     **then if** the point stored at $\mu$ lies in $h$ **then** $\Upsilon \leftarrow \{\mu\}$
4.     **else for** each child $\nu$ of the root of $\mathcal{T}$
5.              **do if** $t(\nu) \subset h$
6.                    **then** $\Upsilon \leftarrow \Upsilon \cup \{\nu\}$
7.                    **else if** $t(\nu) \cap h \neq \emptyset$
8.                             **then** $\Upsilon \leftarrow \Upsilon \cup$ SELECTINHALFPLANE($h, \mathcal{T}_\nu$)
9.  **return** $\Upsilon$
```

Figure 16.5 illustrates the operation of the query algorithm. The selected children of the root are shown in black. The children that are visited recursively (as well as the root itself, since it has also been visited) are grey. As said before, a

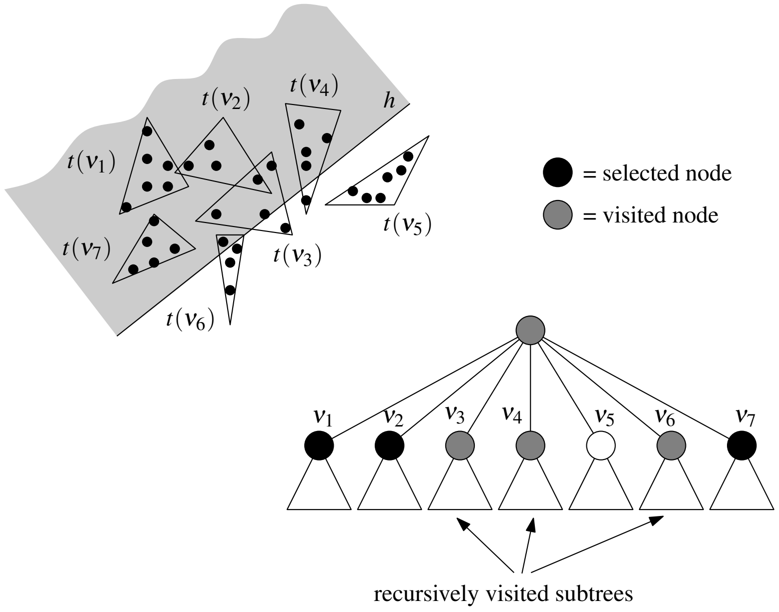


Figure 16.5
Answering a half-plane range query using a partition tree

half-plane range counting query can be answered by calling SELECTINHALFPLANE and summing the cardinalities of the selected nodes, which are stored at the nodes. In practice, one would probably not keep track of the set Υ, but one would maintain a counter; when a node is selected, the cardinality of its canonical subset is added the counter.

We have described the partition tree, a data structure for half-plane range counting, and its query algorithm. Now it's time to analyze our structure. We start with the amount of storage.

Lemma 16.2 *Let S be a set of n points in the plane. A partition tree on S uses $O(n)$ storage.*

Proof. Let $M(n)$ be the maximum number of nodes that a partition tree on a set of n points can have, and let n_ν denote the cardinality of the canonical subset $S(\nu)$. Then $M(n)$ satisfies the following recurrence:

$$M(n) \leqslant \begin{cases} 1 & \text{if } n = 1, \\ 1 + \sum_\nu M(n_\nu) & \text{if } n > 1, \end{cases}$$

where we sum over all children ν of the root of the tree. Because the classes in a simplicial partition are disjoint, we have $\sum_\nu n_\nu = n$. Furthermore, $n_\nu \leqslant 2n/r$ for all ν. Hence, for any constant $r > 2$ the recurrence solves to $M(n) = O(n)$.

The storage needed for a single node of the tree is $O(r)$. Since r is a constant, the lemma follows. $\square$

Linear storage is the best one could hope for, but what about the query time? Here it becomes important what the exact value of r is. When we perform a query we have to recurse in at most $c\sqrt{r}$ subtrees, for some constant c that does not depend on r or n. It turns out that this constant has an influence on the exponent of n in the query time. To decrease this influence, we need to make r large enough, and then, as we will see, we can get a query time close to $O(\sqrt{n})$.

Lemma 16.3 *Let S be a set of n points in the plane. For any $\varepsilon > 0$, there is a partition tree for S such that for a query half-plane h we can select $O(n^{1/2+\varepsilon})$ nodes from the tree with the property that the subset of points from S in h is the disjoint union of the canonical subsets of the selected nodes. The selection of these nodes takes $O(n^{1/2+\varepsilon})$ time. As a consequence, half-plane range counting queries can be answered in $O(n^{1/2+\varepsilon})$ time.*

Proof. Let $\varepsilon > 0$ be given. According to Theorem 16.1 there is a constant c such that for any parameter r we can construct a simplicial partition of size r with crossing number at most $c\sqrt{r}$. We base the partition tree on simplicial partitions of size $r := \lceil 2(c\sqrt{2})^{1/\varepsilon} \rceil$. Let $Q(n)$ denote the maximum query time for any query in a tree for a set of n points. Let h be a query half-plane, and let n_ν denote the cardinality of the canonical subset $S(\nu)$. Then $Q(n)$ satisfies the following recurrence:

$$Q(n) \leqslant \begin{cases} 1 & \text{if } n = 1, \\ r + \sum_{\nu \in C(h)} Q(n_\nu) & \text{if } n > 1, \end{cases}$$

where we sum over the set $C(h)$ of all children ν of the root such that the boundary of h crosses $t(\nu)$. Because the simplicial partition underlying the data structure has crossing number $c\sqrt{r}$, we know that the number of nodes in the set $C(h)$ is at most $c\sqrt{r}$. We also know that $n_\nu \leqslant 2n/r$ for each ν, because the simplicial partition is fine. Using these two facts one can show that with our choice of r the recurrence for $Q(n)$ solves to $O(n^{1/2+\varepsilon})$. $\square$

You may be a bit disappointed by the query time: the query time of most geometric data structures we have seen up to now is $O(\log n)$ or a polynomial in $\log n$, whereas the query time for the partition tree is around $O(\sqrt{n})$. Apparently this is the price we have to pay if we want to solve truly 2-dimensional query problems, such as half-plane range counting. Is it impossible to answer such queries in logarithmic time? No: later in this chapter we shall design a data structure for half-plane range queries with logarithmic query time. But the improvement in query time will not come for free, as that data structure will need quadratic storage.

It is helpful to compare the approach we have taken here with the range trees from Chapter 5 and the segment trees from Chapter 10. In these data structures, we would like to return information about a subset of a given set of geometric objects (points in range trees and partition trees, intervals in segment trees), or to report the subset itself. If we could precompute the requested information for every possible subset that can appear in a query, queries could be answered very fast. However, the number of possible different answers often prohibits such an approach. Instead, we identified what we have called *canonical subsets*, and we precomputed the required information for these subsets only. A query is then solved by expressing the answer to the query as the disjoint union of some of these canonical subsets. The query time is roughly linear in the number of canonical subsets that are required to express any possible query subset. The storage is proportional to the total number of precomputed canonical subsets for range trees and partition trees, and proportional to the sum of the sizes of the precomputed canonical subsets for segment trees. There is a trade-off between query time and storage: to make sure that every possible query can be expressed as the union of only a few canonical subsets, we need to provide a large repertoire of such subsets, and need a lot of storage. To decrease the storage, we need to decrease the number of precomputed canonical subsets—but that may mean that the number of canonical subsets needed to express a given query will be larger, and the query time increases.

This phenomenon can be observed clearly for 2-dimensional range searching: the partition tree we constructed in this section provides a repertoire of only $O(n)$ canonical subsets, and needs only linear storage, but in general one needs $\Omega(\sqrt{n})$ canonical subsets to express the set of points lying in a half-plane. Only by providing roughly a quadratic number of canonical subsets can one achieve logarithmic query time.

Now let's go back to the problem that we wanted to solve, namely triangular range queries. Which modifications do we need if we want to use partition trees for triangles instead of half-planes as query regions? The answer is simple: none. We can use exactly the same data structure and query algorithm, with the query half-plane replaced by a query triangle. In fact, the solution works for any query range γ. The only question is what happens to the query time.

When the query algorithm visits a node, there are three types of children: the children ν for which $t(\nu)$ lies completely inside the query range, the children for which $t(\nu)$ lies outside the range, and the children for which $t(\nu)$ lies partially inside the query range. Only the children of the third type have to be visited recursively. The query time therefore depends on the number of triangles in the partition that are crossed by the boundary of the query range γ. In other words, we have to see what the crossing number of γ is with respect to the simplicial partition. For a triangular query region, this is easy: a triangle in the partition is crossed by the boundary of γ only if it is crossed by one of the three lines through the edges of γ. Since each one of these lines intersects at most $c\sqrt{r}$ triangles, the crossing number of γ is at most $3c\sqrt{r}$.

The recursion for the query time therefore remains nearly the same, only the constant c changes to $3c$. As a result, we will need to choose r larger, but in the end the query time remains asymptotically the same. We get the following theorem:

Theorem 16.4 *Let S be a set of n points in the plane. For any $\varepsilon > 0$, there is a data structure for S, called a partition tree, that uses $O(n)$ storage, such that the points from S lying inside a query triangle can be counted in $O(n^{1/2+\varepsilon})$ time. The points can be reported in $O(k)$ additional time, where k is the number of reported points. The structure can be constructed in $O(n^{1+\varepsilon})$ time.*

Proof. The only two issues that have not been discussed yet are the construction time and the reporting of points.

Constructing a partition tree is easy: the recursive definition given before immediately implies a recursive construction algorithm. We denote the time this algorithm needs to construct a partition tree for a set of n points by $T(n)$. Let $\varepsilon > 0$ be given. According to Theorem 16.1 we can construct a fine simplicial partition for S of size r with crossing number $O(\sqrt{r})$ in time $O(n^{1+\varepsilon'})$, for any $\varepsilon' > 0$. We let $\varepsilon' = \varepsilon/2$. Hence, $T(n)$ satisfies the recurrence

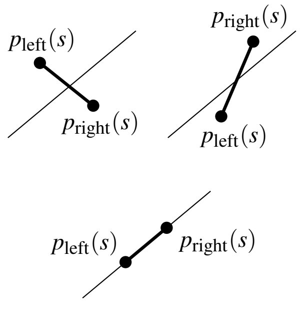
$$T(n) = \begin{cases} O(1) & \text{if } n = 1, \\ O(n^{1+\varepsilon/2}) + \sum_{\nu} T(n_{\nu}) & \text{if } n > 1, \end{cases}$$

where we sum over all children ν of the root of the tree. Because the classes in a simplicial partition are disjoint, we have $\sum_{\nu} n_{\nu} = n$, and the recurrence solves to $T(n) = O(n^{1+\varepsilon})$.

It remains to show that the k points in a query triangle can be reported in $O(k)$ additional time. These points are stored in the leaves below the selected nodes. Hence, they can be reported by traversing the subtrees rooted at the selected nodes. Because the number of internal nodes of a tree is linear in the number of leaves of that tree when each internal node has degree at least two, this takes time linear in the number of reported points. $\square$

16.2 Multi-Level Partition Trees

Partition trees are powerful data structures. Their strength is that the points lying in a query half-plane can be selected in a small number of groups, namely, the canonical subsets of the nodes selected by the query algorithm. In the example above we used a partition tree for half-plane range counting, so the only information that we needed about the selected canonical subsets was their cardinality. In other query applications we will need other information about the canonical subsets, so we have to precompute and store that data. The information that we store about a canonical subset does not have to be a single number, like its cardinality. We can also store the elements of the canonical subset in a list, or a tree, or any kind of data structure we like. This way we get a *multi-level data structure*. The concept of multi-level data structures is not new: we already used it in Chapter 5 to answer multi-dimensional rectangular range queries and in Chapter 10 for windowing queries.



We now give an example of a multi-level data structure based on partition trees. Let S be a set of n line segments in the plane. We want to count the number of segments intersected by a query line ℓ. Let $p_{\text{right}}(s)$ and $p_{\text{left}}(s)$ denote the right and left endpoint of a segment s, respectively. A line ℓ intersects s if and only if either the endpoints of s lie to distinct sides of ℓ, or s has an endpoint on ℓ. We show how to count the number of segments $s \in S$ with $p_{\text{right}}(s)$ lying above ℓ and $p_{\text{left}}(s)$ lying below ℓ. The segments with an endpoint on ℓ, and the ones with $p_{\text{right}}(s)$ lying below ℓ and $p_{\text{left}}(s)$ above ℓ, can be counted with a similar data structure. Here we choose—for a vertical line ℓ—the left side to be below ℓ and the right side above ℓ.

The idea of the data structure is simple. We first find all the segments $s \in S$ such that $p_{\text{right}}(s)$ lies above ℓ. In the previous section we saw how to use a partition tree to select these segments in a number of canonical subsets. For each of the selected canonical subsets we are interested in the number of segments s with $p_{\text{left}}(s)$ below ℓ. This is a half-plane range counting query, which can be answered if we store each canonical subset in a partition tree. Let's describe this solution in a little more detail. The data structure is defined as follows. For a set S' of segments, let $P_{\text{right}}(S') := \{p_{\text{right}}(s) : s \in S'\}$ be the set of right endpoints of the segments in S', and let $P_{\text{left}}(S') := \{p_{\text{left}}(s) : s \in S'\}$ be the set of left endpoints of the segments in S'.

- The set $P_{\text{right}}(S)$ is stored in a partition tree $\mathcal{T}$. The canonical subset of a node ν of $\mathcal{T}$ is denoted $P_{\text{right}}(\nu)$. The set of segments corresponding to the left endpoints in $P_{\text{right}}(\nu)$ is denoted $S(\nu)$, that is, $S(\nu) = \{s : p_{\text{right}}(s) \in P_{\text{right}}(\nu)\}$. (Abusing the terminology slightly, we sometimes call $S(\nu)$ the canonical subset of ν.)

- With each node ν of the first-level tree $\mathcal{T}$, we store the set $P_{\text{left}}(S(\nu))$ in a second-level partition tree $\mathcal{T}_{\nu}^{\text{assoc}}$ for half-plane range counting. This partition tree is the *associated structure* of ν.

With this data structure we can select the segments $s \in S$ with $p_{\text{right}}(s)$ above ℓ and $p_{\text{left}}(s)$ below ℓ in a number of canonical subsets. The query algorithm for this is described below. To count the number of such segments, all we have to do is sum the cardinalities of the selected subsets. Let $\mathcal{T}_{\nu}$ denote the subtree of $\mathcal{T}$ rooted at ν.

```
Algorithm SELECTINTSEGMENTS($\ell, \mathcal{T}$)
Input. A query line $\ell$ and a partition tree or subtree of it.
Output. A set of canonical nodes for all segments in the tree that are intersected
   by $\ell$.
1.  $\Upsilon \leftarrow \emptyset$
2.  if $\mathcal{T}$ consists of a single leaf $\mu$
3.    then if the segment stored at $\mu$ intersects $\ell$ then $\Upsilon \leftarrow \{\mu\}$
4.    else for each child $\nu$ of the root of $\mathcal{T}$
5.             do if $t(\nu) \subset \ell^+$
6.                  then $\Upsilon \leftarrow \Upsilon \cup$ SELECTINHALFPLANE($\ell^-, \mathcal{T}_{\nu}^{\text{assoc}}$)
7.                  else if $t(\nu) \cap \ell \neq \emptyset$
8.                         then $\Upsilon \leftarrow \Upsilon \cup$ SELECTINTSEGMENTS($\ell, \mathcal{T}_{\nu}$)
9.  return $\Upsilon$
```

The query algorithm just given can find the segments with the right endpoint above the query line and the left endpoint below it. Interestingly, the same partition tree can be used to find the segments with the left endpoint above the query line and the right endpoint below it. Only the query algorithm has to be changed: exchange the "ℓ^+" and the "ℓ^-" and we are done.

Let's analyze our multi-level partition tree for segment intersection selection. We start with the amount of storage.

Lemma 16.5 *Let S be a set of n segments in the plane. A two-level partition tree for segment intersection selection queries in S uses $O(n \log n)$ storage.*

Proof. Let n_ν denote the cardinality of the canonical subset $S(\nu)$ in the first-level partition tree. The storage for this node consists of a partition tree for S_ν, and as we know from the previous section, it needs linear storage. Hence, the storage $M(n)$ for a two-level partition tree on n segments satisfies the recurrence

$$M(n) = \begin{cases} O(1) & \text{if } n = 1, \\ \sum_\nu [O(n_\nu) + M(n_\nu)] & \text{if } n > 1, \end{cases}$$

where we sum over all children ν of the root of the tree. We know that $\sum_\nu n_\nu = n$ and $n_\nu \leqslant 2n/r$. Since $r > 2$ is a constant the recurrence for $M(n)$ solves to $M(n) = O(n \log n)$. $\square$

Adding a second level to the partition tree has increased the amount of storage by a logarithmic factor. What about the query time? Surprisingly, the asymptotic query time does not change at all.

Lemma 16.6 *Let S be a set of n segments in the plane. For any $\varepsilon > 0$, there is a two-level partition tree for S such that for a query line ℓ we can select $O(n^{1/2+\varepsilon})$ nodes from the tree with the property that the subset of segments from S intersected by ℓ is the disjoint union of the canonical subsets of the selected nodes. The selection of these nodes takes $O(n^{1/2+\varepsilon})$ time. As a consequence, the number of intersected segments can be counted in $O(n^{1/2+\varepsilon})$ time.*

Proof. Again we use a recurrence to analyze the query time. Let $\varepsilon > 0$ be given. Let n_ν denote the cardinality of the canonical subset $S(\nu)$. Lemma 16.3 tells us that we can construct the associated structure $\mathcal{T}_\nu^{\text{assoc}}$ of node ν in such a way that the query time in $\mathcal{T}_\nu^{\text{assoc}}$ is $O(n_\nu^{1/2+\varepsilon})$. Now consider the full two-level tree $\mathcal{T}$ on S. We base this tree on a fine simplicial partition of size r with crossing number at most $c\sqrt{r}$, for $r := \lceil 2(c\sqrt{2})^{1/\varepsilon} \rceil$; such a partition exists by Theorem 16.1. Let $Q(n)$ denote the query time in the two-level tree for a set of n segments. Then $Q(n)$ satisfies the recurrence:

$$Q(n) = \begin{cases} O(1) & \text{if } n = 1, \\ O(rn^{1/2+\varepsilon}) + \sum_{i=1}^{c\sqrt{r}} Q(2n/r) & \text{if } n > 1. \end{cases}$$

With our choice of r the recurrence for $Q(n)$ solves to $O(n^{1/2+\varepsilon})$. This bound on the query time immediately implies the bound on the number of selected canonical subsets. $\square$

16.3 Cutting Trees

In the previous sections we have solved planar range searching problems with partition trees. The storage requirements of partition trees are good: they use roughly linear storage. The query time, however, is $O(n^{1/2+\varepsilon})$, and this is rather high. Can we achieve a better query time, for example $O(\log n)$, if we are willing to spend more than linear storage? To have any hope of success, we must abandon the approach of using simplicial partitions: it is not possible to construct simplicial partitions with less than $O(\sqrt{r})$ crossing number, which would be needed to achieve a query time faster than $O(\sqrt{n})$.

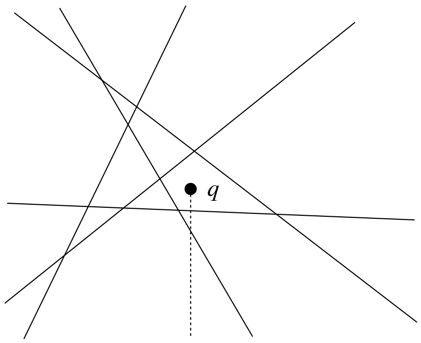


Figure 16.6
Half-plane range counting in the dual plane: how may lines are below a query point?

To come up with a new approach to the problem, we need to look at it in a different light. We apply the duality transform from Chapter 8. The first problem we solved in Section 16.1 was the half-plane range counting problem: given a set of points, count the number of points lying in a query half-plane. Let's see what we get when we look at this problem in the dual plane. Assume that the query half-plane is positive, that is, we want to count the points above the query line. In the dual plane we then have the following setting: given a set L of n lines in the plane, count the number of lines below a query point q. With the tools we constructed in the previous chapters it is easy to design a data structure with logarithmic query time for this problem: the key observation is that the number of lines below the query point q is uniquely determined by the face of the arrangement $\mathcal{A}(L)$ that contains q. Hence, we can construct $\mathcal{A}(L)$ and preprocess it for point location queries, as described in Chapter 6, and store with each face the number of lines below it. Counting the number of lines below a query point now boils down to doing point location. This solution uses $O(n^2)$ storage and it has $O(\log n)$ query time.

Note that this was a situation where we could afford to precompute the answer for every possible query—in other words, the collection of canonical subsets consists of all possible subsets that can appear. But if we go to triangular range counting, this approach is not so good: there are just too many possible triangles to precompute all possible answers. Instead, we'll try to express the set of lines below a query point by a small number of canonical subsets in a recursive way. We can then use the multi-level approach from the previous section to solve the triangular range searching problem.

We construct the whole collection of canonical subsets using a data structure called a *cutting tree*. The idea behind cutting trees is the same as for partition trees: the plane is partitioned into triangular regions, as depicted in Figure 16.7. This time, however, we require that the triangles be disjoint. How can such

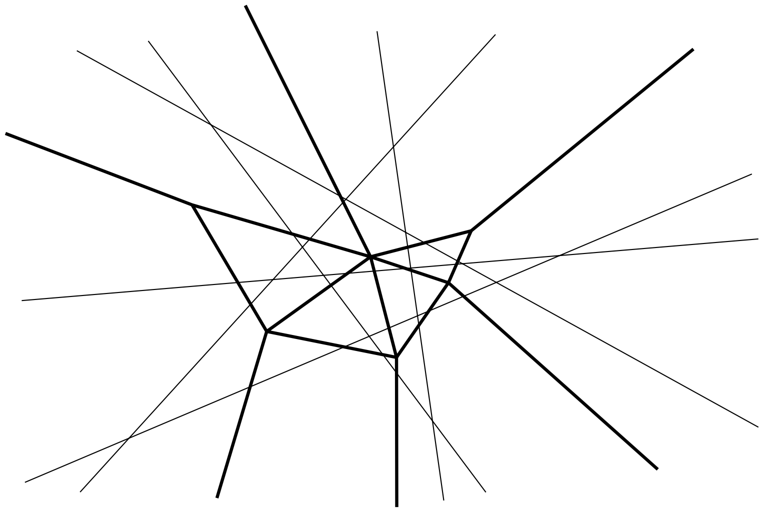


Figure 16.7
A $(1/2)$-cutting of size ten for a set of six lines

a partitioning help to count the number of lines below a query point? Let $L := \{\ell_1, \ell_2, \ldots, \ell_n\}$ be the set of lines that we obtained after dualizing the points to be preprocessed for triangular range queries. Consider a triangle t of the partitioning, and a line ℓ_i that does not intersect t. If ℓ_i lies below t, then ℓ_i lies below any query point inside t. Similarly, if ℓ_i lies above t, it lies above any query point inside t. This means that if our query point q lies in t, then the only lines of which we don't know yet whether they lie above or below q are the ones that intersect t. Our data structure will store each triangle of the partitioning, with a counter indicating the number of lines below it; for each triangle we also have a recursively defined structure on the lines intersecting it. To query in this structure, we first determine in which triangle t the query point q falls. We then compute how many lines from the ones that intersect t are below q, by recursively visiting the subtree corresponding to t. Finally, we add the number we computed in the recursive call to the number of lines below t. The efficiency of this approach depends on the number of lines intersecting a triangle: the smaller this number, the fewer lines on which we have to recurse. We now formally define the kind of partitioning we need.

Let L be a set of n lines in the plane, and let r be a parameter with $1 \leqslant r \leqslant n$. A line is said to cross a triangle if it intersects the interior of the triangle. A $(1/r)$-*cutting* for L is a set $\Xi(L) := \{t_1, t_2, \ldots, t_m\}$ of possibly unbounded triangles with disjoint interiors that together cover the plane, with the property that no triangle of the partitioning is crossed by more than n/r lines from L. The *size* of the cutting $\Xi(L)$ is the number of triangles it consists of. Figure 16.7 gives an example of a cutting.

Theorem 16.7 *For any set L of n lines in the plane, and any parameter r with $1 \leqslant r \leqslant n$, a $(1/r)$-cutting of size $O(r^2)$ exists. Moreover, such a cutting (with for each triangle in the cutting the subset of lines from L that cross it) can be constructed in $O(nr)$ time.*

In Section 16.4 references are given to the papers where this theorem is proved. We shall only concern ourselves with how cuttings can be used to design data structures. The data structure based on cuttings is called a *cutting tree*. The basic structure of a cutting tree for a set L of n lines is as follows.

- If the cardinality of L is one then the cutting tree consists of a single leaf where L is stored explicitly. The set L is the *canonical subset* of the leaf.
- Otherwise, the structure is a tree $\mathcal{T}$. There is a one-to-one correspondence between the children of the root of the tree and the triangles of a $(1/r)$-cutting $\Xi(L)$ for the set L, where r is a sufficiently large constant. (Below we shall see how r should be chosen.) The triangle of the cutting that corresponds to a child ν is denoted by $t(\nu)$. The subset of lines in L that lie below $t(\nu)$ is called the *lower canonical subset* of ν; it is denoted $L^-(\nu)$. The subset of lines in L that lie above $t(\nu)$ is called the *upper canonical subset* of ν; it is denoted $L^+(\nu)$. The subset of lines that cross $t(\nu)$ is called the *crossing subset* of $t(\nu)$. The child ν is the root of a recursively defined partition tree on its crossing subset; this subtree is denoted by $\mathcal{T}_\nu$.
- With each child ν we store the triangle $t(\nu)$. We also store information about the lower and upper canonical subsets $L^-(\nu)$ and $L^+(\nu)$; for counting the number of lines below the query point we only need to store the cardinality of the set $L^-(\nu)$, but for other applications we may store other information.

Figure 16.8 illustrates the notions of lower canonical subset, upper canonical subset, and crossing subset. We describe an algorithm for selecting the lines

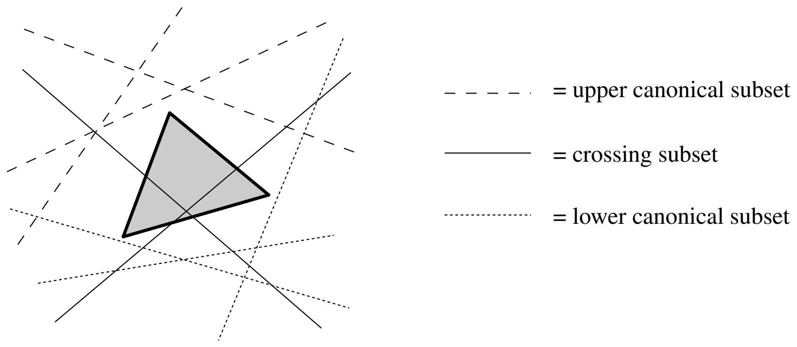


Figure 16.8
The canonical subsets and the crossing subset for a triangle

from L below a query point in a number of canonical subsets. To count the number of such lines we have to sum the cardinalities of the selected canonical subsets. Let q be the query point. The set of selected nodes will be denoted by Υ.

```
Algorithm SELECTBELOWPOINT($q, \mathcal{T}$)
Input. A query point $q$ and a cutting tree or subtree of it.
Output. A set of canonical nodes for all lines in the tree that lie below $q$.
1.  $\Upsilon \leftarrow \emptyset$
2.  if $\mathcal{T}$ consists of a single leaf $\mu$
3.    then if the line stored at $\mu$ lies below $q$ then $\Upsilon \leftarrow \{\mu\}$
4.    else for each child $\nu$ of the root of $\mathcal{T}$
5.              do Check if $q$ lies in $t(\nu)$.
6.           Let $\nu_q$ be the child such that $q \in t(\nu_q)$.
7.           $\Upsilon \leftarrow \{\nu_q\} \cup$ SELECTBELOWPOINT($q, \mathcal{T}_{\nu_q}$)
8.  return $\Upsilon$
```

Lemma 16.8 *Let L be a set of n lines in the plane. Using a cutting tree, the lines from L below a query point can be selected in $O(\log n)$ time in $O(\log n)$ canonical subsets. As a consequence, the number of such lines can be counted in $O(\log n)$ time. For any $\varepsilon > 0$, a cutting tree on L can be constructed that uses $O(n^{2+\varepsilon})$ storage.*

Proof. Let $Q(n)$ denote the query time in a cutting tree for a set of n lines. Then $Q(n)$ satisfies the recurrence:

$$Q(n) = \begin{cases} O(1) & \text{if } n = 1, \\ O(r^2) + Q(n/r) & \text{if } n > 1. \end{cases}$$

This recurrence solves to $Q(n) = O(\log n)$ for any constant $r > 1$.

Let $\varepsilon > 0$ be given. According to Theorem 16.7 we can construct a $(1/r)$-cutting for L of size cr^2, where c is a constant. We construct a cutting tree based on $(1/r)$-cuttings for $r = \lceil (2c)^{1/\varepsilon} \rceil$. The amount of storage used by the cutting tree, $M(n)$, satisfies

$$M(n) = \begin{cases} O(1) & \text{if } n = 1, \\ O(r^2) + \sum_\nu M(n_\nu) & \text{if } n > 1, \end{cases}$$

where we sum over all children ν of the root of the tree. The number of children of the root is cr^2, and $n_\nu \leqslant n/r$ for each child ν. Hence, with our choice of r the recurrence solves to $M(n) = O(n^{2+\varepsilon})$. $\square$

We conclude that we can count the number of lines below a query point in $O(\log n)$ time with a structure that uses $O(n^{2+\varepsilon})$ storage. By duality, we can do half-plane range counting within the same bounds. Now let's look at triangular range counting again: given a set S of points in the plane, count the number of points inside a query triangle. Following the approach for half-plane queries, we go to the dual plane. What problem do we get in the dual plane? The set of points dualizes to a set of lines, of course, but it is less clear what happens to the query triangle. A triangle is the intersection of three half-planes, so a point p lies in a triangle if and only if it lies in each of the half-planes. In Figure 16.9, for instance, point p lies in the triangle because $p \in \ell_1^+$ and $p \in \cap \ell_2^-$ and $p \in \cap \ell_3^-$. The line dual to p therefore has $\ell_1{}^*$ above it, and $\ell_2{}^*$ and $\ell_3{}^*$ below it. In general, the dual statement of the triangular range searching problem is: given a set L

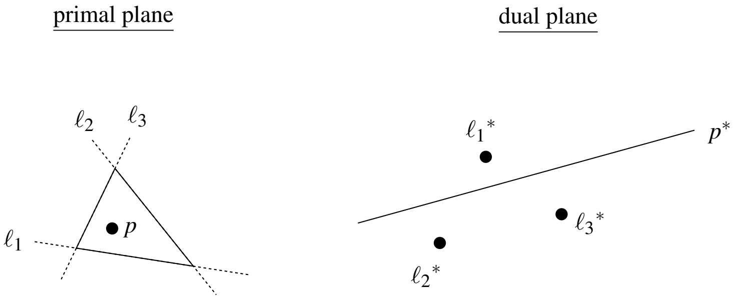


Figure 16.9
Triangular range searching

of lines in the plane, and a triple q_1, q_2, q_3 of query points labeled "above" or "below", count the number of lines from L that lie on the specified sides of the three query points. This problem can be solved with a three-level cutting tree. We now describe a data structure for the following, slightly simpler, problem: given a set L of lines, and a pair q_1, q_2 of query points, select the lines that lie below both query points. After having seen the two-level cutting tree that solves this problem, designing a three-level cutting tree for the dual of the triangular range searching problem should be easy.

A two-level cutting tree on a set L of n lines for selecting the lines below a pair q_1, q_2 of query points is defined as follows.

- The set L is stored in a cutting tree $\mathcal{T}$.
- With each node ν of the first-level tree $\mathcal{T}$, we store its lower canonical subset in a second-level cutting tree $\mathcal{T}_\nu^{\text{assoc}}$.

The idea is that the first level of the tree is used to select the lines below q_1 in a number of canonical subsets. The associated structures (or, trees at the second level) storing the selected canonical subsets are then used to select the lines that lie below q_2. Because the associated structures are one-level cuttings trees, we can use algorithm SELECTBELOWPOINT to query them. The total query algorithm is thus as follows.

```
Algorithm SELECTBELOWPAIR($q_1, q_2, \mathcal{T}$)
Input. Two query points $q_1$ and $q_2$ and a cutting tree or subtree of it.
Output. A set of canonical nodes for all lines in the tree that lie below $q_1$ and $q_2$.
1.  $\Upsilon \leftarrow \emptyset$
2.  if $\mathcal{T}$ consists of a single leaf $\mu$
3.     then if the line stored at $\mu$ lies below $q_1$ and $q_2$ then $\Upsilon \leftarrow \{\mu\}$
4.     else for each child $\nu$ of the root of $\mathcal{T}$
5.                do Check if $q_1$ lies in $t(\nu)$.
6.             Let $\nu_{q_1}$ be the child such that $q_1 \in t(\nu_{q_1})$.
7.             $\Upsilon_1 \leftarrow$ SELECTBELOWPOINT($q_2, \mathcal{T}^{\text{assoc}}_{\nu_{q_1}}$)
8.             $\Upsilon_2 \leftarrow$ SELECTBELOWPAIR($q_1, q_2, \mathcal{T}_{\nu_{q_1}}$)
9.             $\Upsilon \leftarrow \Upsilon_1 \cup \Upsilon_2$
10. return $\Upsilon$
```

Recall that adding an extra level to a partition tree did not increase its query time, whereas the amount of storage it used increased by a logarithmic factor. For cutting trees this is exactly the other way around: adding an extra level increases the query time by a logarithmic factor, whereas the amount of storage stays the same. This is proved in the next lemma.

Lemma 16.9 *Let L be a set of n lines in the plane. Using a two-level cutting tree, the lines from L below a pair of query points can be selected in $O(\log^2 n)$ time in $O(\log^2 n)$ canonical subsets. As a consequence, the number of such lines can be counted in $O(\log^2 n)$ time. For any $\varepsilon > 0$, such a two-level cutting tree on L can be constructed that uses $O(n^{2+\varepsilon})$ storage.*

Proof. Let $Q(n)$ denote the query time in a two-level cutting tree for a set of n lines. The associated structures are one-level cutting trees, so the query time for the associated structures is $O(\log n)$ by Lemma 16.8. Hence, $Q(n)$ satisfies the recurrence:

$$Q(n) = \begin{cases} O(1) & \text{if } n = 1, \\ O(r^2) + O(\log n) + Q(n/r) & \text{if } n > 1. \end{cases}$$

This recurrence solves to $Q(n) = O(\log^2 n)$ for any constant $r > 1$.

Let $\varepsilon > 0$ be given. According to Lemma 16.8 we can construct the associated structures of the children of the root such that each of them uses $O(n^{2+\varepsilon})$ storage. Hence, the amount of storage used by the cutting tree, $M(n)$, satisfies

$$M(n) = \begin{cases} O(1) & \text{if } n = 1, \\ \sum_\nu [O(n^{2+\varepsilon} + M(n_\nu))] & \text{if } n > 1, \end{cases}$$

where we sum over all children ν of the root of the tree. The number of children of the root is $O(r^2)$, and $n_\nu \leqslant n/r$ for each child ν. It follows that, if r is a large enough constant, the recurrence solves to $M(n) = O(n^{2+\varepsilon})$. (If you are a bit bored by now, you are on the right track: cutting trees, partition trees, and their multi-level variants are all analyzed in the same way.) $\square$

We designed and analyzed a two-level cutting tree for selecting (or counting) the number of lines below a pair of query points. For the triangular range searching we need a three-level cutting tree. The design and analysis of three-level cutting trees follows exactly the same pattern as for two-level cutting trees. Therefore you should hopefully have no difficulties in proving the following result.

Theorem 16.10 *Let S be a set of n points in the plane. For any $\varepsilon > 0$, there is a data structure for S, called a cutting tree, that uses $O(n^{2+\varepsilon})$ storage such that the points from S lying inside a query triangle can be counted in $O(\log^3 n)$ time. The points can be reported in $O(k)$ additional time, where k is the number of reported points. The structure can be constructed in $O(n^{2+\varepsilon})$ time.*

One can even do a little bit better than in this theorem. This is discussed in Section 16.4 and in the exercises.

16.4 Notes and Comments

Range searching is one of the best studied problems in computational geometry. For extensive overviews of results on range searching, see the surveys by Agarwal [1] and Agarwal and Erickson [2]. We can distinguish between *orthogonal range searching* and *simplex range searching*. Orthogonal range searching was the topic of Chapter 5. In this chapter we discussed the planar variant of simplex range searching, namely triangular range searching. We conclude with a brief overview of the history of simplex range searching, and a discussion of the higher-dimensional variants of the theory we presented.

We begin our discussion with data structures for simplex range searching in the plane that use roughly linear storage. Willard [388] was the first to present such a data structure. His structure is based on the same idea as the partition trees described in this chapter, namely a partition of the plane into regions. His partition, however, did not have such a good crossing number, so the query time of his structure was $O(n^{0.774})$. As better simplicial partitions were developed, more efficient partition trees were possible [111, 169, 209, 394]. Improvements were also obtained using a somewhat different structure than a partition tree, namely a spanning tree with low stabbing number [112, 384]. The best solution for triangular range searching so far has been given by Matoušek [263]. Theorem 16.1 is proved in that paper. Matoušek also describes a more complicated data structure with $O(\sqrt{n}2^{O(\log^* n)})$ query time. This structure, however, cannot be used so easily as a basis for multi-level trees.

The simplex range searching problem in $\mathbb{R}^d$ is stated as follows: preprocess a set S of points in $\mathbb{R}^d$ into a data structure, such that the points from S lying in a query simplex can be counted (or reported) efficiently. Matoušek also proved results for simplicial partitions in higher dimensions. The definition of simplicial partitions in $\mathbb{R}^d$ is similar to the definition in the plane; the only difference is that the triangles of the partition are replaced by d-simplices, and that the crossing number is defined with respect to hyperplanes instead of lines. Matoušek proved that any set of points in $\mathbb{R}^d$ admits a simplicial partition of size r with crossing number $O(r^{1-1/d})$. Using such a simplicial partition one can construct, for any $\varepsilon > 0$, a partition tree for simplex range searching in $\mathbb{R}^d$ that uses linear space and has $O(n^{1-1/d+\varepsilon})$ query time. The query time can be improved to $O(n^{1-1/d}(\log n)^{O(1)})$. The query time of Matoušek's structure comes close to the lower bounds proved by Chazelle [89], which state that a data structure for triangular range searching that uses $O(m)$ storage must have $\Omega(n/(m^{1/d}\log n))$ query time. A structure that uses linear space must thus have $\Omega(n^{1-1/d}/\log n)$ query time. (In the plane, a slightly sharper lower bound is known, namely $\Omega(n/\sqrt{m})$.)

Data structures for simplex range searching with logarithmic query time have also received a lot of attention. Clarkson [131] was the first to realize that cuttings can be used as the basis for a data structure for range searching. Using a probabilistic argument, he proved the existence of $O(\log r/r)$-cuttings of size $O(r^d)$ for sets of hyperplanes in $\mathbb{R}^d$, and he used this to develop a data structure for half-space range queries. After this, several people worked on improving the results and on developing efficient algorithms for computing cuttings. Currently the best known algorithm is by Chazelle [95]. He has shown that for any parameter r, it is possible to compute a $(1/r)$-cutting of size $O(r^d)$ with a deterministic algorithm that takes $O(nr^{d-1})$ time. These cuttings can be used to design a (multi-level) cutting tree for simplex range searching, as shown in this chapter for the planar case. The resulting data structure has $O(\log^d n)$ query time and uses $O(n^{d+\varepsilon})$ storage. The query time can be reduced to $O(\log n)$. Due to a special property of Chazelle's cuttings, it is also possible to get rid of the $O(n^{\varepsilon})$ factor in the storage [265], but for the new structure it is no longer possible to reduce the query time to $O(\log n)$. These bounds are again close to Chazelle's lower bounds.

By combining partition trees and cutting trees the right way, one can get data structures that have storage in between that of partition trees and cutting trees. In particular, for any $n \leqslant m \leqslant n^d$, a data structure of size $O(m^{1+\varepsilon})$ has $O(n^{1+\varepsilon}/m^{1/d})$ query time, close to the lower bound: Exercise 16.16 shows how to do this.

Partition trees use linear space, but their query time is rather high. Cutting trees, on the other hand, have logarithmic query time but they need a lot of storage, Ideally one would like to have a structure that uses linear space and has logarithmic query time. While Chazelle's lower bounds [89] show that this is not possible for exact range searching, one can achieve such bounds for *approximate range searching*. The idea here is that points that are "almost" in the query range (that is, that are very close to it) may be reported as well—see the survey by Duncan and Goodrich [151] for details.

In the discussion above we have concentrated on simplex range searching. Half-space range searching is, of course, a special case of this. It turns out that for this special case better results can be achieved. For example, for half-plane range reporting (not for counting) in the plane, there is a data structure with $O(\log n + k)$ query time that uses $O(n)$ storage [107]. Here k is the number of reported points. Improved results are possible in higher dimensions as well: the points in a query half-space can be reported in $O(n^{1-1/\lfloor d/2 \rfloor}(\log n)^{O(1)} + k)$ time with a data structure that uses $O(n \log\log n)$ storage [264].

Finally, Agarwal and Matoušek [8] generalized the results on range searching to query ranges that are semi-algebraic sets.

16.5 Exercises

16.1 Let S be a set of n points in the plane.

a. Suppose that the points in S lie on a $\sqrt{n} \times \sqrt{n}$ grid. (Assume for simplicity that n is a square.) Let r be a parameter with $1 \leqslant r \leqslant n$. Draw a fine simplicial partition for S of size r and crossing number $O(\sqrt{r})$.
b. Now suppose all points from S are collinear. Draw a fine simplicial partition for S of size r. What is the crossing number of your partition?

16.2 Prove that the selected nodes in a partition tree are exactly the nodes ν with the following property: $t(\nu) \subset h$ (or, in case ν is a leaf, the point stored at ν lies in h) and there is no ancestor μ of ν such that $t(\mu) \subset h$. Use this to prove that $S \cap h$ is the disjoint union of the canonical subsets of the selected nodes.

16.3 Prove that the recurrence for $M(n)$ given in the proof of Lemma 16.2 solves to $M(n) = O(n)$.

16.4 Prove that the recurrence for $Q(n)$ given in the proof of Lemma 16.3 solves to $Q(n) = O(n^{1/2+\varepsilon})$.

16.5 Suppose we have a partition tree as defined on page 339, except that the simplicial partitions used in the construction are not necessarily fine. What does this mean for the amount of storage used by the partition tree? And for its query time?

16.6 Lemma 16.3 shows that for any $\varepsilon > 0$ we can build a partition tree with $O(n^{1/2+\varepsilon})$ query time, by choosing the parameter r that determines the branching degree of the tree to be a large enough constant. We can do even better if we choose r depending on n. Show that the query time reduces to $O(\sqrt{n}\log n)$ if we choose $r = \sqrt{n}$. (Note that the value of r is not the same any more at different nodes in the tree. However, this is not a problem.)

16.7 Prove that the recurrence for $M(n)$ given in the proof of Lemma 16.5 solves to $M(n) = O(n \log n)$.

16.8 Prove that the recurrence for $Q(n)$ given in the proof of Lemma 16.6 solves to $Q(n) = O(n^{1/2+\varepsilon})$.

16.9 Let T be a set of n triangles in the plane. An *inverse range counting query* asks to count the number of triangles from T containing a query point q.

a. Design a data structure for inverse range counting queries that uses roughly linear storage (for example, $O(n \log^c n)$ for some constant c). Analyze the amount of storage and the query time of your data structure.
b. Can you do better if you know that all triangles are disjoint?

16.10 Let L be a set of n lines in the plane.

a. Suppose that L consists of $\lfloor n/2 \rfloor$ vertical lines and $\lceil n/2 \rceil$ horizontal lines. Let r be a parameter with $1 \leqslant r \leqslant n$. Draw a $(1/r)$-cutting for L of size $O(r^2)$.
b. Now suppose all lines from L are vertical. Draw a $(1/r)$-cutting for L. What is the size of your cutting?

16.11 Prove that the recurrences for $Q(n)$ and $M(n)$ given in the proof of Lemma 16.8 solve to $Q(n) = O(\log n)$ and $M(n) = O(n^{2+\varepsilon})$.

16.12 Prove that the recurrences for $Q(n)$ and $M(n)$ given in the proof of Lemma 16.9 solve to $Q(n) = O(\log^2 n)$ and $M(n) = O(n^{2+\varepsilon})$.

16.13 A query in a two-level cutting tree visits the associated structures of the nodes on one path in the tree. The query time in an associated structure storing m lines is $O(\log m)$ by Lemma 16.8. Because the depth of the main tree is $O(\log n)$, the total query time is $O(\log^2 n)$. If we could choose the value of the parameter r of the main cutting tree larger than constant, for example n^δ for some small $\delta > 0$, then the depth of the main tree would become smaller, resulting in a reduction of the query time. Unfortunately, there is an $O(r^2)$ term in the recurrence for $Q(n)$ in the proof of Lemma 16.9.

a. Describe a way to get around this problem, so that you can choose $r := n^\delta$.
b. Prove that the query time of your two-level data structure is $O(\log n)$.
c. Prove that the amount of storage of the data structure is still $O(n^{2+\varepsilon})$.

16.14 Design a data structure for triangular range searching that has $O(\log^3 n)$ query time. Describe the data structure as well as the query algorithm precisely, and analyze both storage and query time.

16.15 Let S be a set of n points in the plane, each having a positive real weight associated with them. Describe two data structures for the following query problem: find the point in a query half-plane with the largest weight. One data structure should use linear storage, and the other data structure should have logarithmic query time. Analyze the amount of storage and the query time of both data structures.

16.16 In this chapter we have seen a data structure for half-plane range searching with linear storage but a rather high query time (the partition tree) and a data structure with logarithmic query time but a rather high use of storage (the cutting tree). Sometimes one would like to have something in between: a data structure that has a better query time than partition trees, but uses less storage than a cutting tree. In this exercise we show how to design such a structure.

Suppose that we have $O(m^{1+\varepsilon})$ storage available, for some m between n and n^2. We want to build a structure for selecting points in half-planes that uses $O(m^{1+\varepsilon})$ storage and has as fast a query time as possible. The idea is to start with the fastest structure we have (the cutting tree) and switch to the slow structure (the partition tree) when we run out of storage. That is, we continue the construction of the cutting tree recursively until the number of lines we have to store drops below some threshold $\hat{n}$.

a. Describe the data structure and the query algorithm in detail.
b. Compute the value of the threshold $\hat{n}$ such that the amount of storage is $O(m^{1+\varepsilon})$.
c. Analyze the query time of the resulting data structure.